

# Microprocessor Final Design Document

Steven Bell

16 December 2010

CENG-3013

Oklahoma Christian University

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Requirements	4
<b>2</b>	<b>General Architecture Notes</b>	<b>5</b>
2.1	Chip-level IO	5
2.2	Instruction set	6
2.2.1	NOP	6
2.2.2	ADD	6
2.2.3	SUB	6
2.2.4	MUL	6
2.2.5	AND	6
2.2.6	OR	6
2.2.7	LSR	6
2.2.8	LSL	7
2.2.9	ASR	7
2.2.10	ASL	7
2.2.11	COMP	7
2.2.12	NEG	7
2.2.13	LOAD	7
2.2.14	STORE	7
2.2.15	MOVE	7
2.2.16	Jump	7
2.2.17	HALT	8
<b>3</b>	<b>Module design</b>	<b>8</b>
3.1	Generic 16-bit register	8
3.1.1	Design	8
3.1.2	Testing	8
3.2	Program counter	9
3.2.1	Design	9
3.2.2	Testing	9
3.3	General-purpose register block	9
3.3.1	Design	9
3.3.2	Testing	10
3.4	Arithmetic logic unit (ALU)	10
3.4.1	Design	11
3.4.2	Testing	11
3.5	ALU Latch	11
3.5.1	Design	11
3.5.2	Testing	12
3.6	Datapath	12
3.6.1	Design	12
3.6.2	Testing	13
3.7	Control module	14
3.7.1	State machine	14
3.7.2	Control signal translation	16
3.8	Address multiplexer	18
3.9	Memory IO	18
3.9.1	Design	19
3.9.2	Testing	19
3.10	Complete chip	19
3.10.1	Design	19

3.10.2 Testing . . . . .	20
<b>4 Hardware implementation</b>	<b>20</b>
4.1 Motivation and design criteria . . . . .	20
4.2 Part selection . . . . .	21
4.3 PCB Design . . . . .	21
4.4 Breadboard layout . . . . .	21
4.4.1 Pin mapping . . . . .	23
<b>5 Assembler</b>	<b>23</b>
<b>6 Conclusion</b>	<b>23</b>
6.1 Current status . . . . .	23
6.2 Future improvements . . . . .	24
<b>A Verilog module code</b>	<b>25</b>
A.1 Global constants . . . . .	25
A.2 Generic 16-bit register . . . . .	26
A.3 Program counter . . . . .	27
A.4 ALU . . . . .	27
A.5 ALU latch . . . . .	30
A.6 General-purpose register block . . . . .	31
A.7 Datapath . . . . .	32
A.8 Address Multiplexer . . . . .	33
A.9 Memory IO . . . . .	34
A.10 Control module state machine . . . . .	35
A.11 Control signals translator . . . . .	37
A.12 CPU . . . . .	40
<b>B Verilog testbench code</b>	<b>43</b>
B.1 Generic 16-bit register . . . . .	43
B.2 Program counter . . . . .	44
B.3 ALU . . . . .	46
B.4 ALU latch . . . . .	51
B.5 General-purpose register block . . . . .	52
B.6 Datapath . . . . .	55
B.7 Control module state machine . . . . .	61
B.8 Control signals translator . . . . .	65
B.9 CPU . . . . .	73
<b>C Output waveforms</b>	<b>76</b>
<b>D PCB diagrams</b>	<b>110</b>
D.1 Eagle schematic . . . . .	110
D.2 Eagle layout . . . . .	111
<b>E Assembler code</b>	<b>112</b>
<b>F Final test program</b>	<b>130</b>
<b>G Logic analyzer captures</b>	<b>135</b>

# 1 Introduction

The objective of this course (CENG-3013, Integrated Circuit Design) is to design an 8-bit microprocessor, model and simulate it using the Verilog hardware description language, and finally to implement it in hardware using a programmable logic device (PLD). This document describes my microprocessor design, the test code used to verify it, and the physical hardware implementation.

Three things make my work unique:

- Rather than using ABEL test vectors to test each Verilog module, I wrote all of my testbench code in Verilog. This had several advantages.
  - Using an all-Verilog approach enabled me to use tools other than Lattice ispLEVER Classic, most of which were far superior. I primarily used Icarus Verilog (<http://www.icarus.com/eda/verilog/>) combined with a Bash build script to automate the build and test process for all of the modules. Rather than having to build and test each module independently, I was able to run all of the tests at once and see the pass/fail results immediately. The build script also performed macro substitution, so a test failure message could refer to a specific line of code, rather than merely printing out a simulation timestep.
  - Verilog testbenches are vastly more flexible and efficient for testing large and complex modules. The CPU testbench simulates a block of ROM, which is loaded from a separate file. This makes it very easy to write additional test vectors and to copy the test code into the physical EEPROM. Likewise, the control module test consists of a short segment of Verilog code coupled with an easy-to-read text file containing the test process. This type of high-level testing made it extremely easy to add and edit tests.
  - Both Icarus Verilog and Aldec ActiveHDL perform functional simulation rather than gate-level simulation. This let me focus first on whether the code worked correctly, rather than puzzling over why the optimizer had removed my registers or combined all of my signals. Once the code worked, I could synthesize it with Synplicity and work out synthesis bugs.
- I constructed my own board using a Lattice MachXO PLD instead of using an LSI5512. The latter are no longer available, and have been replaced by the costly and already-outdated Mach4000 series. The MachXO parts are much less expensive, and (until the upcoming release of the MachXO2) are Lattice's flagship CPLD, with solid tool support and documentation.
- To circumvent the painstaking process of creating hexadecimal opcodes for the final test program, I wrote an assembler for my microprocessor which parses an input file and creates output files for Verilog simulation and EEPROM programming.

## 1.1 Requirements

The requirements in this section are taken from the project specification posted at the beginning of the semester:

- 8-bit data bus
- 16-bit address bus
- Eight 8-bit general purpose registers which can be used in pairs as four 16-bit registers.
- Instructions formatted as shown in Table 1.
- Instruction set as shown in Table 2.
- Jump condition codes as shown in Table 3.

Table 1: Instruction format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode				Source type			Destination type		Source register		Destination register			Condition	

Table 2: Required instruction set

Instruction	Opcode	Operands
NOP	0x00	-
Add	0x01	reg, reg/imm
Subtract	0x02	reg, reg/imm
Multiply	0x03	reg, reg/imm
Logical AND	0x04	reg, reg/imm
Logical OR	0x05	reg, reg/imm
Logical shift right	0x06	reg
Logical shift left	0x07	reg
Complement bits	0x08	reg
Load	0x10	reg, imm/address
Store	0x11	address, reg
Move	0x12	address, address
Jump	0x0F	address, condition
Halt	0x1F	-

Table 3: Jump condition codes

Condition	Bit designation
Always	00
Carry	01
Zero	10
Negative	11

## 2 General Architecture Notes

Where multi-byte values are used, they are stored in big-endian format, with the bits ordered from most significant to least significant. The processor has an 8-bit internal data bus which transfers data between the individual modules and connects to external RAM and ROM.

### 2.1 Chip-level IO

- Clock - Square wave input from function generator
- Reset - Active low input from switch
- Address bus - 16-bit output bus
- Data bus - 8-bit bidirectional bus
- ROM enable - Active low signal which enables the EEPROM
- RAM enable - Active low signal which enables the SRAM
- Memory write - Active high signal which switches the direction of the bidirectional buffer
- $\overline{\text{Write}}$  - Active low signal which enables writing to SRAM

## 2.2 Instruction set

This section briefly describes each of the instructions. Operations are considered to be 8-bit operations with 8-bit results, except for the multiply, which produces a 16-bit result. For each operation, the zero flag is set if the result contains all zeros.

The negative flag mirrors the most significant bit in the result. In the two's complement number system, this is equivalent to telling whether the value is positive or negative. Mathematical operations are performed exactly the same as for unsigned values, and the programmer can even choose to ignore the negative flag and work with the unsigned values (Wakerly, J. "Digital Design Principles and Practices, 4th Ed.", pp 37-43). In the event that a subtraction causes a borrow, the carry bit will be set, and this can be used to determine if a larger value was subtracted from a smaller value. This design is patterned after the operation of the negative flag in the Motorola M68HC11 ([www.freescale.com/files/microcontrollers/doc/ref\\_manual/M68HC11RM.pdf](http://www.freescale.com/files/microcontrollers/doc/ref_manual/M68HC11RM.pdf), page 204).

### 2.2.1 NOP

No operation. After reading this instruction, the processor continues immediately to the next instruction without any extra clock cycles.

### 2.2.2 ADD

Adds a value to a register. The source value can be another register, a memory location, or an 8-bit immediate. The carry flag is set if a one was carried out of the highest bit. The negative flag is set if the highest bit is a 1, and the zero flag is set if the result is zero.

### 2.2.3 SUB

Subtracts a value from a register. The source value can be another register, a memory location, or an 8-bit immediate. The carry flag is set if a one was borrowed. The negative flag is set if the highest bit is a 1, and the zero flag is set if the result is zero. Note that because the values are only 8 bits wide, 0x00 - 0x81 gives 0x80, which is not a negative number. Thus, the negative flag will not be set, but the carry bit will.

### 2.2.4 MUL

Multiplies an 8-bit register by an 8-bit value, producing a 16-bit result. The source value can be another register, a memory location, or an immediate. The result will be written to the destination register and its 16-bit pair, i.e, multiplying register a by register c will replace both a and b with the result. Best practice is to load both operands into a single register pair, so that no registers with other data are accidentally overwritten.

### 2.2.5 AND

Performs a bitwise AND between a register and a value. The source value can be another register, a memory location, or an 8-bit immediate. The negative flag is set if the highest bit is a 1, and the zero flag is set if the result is zero.

### 2.2.6 OR

Performs a bitwise OR between a register and a value. The source value can be another register, a memory location, or an 8-bit immediate. The negative flag is set if the highest bit is a 1, and the zero flag is set if the result is zero.

### 2.2.7 LSR

Performs a logical right-shift on a register and stores the result in the same register. All eight bits are shifted, and a zero is shifted into the top bit position. The carry flag is set if a 1 was shifted out. The negative flag is set if the highest bit is a 1, and the zero flag is set if the result is zero.

### 2.2.8 LSL

Performs a logical left-shift on a register and stores the result in the same register. All eight bits are shifted, and a zero is shifted into the lowest bit position. The carry flag is set if a 1 was shifted out. The negative flag is set if the highest bit is a 1, and the zero flag is set if the result is zero.

### 2.2.9 ASR

Performs an arithmetic right-shift on a register. Only the lower seven bytes are shifted; a copy of the sign bit is shifted into the bit position below the sign bit. The carry flag is set if a 1 was shifted out. The negative flag is set if the highest bit is a 1, and the zero flag is set if the result is zero.

### 2.2.10 ASL

Performs an arithmetic left-shift on a register. Only the lower seven bytes are shifted; a zero is shifted into the lowest bit position. The carry flag is set if a 1 was shifted out. The negative flag is set if the highest bit is a 1, and the zero flag is set if the result is zero.

### 2.2.11 COMP

Complements all of the bits in a register and stores the result in the same register. The negative flag is set if the highest bit is a 1, and the zero flag is set if the result is zero.

### 2.2.12 NEG

Performs a 2's complement (binary negation) on a register and stores the result in the same register. The negative flag is set if the highest bit is a 1, and the zero flag is set if the result is zero.

### 2.2.13 LOAD

Loads a value into a register. The source value can be a memory location or an immediate. The source can also be another register, which is also handled internally with the LOAD opcode. The assembler uses the mnemonic "COPY" to refer to loading one register into another.

### 2.2.14 STORE

Stores a register to RAM. A memory address is required.

### 2.2.15 MOVE

Copies the contents of one memory address to another. Two addresses are required; the register contents are not affected.

### 2.2.16 Jump

A jump relocates the current program counter to the memory location specified. After the jump completes, the processor immediately continues executing instructions at the new memory location.

The processor can jump:

- Always, regardless of ALU flags. This uses the assembler mnemonic "JMP"
- If the ALU carry bit was set in the last ALU operation. This uses the assembler mnemonic "JCAR"
- If the ALU zero bit was set in the last ALU operation. This uses the assembler mnemonic "JZERO"
- If the ALU negative bit was set in the last ALU operation. This uses the assembler mnemonic "JNEG"

### 2.2.17 HALT

Halts the processor's operation. No more instructions are read and the processor stays in the HALT state until it is reset.

## 3 Module design

### 3.1 Generic 16-bit register

A 16-bit register is used three times in the design: the jump register, the memory address register, and the instruction register. The same module is instantiated in all three of these cases.

#### 3.1.1 Design

The module has an 8-bit input, which in all three instantiations comes from the data bus. Two signals, `setHigh` and `setLow` determine whether this input is stored in the top half or bottom half of the register.

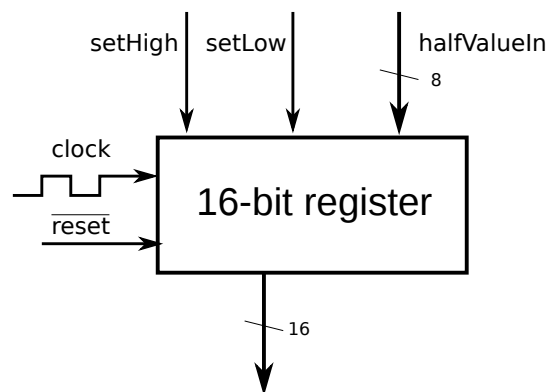


Figure 3.1: 16-bit register block diagram

The module code is shown in Appendix [A.2 on page 26](#).

#### 3.1.2 Testing

Since this was the first module coded, the test code was written in ABEL. It used the following procedure:

1. Reset the module, and ensure that the output is zero.
2. Load the high half of the register
3. Load the low half of the register
4. Load the high half while performing a reset. The reset should take precedence and the output should be zero.

The testbench code is shown in Appendix [B.1 on page 43](#).

## 3.2 Program counter

### 3.2.1 Design

The program counter holds the address of the next instruction byte and is used to index ROM when fetching instructions. It increases the output count by one on the rising edge of the clock when the `increment` signal is high. When the `set` signal is asserted, it loads the value from the jump register through the input `newCount`.

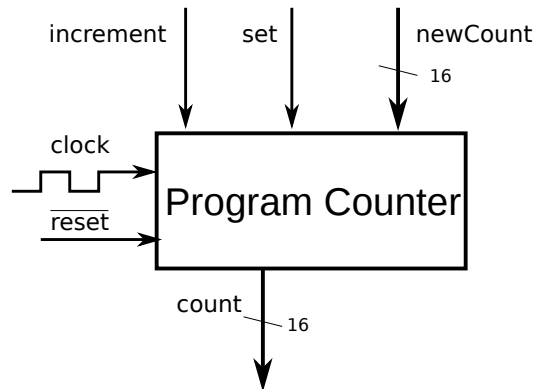


Figure 3.2: Program counter block diagram

The module code is shown in [Appendix A.3 on page 27](#).

### 3.2.2 Testing

The test uses the following procedure:

1. Perform a reset and check that the output is zero.
2. Leave the increment signal low and ensure that the counter does not increment and remains at zero.
3. Load the value `0xFFFFE` into the counter and check that the output equals `0xFFFFE` and was not incremented on this clock cycle.
4. Increment the counter and check that it equals `0xFFFF`.
5. Increment the counter again and check that it rolls over to `0x0000`.

The testbench code is shown in [Appendix B.2 on page 44](#).

## 3.3 General-purpose register block

The general-purpose register block contains 8 eight-bit registers which are used for data manipulation. They are grouped into pairs, creating 4 sixteen-bit registers which can receive the result of a multiply operation.

### 3.3.1 Design

Values are only stored on the rising edge of the clock, but the outputs are set via combinational logic. This means that the outputs are available immediately so register-to-register copies and ALU operations from the registers take place immediately without having to wait an additional clock cycle for the data to become available.

The register block has three 3-bit address inputs which are used to index the registers. The `input_select` input determines which register receives a value from the data bus if `read_data` is high; `output_select` determines which register is written when `write_data` is high. The `alu_output_value` bus goes directly to the ALU and is always active. The `alu_output_select` address determines which register is sent directly to the ALU on this bus.

The register pairing is not due to any hardware feature within the register block - the registers are implemented simply as an 8-element array of 8-bit registers. The pairing is performed by the control signals module, which

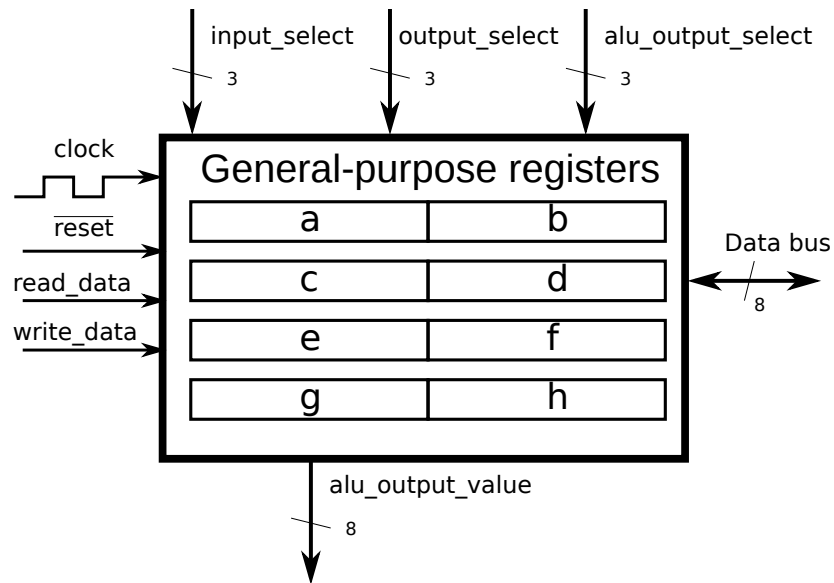


Figure 3.3: General-purpose registers block diagram

The module code is shown in [Appendix A.6 on page 31](#).

### 3.3.2 Testing

The test uses the following procedure:

1. Perform a reset.
2. Read all of the registers out and check that they read zero.
3. Sequentially load values into each of the registers.
4. Iterate back through the registers, read out their values, and check that they match what was stored.

The testbench code is shown in [Appendix B.5 on page 52](#)

## 3.4 Arithmetic logic unit (ALU)

The arithmetic logic unit implements all of the arithmetic operations specified: addition, subtraction, multiplication, logical AND and OR, left and right logical shifts, left and right arithmetic shifts, bitwise complement, and negation. It also contains a passthrough instruction so that the ALU latch can be used as a temporary register for the MOVE operation. Output flags are set based on the results of the operation.

### 3.4.1 Design

The ALU consists entirely of combinational logic and operations are performed whenever the inputs change. This wastes a tiny bit of power due to unnecessary gate switching, but simplifies the design. The output is only 8 bits, except for the multiply, which is 16 bits. The zero flag is defined for every operation: if the result is all zeros, the flag is set. Likewise, the negative flag is set whenever the highest bit of the result is a 1, which indicates a negative number in the two's complement system. The behavior and meaning of the carry flag is dependant on the operation. For add and subtract, it indicates a carry out or borrow in; for shifts, it indicates the bit that was shifted out.

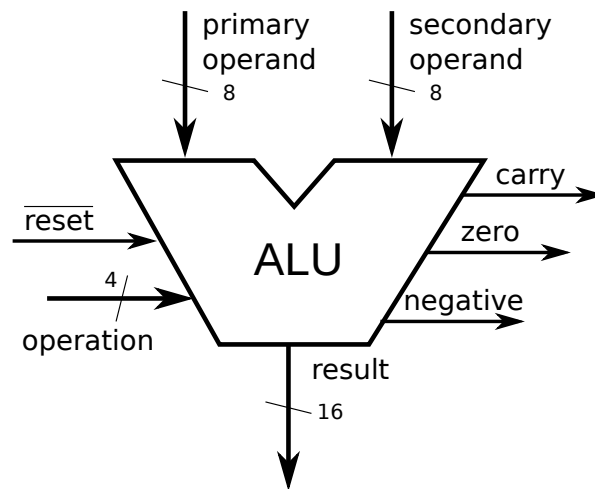


Figure 3.4: ALU block diagram

The ALU module code is shown in [Appendix A.4 on page 27](#).

### 3.4.2 Testing

The ALU test consists of running a set of possible inputs and checking the result and the output flags. Each instruction is tested with several operations. The complete code is shown in [Appendix B.3 on page 46](#).

## 3.5 ALU Latch

The ALU latch grabs the result of the ALU operation, holds it, and then puts it on the databus when the store signals are asserted. It also latches the flags, so that a jump operates based on the last time the result was grabbed.

### 3.5.1 Design

The ALU latch uses a simple sequential design. The `alu_result` and `flags` are stored on the rising edge of the clock if `grab` is high. Combinational logic is used to determine which half of the stored value is put out to the data bus. If neither store signal is high, the output is high-z. The `flags_out` output is always enabled.

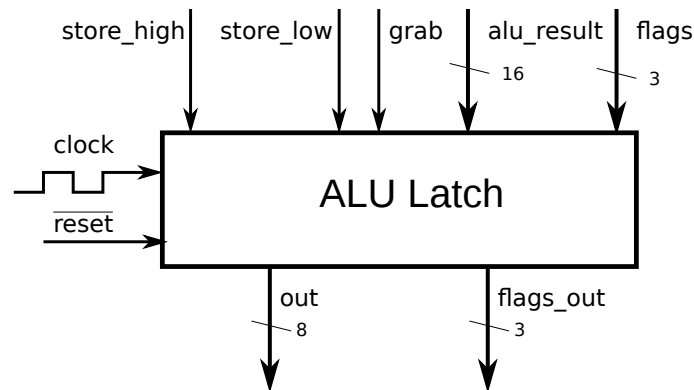


Figure 3.5: ALU latch block diagram

### 3.5.2 Testing

The test uses the following procedure:

1. Perform a reset.
2. Grab a value from the `alu_result` bus and hold it without putting the value on the data bus. The output should be high-z.
3. Put the high half of the value onto the data bus.
4. Put the low half of the value onto the data bus.
5. Grab a new value and put the result onto the data bus immediately.

Note that although the test passes as expected, the simulation ends as soon as the last test finishes so the final signal state is not shown on the waveform.

## 3.6 Datapath

### 3.6.1 Design

The datapath module combines the program counter, jump register, general-purpose registers, ALU, ALU latch, memory address register, and instruction register into a single unit connected by a data bus. The data bus is a bidirectional module port, so data can be brought in and out of the chip. A block diagram of the datapath is shown in Figure 3.6.

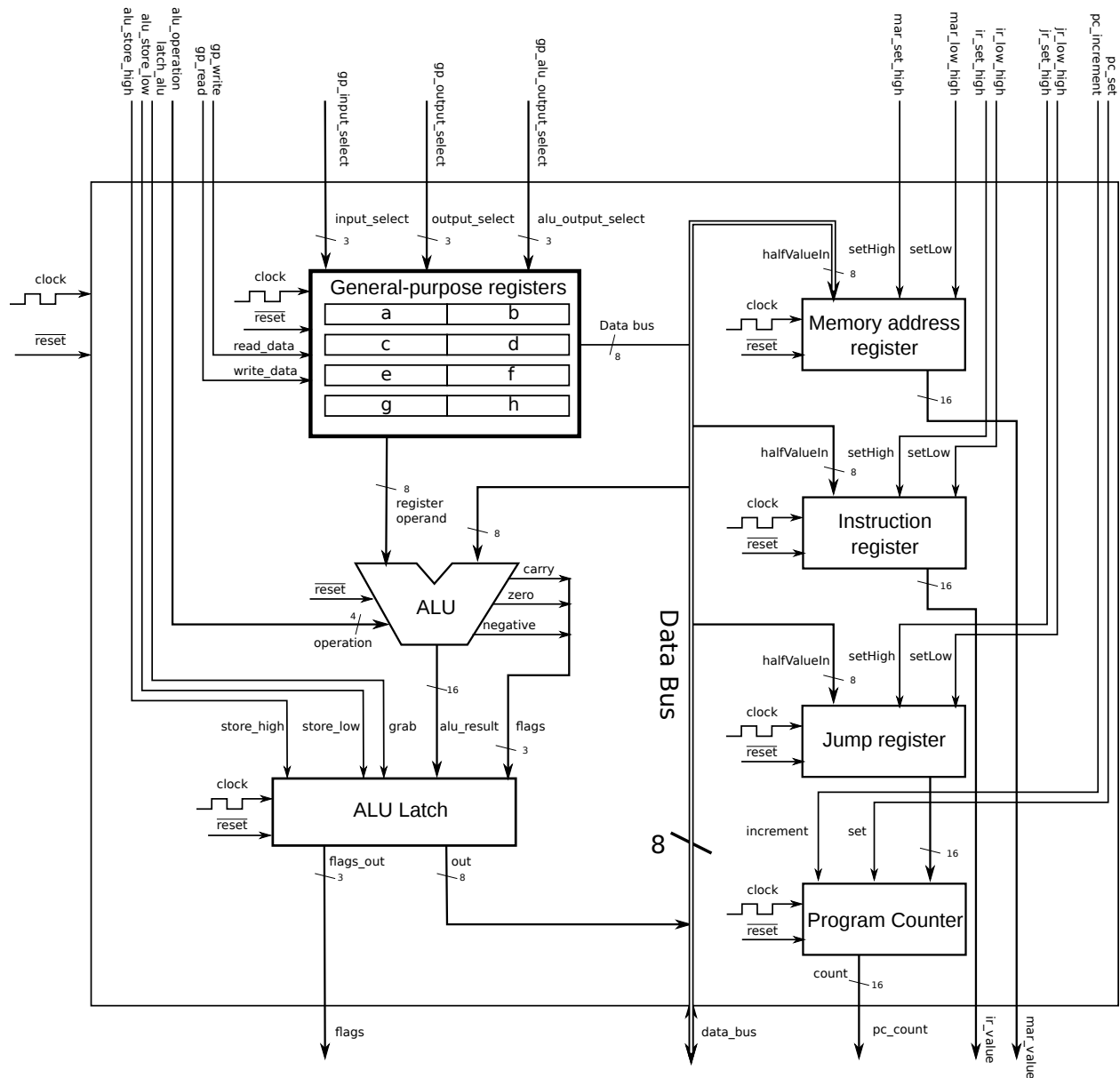


Figure 3.6: Datapath block diagram

### 3.6.2 Testing

The test uses the following procedure:

1. Perform a reset.
2. Load instruction register from the data bus (high and low).
3. Loading the jump register from the data bus (high and low).
4. Load program counter from jump register.
5. Iterate through each register and load a value from the data bus.
6. Iterate back through the registers and write it back to the data bus.

7. Load the memory address register from the data bus.
8. Load memory address register from the general-purpose registers.
9. Perform an ALU binary operation using two general-purpose registers.
10. Perform an ALU binary operation with an immediate.
11. Perform an ALU unary operation with a general-purpose register.
12. Perform an ALU passthrough and latch the ALU result.
13. Store the ALU result to data bus.

Reset for the general purpose registers is not tested.

### 3.7 Control module

The control module consists of two separate modules: a state machine which reads the output of the instruction register and determines what to do on the next clock cycle, and a signal translation module which maps the control state into controls signals for all of the other modules.

#### 3.7.1 State machine

**Design** The state diagram for the state machine is shown in Figure 3.8. The states are all defined as constants in a separate header file, shown in Appendix A.1 on page 25.

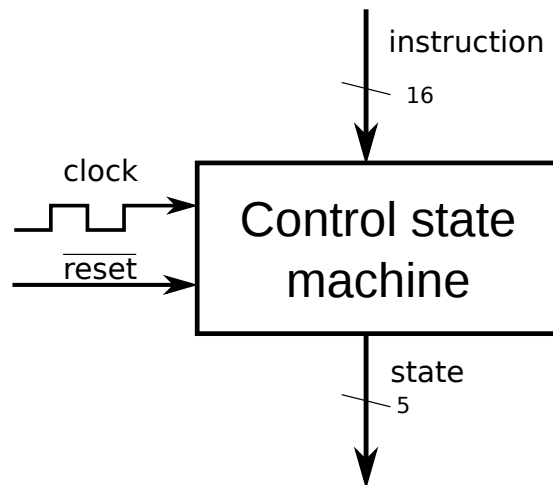


Figure 3.7: Block diagram of control module

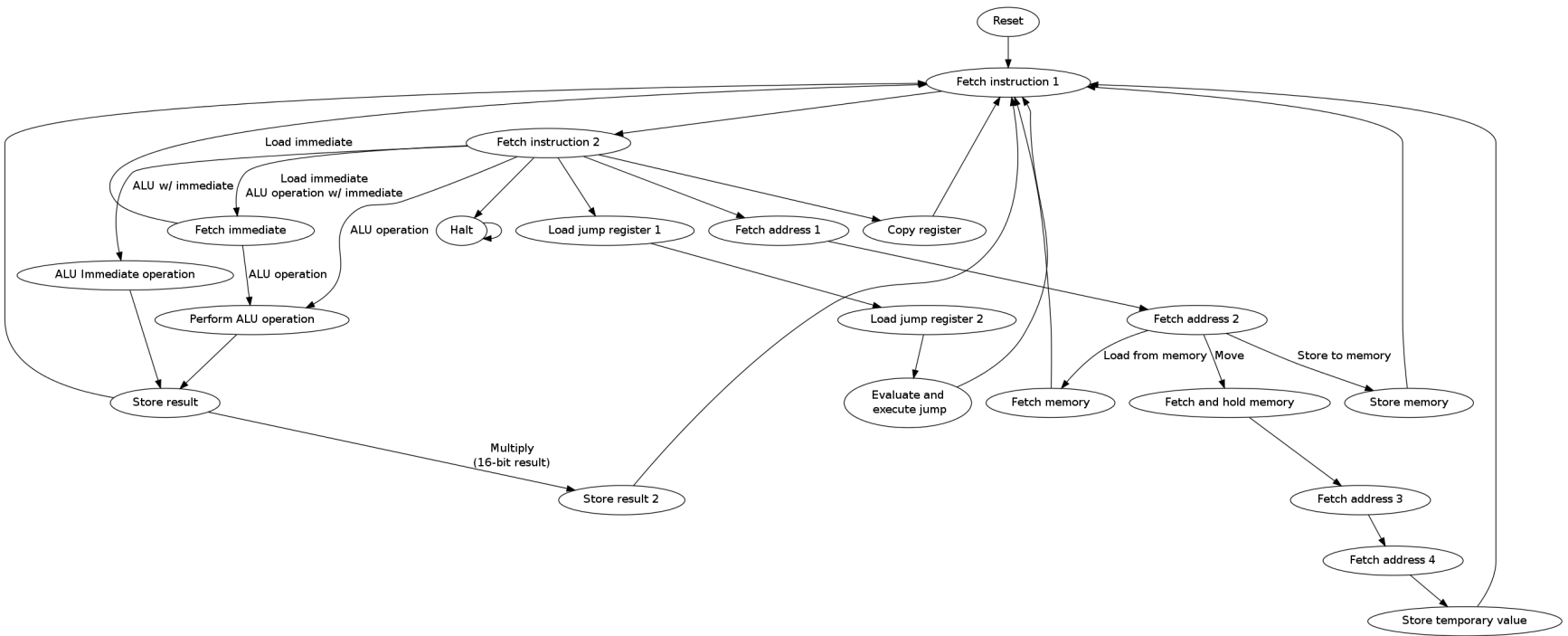


Figure 3.8: Control module state transition diagram. Each state takes exactly one clock cycle.

**Testing** The testbench for the control state machine uses an external file which contains instruction bytes interleaved with the sequence of states which the module must go through. The testbench reads a line of the file and checks that the current state matches the expected state. If the state is either `FETCH_1` or `FETCH_2`, a hexadecimal byte is provided in the file and the testbench loads this into the simulated instruction register. Lines ending in a `#` symbol are comments and are ignored. A sample segment of the test file is shown below.

```
1 NOP#
2 FETCH_1
3 00
4 FETCH_2
5 00
6 Add#
7 FETCH_1
8 08
9 FETCH_2
10 20
11 ALU_OPERATION
12 STORE_RESULT_1
13 Multiply#
14 FETCH_1
15 18
16 FETCH_2
17 20
18 ALU_OPERATION
19 STORE_RESULT_1
20 STORE_RESULT_2
```

The test file checks every path of the state machine and ensures that each one works correctly.

### 3.7.2 Control signal translation

**Design** The state-to-control-vector matrix is shown in Table 4. Pure combinational logic with `assign` statements was used to produce the proper outputs. The general-purpose register addresses are taken directly from the opcode, except where the destination address is modified to produce a 16-bit store for the multiply operation. The ALU operation is taken directly from opcode bits [14:11].

Table 4: Mapping of states to control signals

	gp_read	gp_write	pc_set	pc_increment	mem_read	mem_write	latch_alu	alu_store_high	alu_store_low	ir_load_high	ir_load_low	jr_load_high	jr_load_low	mar_load_high	mar_load_low
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
FETCH_1	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0
FETCH_2	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0
FETCH_IMMEDIATE	?	0	0	1	1	0	0	0	0	0	0	0	0	0	0
ALU_OPERATION	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
ALU_IMMEDIATE	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
STORE_RESULT	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0
STORE_RESULT_2	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
COPY_REGISTER_1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
COPY_REGISTER_2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
FETCH_ADDRESS_1	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0
FETCH_ADDRESS_2	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1
FETCH_MEMORY	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
STORE_MEMORY	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
TEMP_FETCH	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
FETCH_ADDRESS_3	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0
FETCH_ADDRESS_4	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1
TEMP_STORE	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0
LOAD_JUMP_1	0	0	0	1	1	0	0	0	0	0	0	1	0	0	0
LOAD_JUMP_2	0	0	0	1	1	0	0	0	0	0	0	0	1	0	0
EXECUTE_JUMP	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
HALT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The module also handles jump evaluation and execution. If the operation is a jump and the condition code is “always”, then the program counter `set` signal is asserted. If the operation is a jump, the condition code is `carry`, and the carry flag is set, then the signal is asserted. The same logic is used for the carry and zero jumps. No additional clock cycles are necessary for evaluating the jump.

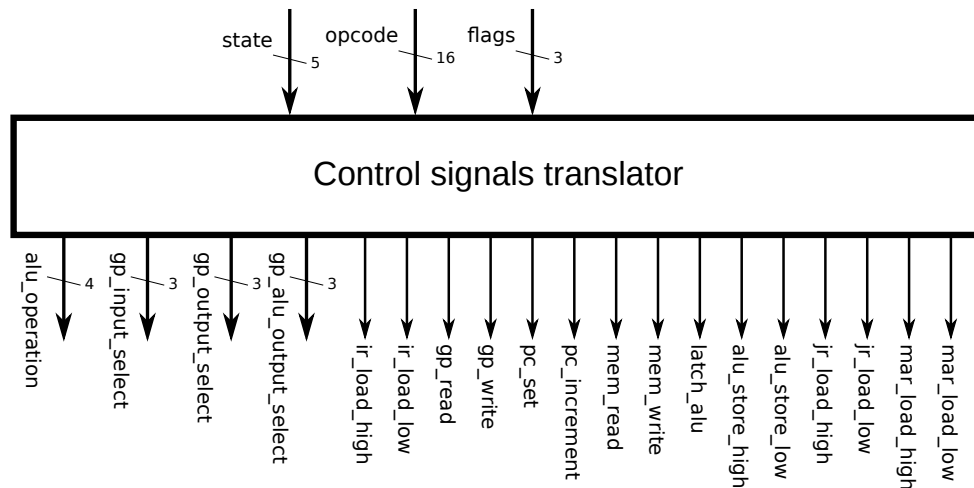


Figure 3.9: Control signal translation module block diagram

The module code for the control signal translation module is shown in [Appendix A.11 on page 37](#).

**Testing** The testbench simply iterates through all of the states, setting the input state and then checking all of the outputs against those expected from the table. The testbench code is shown in [B.8 on page 65](#).

### 3.8 Address multiplexer

The address multiplexer switches the output address between the program counter and memory address register based on the state. If the processor is performing a load or store using a memory location, the MAR address is used; otherwise, the program counter is used. The module code is shown in [Appendix A.8 on page 33](#). The address multiplexer was tested as part of the CPU.

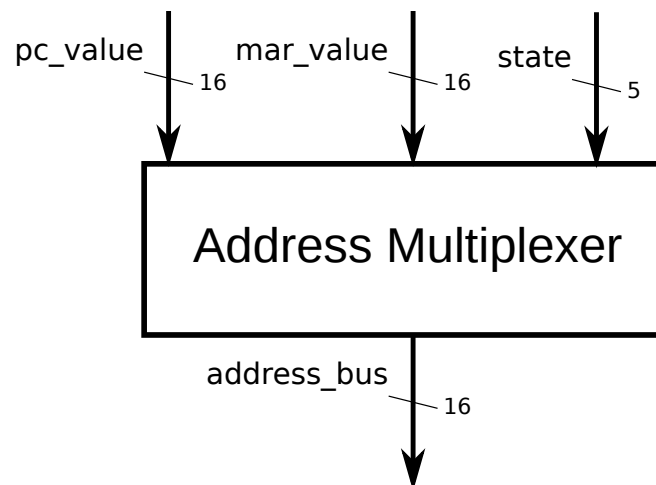


Figure 3.10: Address multiplexer block diagram

### 3.9 Memory IO

The memory IO module performs memory mapping to locate the ROM and RAM in address space, and translates the read and write signals into ROM and RAM chip enable signals.

### 3.9.1 Design

Because the program counter's reset is at 0x0000, it needs to find its first instruction at that memory location. This is done by memory-mapping the ROM to 0x0000 through 0x1FFF. The 8 by 8K RAM can occupy any portion of the address space; it was arbitrarily placed at 0x2000 through 0x3FFF. The module's operation is summarized in Table 5.

Table 5: Combinational operation of memory IO module

	Input address	
	0x0000 - 0x1FFFF	0x2000 - 0x13FFFF
readmemory = 1	rom_enable = 0, ram_enable = 1, write = 0	rom_enable = 1, ram_enable = 0, write = 0
writememory = 1	Invalid	rom_enable = 1, ram_enable = 0, write = 1
both low	rom_enable = 1, ram_enable = 1, write = 0	rom_enable = 1, ram_enable = 1, write = 0

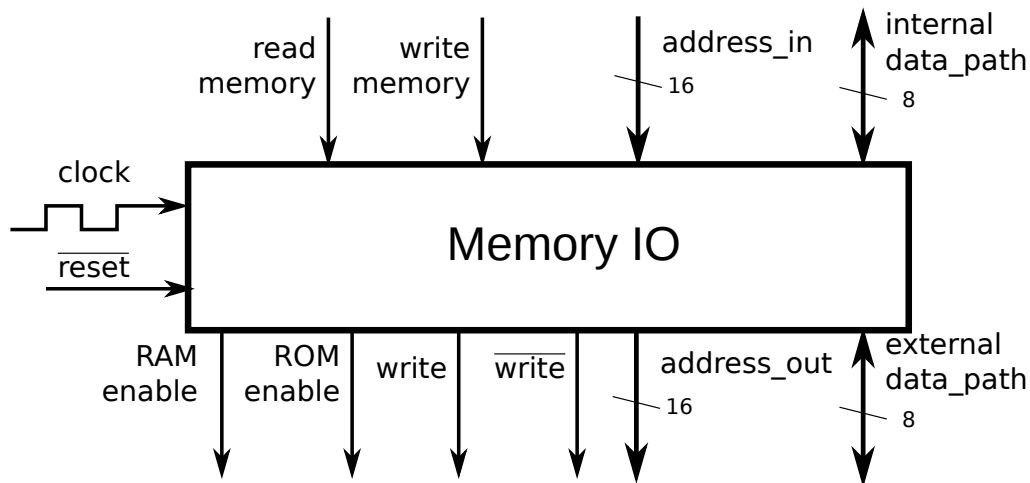


Figure 3.11: Block diagram of memory IO module

### 3.9.2 Testing

The memory IO unit was tested as part of the CPU.

## 3.10 Complete chip

### 3.10.1 Design

The CPU module includes all of the other modules. It connects the control state machine to the datapath via the state translation module, connects the datapath to the outside using the address multiplexer and memory IO module. A block diagram of the CPU is shown in Figure 3.12. For clarity, the datapath control signals are collapsed into a single representative bus.

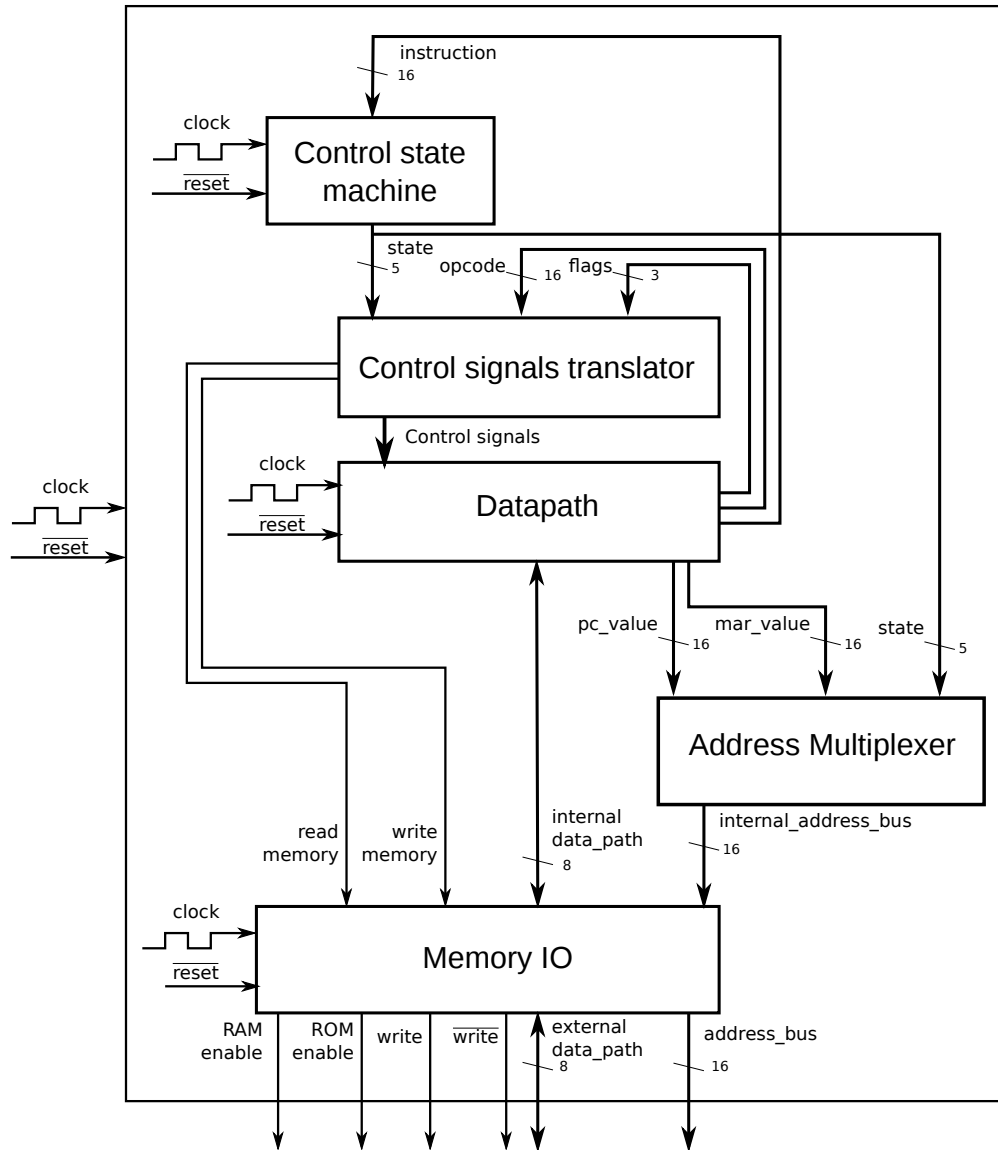


Figure 3.12: CPU block diagram

### 3.10.2 Testing

Rather than run a specific set of test vectors on the CPU, the testbench simulates RAM and ROM and effectively lets the chip run on its own. The initial ROM configuration is read in from a file produced by the assembler or by hand. The file contains a series of hexadecimal bytes, one per line. Lines beginning with the # symbol are comments and are ignored by the testbench.

After performing a reset, the testbench simply reads the address bus and memory enable lines and returns the appropriate value.

## 4 Hardware implementation

### 4.1 Motivation and design criteria

It was clear at the beginning of the class that many people would be competing for time on the implementation board, and its feature set and design are completely fixed. For these reasons, I decided early in the semester

to build my own board.

Two factors caused me to search for parts besides the LSI5512 provided in the class. There were relatively few chips and sockets available, and I assumed that other students would need them. Given that ours was the largest group of students to take this class at OC and that several graduate students would be building their own boards, a part shortage seemed imminent. More importantly, connecting wires to the 256-pin FPGBA socket is difficult and messy at best, and a debugging nightmare at worst. I desired some sort of breakout board which could interface easily with a breadboard.

## 4.2 Part selection

I researched several part families from Lattice, Xilinx, and Altera, comparing logic capacity, prices, IDE support, and other factors.

- The Lattice 4000V series (specifically the LC4512V) is the nominal successor to the discontinued LSI5000 series. It is compatible with the Lattice download cables in the Digital lab and is similar to the LSI5000 series. However, chips are surprisingly expensive (\$60), and is only supported by ispLEVER Classic, which compared to other tools is a very poor IDE.
- The Xilinx CoolRunner XC2C512 is moderately priced at \$40, and comes with a very powerful and polished IDE (Xilinx ISE WebPack), which Xilinx distributes for free. However, we do not have any download cables or programmers for the chip.
- In comparison to the CoolRunner CPLDs, Xilinx FPGAs are much cheaper and have an order of magnitude more gates. However, an FPGA is volatile, so it requires other hardware to be useful. Digilent makes several inexpensive (\$100-\$200) boards which contain a Xilinx Spartan FPGA and hardware to run them, including a USB-JTAG interface. This would be a simple and practical solution, but it is rather expensive and doesn't provide a simple interface to a breadboard.
- The Altera MAX II series is inexpensive (\$20), and has a free and powerful IDE. However, it requires a \$150 download cable.
- The Lattice MachXO series costs less than \$20 and works with programming cables in the Digital lab. It is fully supported by Lattice's newest IDE, known as Lattice Diamond. Although it does not seem as powerful as Xilinx ISE, Diamond is far superior to ispLEVER Classic.

My final selection was to use a MachXO series chip with a custom PCB.

The MachXO series uses a different architecture than traditional CPLDs which uses lookup tables rather than macrocells. The LCMXO-2280 has 2,280 lookup tables, which Lattice equates to 1140 macrocells. Early synthesis reports of my datapath showed approximately 200 macrocells used, and previous designs fit in the 512 macrocell LSI5512, so I was confident that my design would fit easily within the logic constraints of the LCMXO-2280.

## 4.3 PCB Design

I designed a 2-layer breakout PCB for the LCMXO-2280 using Eagle and had it fabricated by Advanced Circuits ([www.4pcb.com](http://www.4pcb.com)). The only other parts on the board were 16 eight-terminal female headers (Newark part #08N6773). One header is set up so that the ispDownload cable can be directly connected to the board (<http://www.latticesemi.com/lit/docs/devtools/dlcable.pdf>); a second header contains the  $V_{CC}$ ,  $V_{IO}$ , and ground connections for the device. The remaining headers are connected directly to the PLD's IO pins. The Eagle schematic and PCB layout are shown in Appendix D on page 110.

## 4.4 Breadboard layout

The breadboard was laid out as shown in Figure 4.1. A pair of 74HC245 buffers was used in series on the data bus to ensure that the 5 V external signals did not damage the chip. The data bus connects to a buffer running at 3.3 V, which connects to a buffer at 5 V, which connects to the RAM and ROM. Because each

buffer has a delay of approximately 100ns, the maximum clock speed of the processor is reduced to about 2 MHz. 74HC244 unidirectional buffers were used on the address bus.

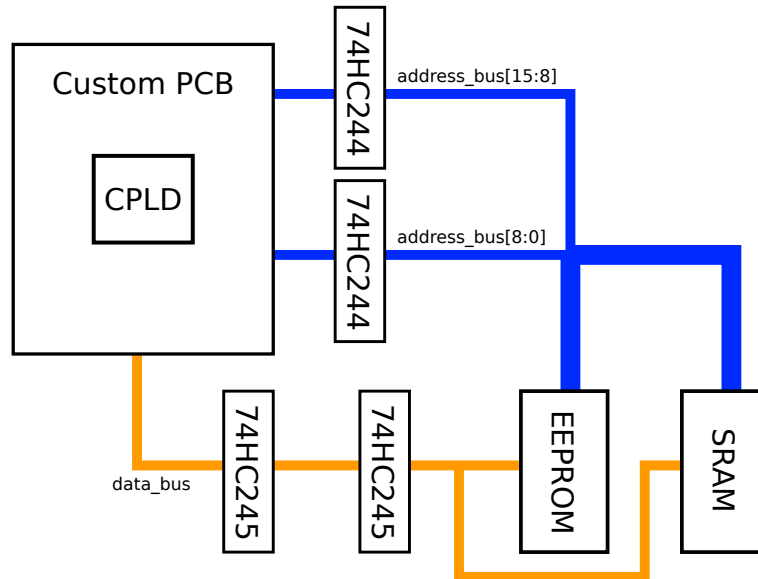


Figure 4.1: Block diagram of breadboard layout

Wires were color-coded using red for +3.3 V and +5 V, black for ground, blue for the address bus, orange for the data bus, green for other signals, and yellow for test points to the logic analyzer.

A photograph of the board is shown in Figure 4.2.

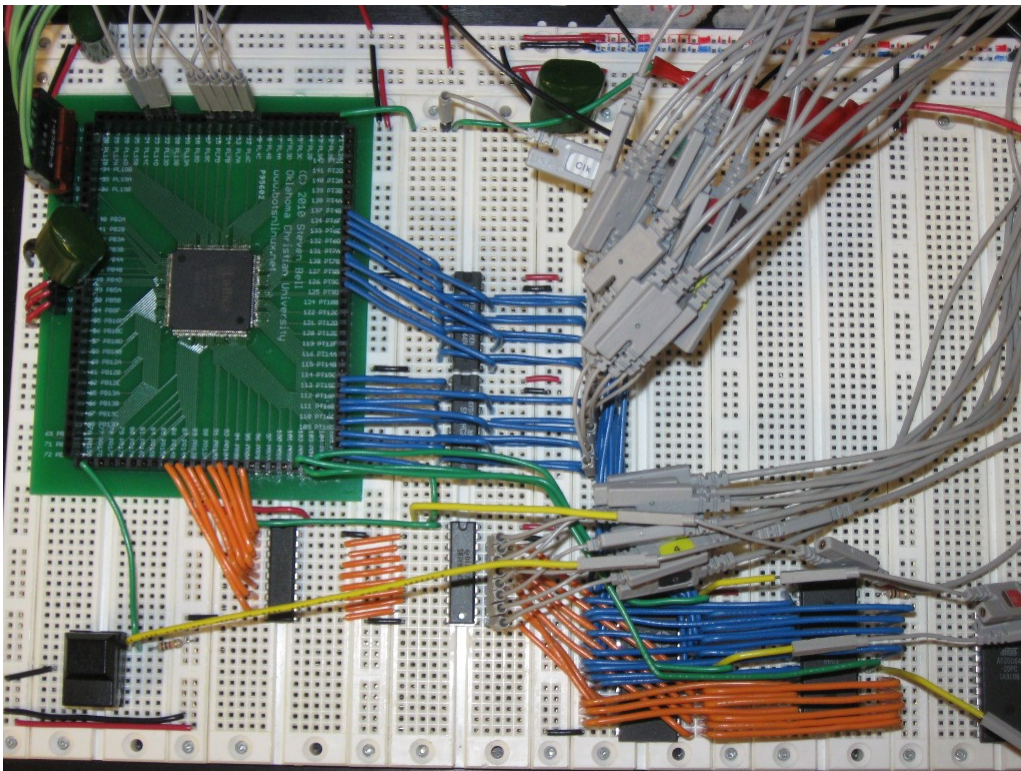


Figure 4.2: Photograph of the finished board

#### 4.4.1 Pin mapping

The MachXO pins are mapped as shown in the table below. The clock input is routed specifically as a clock pin by Lattice Diamond.

Port	Type	Pin	Port	Direction	Pin
clock	clock	1	address_bus_15	output	134
reset	input	76	address_bus_14	output	133
ram_enable	output	104	address_bus_13	output	132
rom_enable	output	105	address_bus_12	output	131
write	output	103	address_bus_11	output	130
write	output	102	address_bus_10	output	127
external_data_bus_7	bidirectional	95	address_bus_9	output	126
external_data_bus_6	bidirectional	94	address_bus_8	output	125
external_data_bus_5	bidirectional	92	address_bus_7	output	113
external_data_bus_4	bidirectional	91	address_bus_6	output	112
external_data_bus_3	bidirectional	90	address_bus_5	output	111
external_data_bus_2	bidirectional	89	address_bus_4	output	110
external_data_bus_1	bidirectional	87	address_bus_3	output	109
external_data_bus_0	bidirectional	86	address_bus_2	output	108
			address_bus_1	output	107
			address_bus_0	output	106

## 5 Assembler

The assembler is a command-line program written in C++ using the Qt framework. It takes an input file with assembly code based very loosely on Motorola 68HC11 assembly, and produces a Motorola SREC (.S19) file along with a hex dump which can be fed into the CPU simulation testbench. The assembler can handle labels as memory addresses, which is particularly useful for jump operations.

The assembler operates as follows:

1. The configuration is set up based on command-line parameters and the input file is opened.
2. One line of the file is read, broken up into individual tokens, and parsed.
3. If a label is encountered, the parser stores its memory address in a hash table for later reference.
4. If a label is referenced, the parser inserts the appropriate address.
5. If a label which has not yet been encountered is referenced, the parser adds the label to a table of unknown labels.
6. When the parser reaches the end of the file, it goes back through the unknown label list and fills in the remaining label references. If a label is still not defined, the assembler prints an error message.
7. The parser writes out the Motorola SREC file and, if requested, the simulation memory dump.

The complete assembler source code is given in Appendix E on page 112. Examples of the assembly source code which is parsed are given in Appendix F on page 130.

## 6 Conclusion

### 6.1 Current status

The required test program detailed in F on page 130 works correctly in simulation and in hardware. Captured waveforms from the logic analyzer are shown in Appendix G on page 135.

There is a bug in the ALU on certain multiplications: When a positive value is multiplied by a negative value to give a (e.g, -10 and 6, equivalent to 0xF5 and 0x06), the result is effectively treated as an 8-bit value but is put into the 16-bit register. Thus, rather than giving a 16-bit signed value, it returns an 8-bit signed value.

Arithmetic shifts are not working properly in the ALU, due to the way Verilog handles the arithmetic shift operator. The solution is to write a bit-level assignment to perform the arithmetic shift.

## 6.2 Future improvements

Several things could be improved on the current design. As my understanding of the Verilog language grew over the course of the semester, my coding style and practices evolved slightly. Were I to start again from scratch, I would:

- Be more careful in my differentiation of blocking and non-blocking assignments. Although my code works, it could probably be improved by using better logic design practices.
- Watch more closely for inferred latches and make sure to clock every module with storage. During the implementation phase, I had to fix several bugs related to inferred latches and unclocked operation.
- Run all of my testbenches based on the positive clock edges. Ultimately, my entire chip operates on the rising edge of the clock; nothing happens on the falling edge. However, I initially tested all of my modules assuming that inputs changed on the negative edges and that they did their work on the rising edge. This obscured my understanding of what was actually happening when I assembled the complete chip.
- Use port names to connect modules rather than argument order. This small syntactical feature was particularly useful at the CPU level, but would have been advantageous even for small modules.
- Define all constants in a single header file (with a .h extension) before writing my ALU and control state machine. I went through several iterations of merging and renaming files as I added new constants and switched build environments.

## A Verilog module code

### A.1 Global constants

```

1  /* constants.vh Definition file for chip-wide opcodes and other parameters
2  * Author: Steven Bell <steven.bell@student.oc.edu>
3  * Date: 4 November 2010
4  * $LastChangedDate: 2010-12-13 18:04:13 -0600 (Mon, 13 Dec 2010) $
5  */
6
7  `ifndef _CONSTANTS_V_
8  `define _CONSTANTS_V_
9
10 `define ZEROFLAG 2
11 `define CARRYFLAG 1
12 `define NEGFLAG 0
13
14 // ALU Opcodes
15 // These are the four lower bits from the opcode (instruction bits 14-11)
16 `define ALU_PASSTHROUGH 4'b1100
17 `define ALU_ADD 4'b0001
18 `define ALU_SUBTRACT 4'b0010
19 `define ALU_MULTIPLY 4'b0011
20 `define ALU_AND 4'b0100
21 `define ALU_OR 4'b0101
22 `define ALU_LOGICAL_SHIFT_RIGHT 4'b0110
23 `define ALU_LOGICAL_SHIFT_LEFT 4'b0111
24 `define ALU_ARITH_SHIFT_RIGHT 4'b1001
25 `define ALU_ARITH_SHIFT_LEFT 4'b1010
26 `define ALU_TWOS_COMPLEMENT 4'b1011
27 `define ALU_COMPLEMENT 4'b1000
28 // Room for 2 more, 4'b1101 and 4'b1110
29 // 4'b0000 is taken for NOP 4'b1111 for JUMP - these will give a 0 result
30
31 // Control module opcodes
32 // These are the complete 5-bit opcodes used in the control module
33 `define NOP 5'b00000
34 `define ADD 5'b00001
35 `define SUBTRACT 5'b00010
36 `define MULTIPLY 5'b00011
37 `define AND 5'b00100
38 `define OR 5'b00101
39 `define LOGICAL_SHIFT_RIGHT 5'b00110
40 `define LOGICAL_SHIFT_LEFT 5'b00111
41 `define COMPLEMENT 5'b01000
42 `define ARITH_SHIFT_RIGHT 5'b01001
43 `define ARITH_SHIFT_LEFT 5'b01010
44 `define TWOS_COMPLEMENT 5'b01011
45 `define PASSTHROUGH 5'b01100
46 `define LOAD 5'b10000
47 `define STORE 5'b10001
48 `define MOVE 5'b10010
49 `define JUMP 5'b01111
50 `define HALT 5'b11111

```

```

51
52 `define SOURCE_REGISTER 2'b00
53 `define SOURCE_IMMEDIATE 2'b10
54 `define SOURCE_MEMORY 2'b01
55
56 // Control module states
57 // Used in the control module and the control_signals module
58 `define S_RESET 5'd0
59 `define S_FETCH_1 5'd1
60 `define S_FETCH_2 5'd2
61 `define S_ALU_OPERATION 5'd3
62 `define S_STORE_RESULT_1 5'd4
63 `define S_STORE_RESULT_2 5'd5
64 `define S_FETCH_IMMEDIATE 5'd6
65 `define S_COPY_REGISTER 5'd7
66 `define S_FETCH_ADDRESS_1 5'd8
67 `define S_FETCH_ADDRESS_2 5'd9
68 `define S_FETCH_MEMORY 5'd10
69 `define S_STORE_MEMORY 5'd11
70 `define S_TEMP_FETCH 5'd12
71 `define S_FETCH_ADDRESS_3 5'd13
72 `define S_FETCH_ADDRESS_4 5'd14
73 `define S_TEMP_STORE 5'd15
74 `define S_LOAD_JUMP_1 5'd16
75 `define S_LOAD_JUMP_2 5'd17
76 `define S_EXECUTE_JUMP 5'd18
77 `define S_HALT 5'd19
78 `define S_ALU_IMMEDIATE 5'd20
79
80 `endif

```

## A.2 Generic 16-bit register

```

1 /* register_16bit.v
2  * Implements a generic 16-bit register which is loaded in two
3  * sequential 8-byte actions.
4  * Author: Steven Bell <steven.bell@student.oc.edu>
5  * Date: 16 September 2010
6  * $LastChangedDate: 2010-09-23 08:44:58 -0500 (Thu, 23 Sep 2010) $
7  */
8
9 module register_16bit(clock, reset, setHigh, setLow, halfValueIn, valueOut);
10     input clock;
11     input reset; // Synchronous reset; active low
12     input setHigh; // When this signal is high, the top half of the value is
13         loaded from the input line (data bus)
14     input setLow;
15     input [7:0] halfValueIn;
16     output reg [15:0] valueOut; // Output value containing both bytes
17
18     always @(posedge clock) begin
19         if(~reset) begin // If the reset line is low, then zero the register
20             valueOut = 0;
21         end

```

```

21     else if(setHigh) begin
22         valueOut[15:8] = halfValueIn; // Load the top half
23         // Leave the bottom half the same
24     end
25     else if(setLow) begin
26         // Leave the top half the same
27         valueOut[7:0] = halfValueIn; // Load the bottom half
28     end
29 end // END always
30
31 endmodule

```

### A.3 Program counter

```

1  /* program_counter.v
2  * Implements the 16-bit loadable program counter.
3  * Author: Steven Bell <steven.bell@student.oc.edu>
4  * Date: 11 September 2010
5  * $LastChangedDate: 2010-11-23 09:34:58 -0600 (Tue, 23 Nov 2010) $
6  */
7
8  /* Program counter which is used to index memory to load instructions.
9  * It can be set to new values to implement a jump. We have to do the set
10 * in one shot, because otherwise we jump partway and can't get the next byte.
11 */
12 module program_counter(clock, reset, increment, set, new_count, count);
13     input clock;
14     input reset; // Synchronous reset; active low
15     input increment; // Only increment the counter when this signal is high
16     input set; // When this signal is high, the counter loads new_count into the
17                 counter
18     input [15:0] new_count; // New value to set the counter to
19     output reg [15:0] count; // Output address of the program counter
20
21     // Clocked operation
22     always @(posedge clock) begin
23         if(~reset) begin // If the reset line is low, then zero the counter
24             count <= 0;
25         end
26         else if(set) begin // If set is high, then load a new value into the
27             counter
28             count <= new_count;
29         end
30         else if(increment) begin // Otherwise, if increment is high, add one to
31             the counter
32             count <= count + 1;
33         end
34     end // END always
35 endmodule

```

### A.4 ALU

```

1  /* alu.v
2  * Arithmetic logic unit.

```

```

3  * Author: Steven Bell <steven.bell@student.oc.edu>
4  * Date: 30 September 2010
5  * $LastChangedDate: 2010-12-13 18:04:13 -0600 (Mon, 13 Dec 2010) $
6  */
7
8  `include "constants.v"
9
10 module alu(clock, reset, primaryOperand, secondaryOperand, operation, result,
11          flags);
12     input clock; // TODO: remove the clock from the port list, since we're not
13         using it anymore
14     input reset; // Asynchronous reset; active low
15     input [7:0] primaryOperand; // Used for all operations except passthrough
16     input [7:0] secondaryOperand; // Used for two-operand operations
17     input [3:0] operation; // Up to 16 operations
18     output [15:0] result;
19     reg [15:0] result;
20     output [2:0] flags;
21     reg [2:0] flags; // Zero, carry, negative
22
23     always @(*) begin
24         result[15:0] = 16'd0; // Reset all of the bits so we don't infer any
25             latches
26
27             // But does the assignment (= rather than <=) cause
28             extra logic?
29
30         // Do the requested operation
31         case(operation)
32             `ALU_PASSTHROUGH: begin
33                 result[7:0] = secondaryOperand; // Send the data bus input directly to
34                     the result
35                 // It's pointless to pass the register operand through. If we need to
36                     move it from
37                 // one register to another, we can just put it on the data bus.
38                 flags[`CARRYFLAG] = 1'b0; // There is never a carry on a passthrough
39                 flags[`NEGFLAG] = result[7];
40             end
41             `ALU_ADD: begin
42                 result[8:0] = primaryOperand + secondaryOperand;
43                 flags[`CARRYFLAG] = result[8]; // See if a bit was carried
44                 flags[`NEGFLAG] = result[7];
45             end
46             `ALU_SUBTRACT: begin
47                 result[8:0] = primaryOperand - secondaryOperand;
48                 flags[`CARRYFLAG] = result[8]; // If the bit is a 1, then we had to
49                     borrow
50                 flags[`NEGFLAG] = result[7];
51             end
52             `ALU_MULTIPLY: begin
53                 result[15:0] = primaryOperand * secondaryOperand;
54                 flags[`CARRYFLAG] = 1'b0;
55                 flags[`NEGFLAG] = result[15];
56             end
57             `ALU_AND: begin
58                 result[7:0] = primaryOperand & secondaryOperand;

```

```

50     flags[`CARRYFLAG] = 1'b0;
51     flags[`NEGFLAG] = result[7];
52     end
53     `ALU_OR: begin
54         result[7:0] = primaryOperand | secondaryOperand;
55         flags[`CARRYFLAG] = 1'b0;
56         flags[`NEGFLAG] = result[7];
57     end
58     `ALU_LOGICAL_SHIFT_RIGHT: begin
59         result[7:0] = primaryOperand >> 1;
60         flags[`CARRYFLAG] = primaryOperand[0]; // Set the carry to be the bit
           that got shifted out
61         flags[`NEGFLAG] = result[7]; // This will always be 0; perhaps we
           should set it explicitly?
62     end
63     `ALU_LOGICAL_SHIFT_LEFT: begin
64         result[7:0] = primaryOperand << 1;
65         flags[`CARRYFLAG] = primaryOperand[7]; // Set the carry to be the bit
           that got shifted out
66         flags[`NEGFLAG] = result[7];
67     end
68     `ALU_ARITH_SHIFT_RIGHT: begin
69         result[7:0] = primaryOperand >>> 1;
70         flags[`CARRYFLAG] = primaryOperand[0]; // Set the carry to be the bit
           that got shifted out
71         flags[`NEGFLAG] = result[7];
72     end
73     `ALU_ARITH_SHIFT_LEFT: begin
74         result[7:0] = primaryOperand <<< 1;
75         flags[`CARRYFLAG] = primaryOperand[7]; // Set the carry to be the bit
           that got shifted out
76         flags[`NEGFLAG] = result[7];
77     end
78     `ALU_TWOS_COMPLEMENT: begin
79         result = ~primaryOperand; // Complement
80         result = result + 8'd1; // and add one
81         flags[`CARRYFLAG] = 1'b0;
82         flags[`NEGFLAG] = result[7];
83     end
84     `ALU_COMPLEMENT: begin
85         result = ~primaryOperand;
86         flags[`CARRYFLAG] = 1'b0;
87         flags[`NEGFLAG] = result[7];
88     end
89     default: begin
90         result[15:0] = 15'd0;
91         flags[`CARRYFLAG] = 1'b0;
92         flags[`NEGFLAG] = 1'b0;
93     end
94 endcase
95
96 if(operation == `ALU_MULTIPLY) begin
97     flags[`ZEROFLAG] = (result[15:0] == 16'h0000) ? 1'b1 : 1'b0;
98 end

```

```

99     else begin
100         flags[`ZEROFLAG] = (result[7:0] == 8'h00) ? 1'b1 : 1'b0;
101     end // if(operation == `MULTIPLY)
102 end //always @(posedge clock)
103
104 endmodule

```

## A.5 ALU latch

```

1  /* alu_latch.v
2  * Grabs the result from the ALU and puts it on the data bus
3  * Author: Steven Bell <steven.bell@student.oc.edu>
4  * Date: 11 October 2010
5  * $LastChangedDate: 2010-12-13 18:04:13 -0600 (Mon, 13 Dec 2010) $
6  */
7
8  module alu_latch(
9      input clock,
10     input reset,
11     input [15:0] alu_result, // Result from the ALU
12     input [2:0] flags, // ALU flags which must also be latched
13     input grab, // Active-high signal telling us whether or not to latch the
14         value
15     input store_high, // Put the top 8 bytes on the data bus (only used for
16         multiply)
17     input store_low, // Put the low 8 bytes on the data bus
18     output reg [7:0] out,
19     output reg [2:0] flags_out
20 );
21
22     reg [15:0] value; // Stores the full-length output value
23
24     always @(posedge clock) begin
25         if(reset == 1'b0) begin
26             value <= 16'b0;
27         end
28         if(grab == 1'b1) begin // Latch the ALU value when the grab signal is high
29             value <= alu_result;
30             flags_out <= flags;
31         end
32     end // always
33
34     /* This part will synthesize into combinational logic which puts
35     * the appropriate set of signals onto the data bus. */
36     always @(*) begin
37         if(store_low == 1'b1) begin
38             out <= value[7:0];
39         end
40         else if(store_high == 1'b1) begin
41             out <= value[15:8];
42         end
43         else begin
44             out <= 8'hzz;
45         end

```

```

44   end
45
46 endmodule

```

## A.6 General-purpose register block

```

1  /* gp_registers.v
2  * 8x8 bit general purpose register set.
3  * Author: Steven Bell <steven.bell@student.oc.edu>
4  * Date: 23 September 2010
5  * $LastChangedDate: 2010-12-14 09:01:55 -0600 (Tue, 14 Dec 2010) $
6  */
7
8  module gp_registers
9  (
10     input  clock,
11     input  reset, // Active-low synchronous reset
12     input  read_data, // Flag telling us whether or not to load a register value
13         from the data bus (active high)
14     input  write_data, // Flag telling us whether or not to write a register to
15         the data bus (active high)
16     // We'll just leave the ALU output always enabled, since it can't mess
17         things up and we
18     // don't care about the slight increase in power consumption due to flipping
19         unnecessary gates.
20
21     input[2:0] input_select, // Register to put the input value into
22     input[2:0] output_select, // Index of the register we want to put on the
23         data bus output
24     input[2:0] alu_output_select, // Index of the register we want to put on the
25         ALU output
26
27     inout[7:0] data_bus, // Contains the input value to store, or the output we
28         write
29     output[7:0] alu_output_value // Output bus to the ALU
30 );
31
32 reg[7:0] register_data[7:0]; // Data array, 8 bits x 8 registers
33 wire[7:0] output_value; // Temporary latch, because the data_bus is a wire
34         and not a reg
35
36 integer i; // Used for iterating through the registers when resetting them
37
38 always@(posedge clock) begin
39     if(reset == 1'b0) begin
40         // Remember that this code produces hardware, and all of the registers
41         // will be reset simultaneously
42         for(i = 0; i < 8; i = i+1) begin
43             register_data[i] <= 8'd0;
44         end
45     end
46
47     if(read_data == 1'b1) begin
48         register_data[input_select] <= data_bus;

```

```

40     end
41 end // always
42
43 // Combinational logic to interface with the data bus
44
45 assign output_value = register_data[output_select];
46 assign data_bus = write_data ? output_value : 8'hzz;
47 assign alu_output_value = register_data[alu_output_select];
48
49 endmodule

```

## A.7 Datapath

```

1 /* datapath.v
2  * Datapath which includes the general purpose registers, special purpose
3  * registers, ALU, and the connections between them.
4  * Author: Steven Bell <steven.bell@student.oc.edu>
5  * Date: 6 October 2010
6  * $LastChangedDate: 2010-12-13 18:04:13 -0600 (Mon, 13 Dec 2010) $
7  */
8
9 module datapath(
10     input  clock,
11     input  reset,
12     input  pc_increment,
13     input  pc_set,
14     input  gp_read,
15     input  gp_write,
16     input [2:0] gp_input_select,
17     input [2:0] gp_output_select,
18     input [2:0] gp_alu_output_select,
19     input [3:0] alu_operation,
20     input  latch_alu,
21     input  alu_store_high,
22     input  alu_store_low,
23     input  mar_set_high,
24     input  mar_set_low,
25     input  ir_set_high,
26     input  ir_set_low,
27     input  jr_set_high,
28     input  jr_set_low,
29
30     output wire [15:0] pc_count, // Program counter output
31     output wire [15:0] mar_value, // Memory address register output
32     output wire [15:0] ir_value, // Instruction register output
33     output wire [2:0] flags, // ALU flags
34
35     inout [7:0] data_bus
36 );
37
38 // Shared connections
39 wire [7:0] register_operand; // Bus from the general-purpose registers to the
    ALU

```

```

40  wire[15:0] pc_jump_count; // Bus from the jump register to the program
    counter (value to jump to)
41  wire[15:0] alu_result;
42  wire[2:0] flags_temp;
43
44  // Program counter (only reads from jump register, not data bus)
45  program_counter m_program_counter(clock, reset, pc_increment, pc_set,
    pc_jump_count, pc_count);
46
47  // Jump register which holds the value for the program counter to jump to
48  register_16bit m_jump_register(clock, reset, jr_set_high, jr_set_low,
    data_bus, pc_jump_count);
49
50  // General-purpose registers
51  gp_registers m_gp_registers(clock, reset, gp_read, gp_write, gp_input_select
    , gp_output_select, gp_alu_output_select, data_bus, register_operand);
52
53  // ALU and ALU latch
54  alu m_alu(clock, reset, register_operand, data_bus, alu_operation,
    alu_result, flags_temp);
55  alu_latch m_alu_latch(clock, reset, alu_result, flags_temp, latch_alu,
    alu_store_high, alu_store_low, data_bus, flags);
56
57  // Memory address register
58  register_16bit m_address_register(clock, reset, mar_set_high, mar_set_low,
    data_bus, mar_value);
59
60  // Instruction register
61  register_16bit m_instruction_register(clock, reset, ir_set_high, ir_set_low,
    data_bus, ir_value);
62
63
64
65  endmodule

```

## A.8 Address Multiplexer

```

1  /* address_mux.v Multiplexer that picks the output address bus value from
    either the MAR or
2     *           the program counter based the control module state.
3  * Author: Steven Bell <steven.bell@student.oc.edu>
4  * Date: 6 December 2010
5  * $LastChangedDate: 2010-12-16 17:48:32 -0600 (Thu, 16 Dec 2010) $
6  */
7
8  `include "constants.v"
9
10 module address_mux(
11     input [15:0] pc_value,
12     input [15:0] mar_value,
13     input [4:0] state,
14     output [15:0] address_bus
15 );
16

```

```

17   assign address_bus = (state === `S_FETCH_MEMORY || state === `S_STORE_MEMORY
18       ||
19       state === `S_TEMP_FETCH || state === `S_TEMP_STORE) ?
        mar_value : pc_value;
19 endmodule

```

## A.9 Memory IO

```

1  /* memio.v Memory IO module which performs memory mapping and holds the
2  * digital IO banks.
3  *
4  * Author: Steven Bell <steven.bell@student.oc.edu>
5  * Date: 5 December 2010
6  * $LastChangedDate$
7  */
8
9  module memio(
10     input read_memory,
11     input write_memory,
12     inout[7:0] internal_data_path,
13     inout[7:0] external_data_path,
14
15     input[15:0] address_in, // Full 16-bit address input
16     output[15:0] address_out, // Output address; not all 16 bits may be used
17     // TODO: digital inputs/outputs
18     output ram_enable, // Active low signal which turns the RAM on
19     output rom_enable, // Active low signal which turns the ROM on
20     output write, // Active high signal which tells the bidirectional buffers we
        're writing
21     output write_bar // Opposite of write; active low signal which tells the RAM
        we're writing
22 );
23 // TODO: use this to make sure we don't crash and burn if both read and
        write are asserted
24 wire enabled;
25 assign enabled = read_memory ^ write_memory;
26
27 assign internal_data_path = read_memory ? external_data_path : 8'hzz;
28 assign external_data_path = write_memory ? internal_data_path : 8'hzz;
29
30
31 // Map the ROM to 0000 - 1FFF
32 // Output is active low
33 assign rom_enable = !(address_in[15:13] === 3'b000);
34
35 // Map the RAM to 2000 - 3FFF
36 // Output is active low
37 assign ram_enable = !(address_in[15:13] === 3'b001);
38
39 assign address_out = address_in; // Passthrough for now
40 assign write = write_memory;
41 assign write_bar = !write_memory;
42
43 endmodule

```

## A.10 Control module state machine

```

1  /* control.v
2  * Control module which fetches instructions and drives all of the other
3  * modules.
4  * Author: Steven Bell <steven.bell@student.oc.edu>
5  * Date: 28 October 2010
6  * $LastChangedDate: 2010-12-13 18:04:13 -0600 (Mon, 13 Dec 2010) $
7  */
8
9  `include "constants.v"
10
11 module control
12 (
13     input  clock,
14     input  reset,
15     input [15:0] instruction, // Value from the instruction register
16     output reg[4:0] state // Current control state
17 );
18
19 always @(posedge clock) begin
20
21     // Moving into reset is handled by an if statement at the end
22     case(state)
23         `S_RESET:
24             // Everything is zero
25             if(reset != 1'b0) begin // If reset is no longer asserted, it's time
26                 // to start!
27                 state <= `S_FETCH_1;
28             end
29         `S_FETCH_1:
30             // Always to go to fetch 2, since every opcode is two bytes
31             state <= `S_FETCH_2;
32         `S_FETCH_2:
33             // Figure out what to do next based on the first half of the opcode
34             // We won't have the second half until the end of the clock
35             if(instruction[15:11] == `NOP) begin
36                 state <= `S_FETCH_1;
37             end
38             else if(instruction[15:11] == `JUMP) begin
39                 state <= `S_LOAD_JUMP_1;
40             end
41             // If the opcode begins with a zero, then it's an ALU operation,
42             // except if it's
43             // 01101 or 01110, which are not used (JUMP and NOP are already
44             // handled)
45             else if(instruction[15] == 1'b0 && instruction[15:11] != 5'b01101 &&
46                 instruction != 5'b01110) begin
47                 if(instruction[10:9] == `SOURCE_REGISTER) begin
48                     state <= `S_ALU_OPERATION;
49                 end
50                 else begin // TODO: Crash gracefully if it's invalid
51                     state <= `S_ALU_IMMEDIATE;
52                 end
53             end
54         endcase
55     end
56 end

```

```

49     end
50     else if(instruction[15:11] == `LOAD) begin
51         if(instruction[10:9] == `SOURCE_REGISTER) begin
52             state <= `S_COPY_REGISTER;
53         end
54         else if(instruction[10:9] == `SOURCE_IMMEDIATE) begin
55             state <= `S_FETCH_IMMEDIATE;
56         end
57         else if(instruction[10:9] == `SOURCE_MEMORY) begin
58             state <= `S_FETCH_ADDRESS_1;
59         end
60     end
61     else if(instruction[15:11] == `STORE ||
62             instruction[15:11] == `MOVE) begin
63         state <= `S_FETCH_ADDRESS_1;
64     end
65     else if(instruction[15:11] == `S_HALT) begin
66         state <= `S_HALT;
67     end
68     else begin
69         state <= `S_HALT;
70     end
71
72     `S_ALU_OPERATION:
73         state <= `S_STORE_RESULT_1;
74
75     `S_ALU_IMMEDIATE:
76         state <= `S_STORE_RESULT_1;
77
78     `S_STORE_RESULT_1:
79         // If the operation was a multiply, we have to do 2 stores
80         if(instruction[15:11] == `MULTIPLY) begin
81             state <= `S_STORE_RESULT_2;
82         end
83         // Otherwise, we're ready for the next operation
84         else begin
85             state <= `S_FETCH_1;
86         end
87     `S_STORE_RESULT_2:
88         state <= `S_FETCH_1;
89     `S_FETCH_IMMEDIATE:
90         // Because this state is used both for storing immediates into the
91         // registers
92         // and for grabbing immediate values for the ALU, this state loads the
93         // immediate
94         // value into the register. It will be overwritten by the ALU result
95         // if there is one.
96         // This could be fixed by creating a separate state for loading
97         // immediates.
98         if(instruction[15:11] == `LOAD) begin
99             state <= `S_FETCH_1;
100        end
101        else begin
102            state <= `S_ALU_OPERATION;

```

```

99     end
100    `S_COPY_REGISTER:
101        state <= `S_FETCH_1;
102    `S_FETCH_ADDRESS_1:
103        state <= `S_FETCH_ADDRESS_2;
104    `S_FETCH_ADDRESS_2:
105        if(instruction[15:11] == `LOAD) begin
106            state <= `S_FETCH_MEMORY;
107        end
108        else if(instruction[15:11] == `STORE) begin
109            state <= `S_STORE_MEMORY;
110        end
111        else if(instruction[15:11] == `MOVE) begin
112            state <= `S_TEMP_FETCH;
113        end
114    `S_FETCH_MEMORY:
115        state <= `S_FETCH_1;
116    `S_STORE_MEMORY:
117        state <= `S_FETCH_1;
118    `S_TEMP_FETCH:
119        state <= `S_FETCH_ADDRESS_3;
120    `S_FETCH_ADDRESS_3:
121        state <= `S_FETCH_ADDRESS_4;
122    `S_FETCH_ADDRESS_4:
123        state <= `S_TEMP_STORE;
124    `S_TEMP_STORE:
125        state <= `S_FETCH_1;
126    `S_LOAD_JUMP_1:
127        state <= `S_LOAD_JUMP_2;
128    `S_LOAD_JUMP_2:
129        state <= `S_EXECUTE_JUMP;
130    `S_EXECUTE_JUMP:
131        state <= `S_FETCH_1;
132    `S_HALT:
133        state <= `S_HALT; // Just stay put!
134    endcase
135
136    if(reset == 0) begin
137        state <= `S_RESET;
138    end
139
140 end
141
142
143 endmodule

```

## A.11 Control signals translator

```

1  /* control_signals.v
2  * Module which translates the control module state into the set of control
   signals
3  * for the rest of the chip.
4  * Author: Steven Bell <steven.bell@student.oc.edu>
5  * Date: 24 November 2010

```

```

6  * $LastChangedDate$
7  */
8
9  `include "constants.v"
10
11 module control_signals(
12     input [4:0] state, // State from the control state machine
13     input [15:0] opcode, // Full 16-bit opcode from the instruction register
14     input [2:0] alu_flags, // Carry/Zero/Negative flags from ALU
15
16     output ir_load_high, // Load the high 8 bits of the instruction from the
        data bus
17     output ir_load_low, // Load the low 8 bits of the instruction
18     output gp_read, // Read a value from the data bus into a register
19     output gp_write, // Write a value from the register onto the data bus
20     output pc_set, // Set the program counter from the jump register
21     output pc_increment, // Increment the program counter
22     output mem_read, // Read a value from memory onto the data bus
23     output mem_write, // Write a value from the data bus out to memory (RAM)
24     output latch_alu,
25     output alu_store_high, // Write the high 8 bits of the ALU result to the
        data bus
26     output alu_store_low, // Write the low 8 bits of the ALU result to the data
        bus
27     output jr_load_high, // Load the high 8 bits of the jump destination into
        the jump register
28     output jr_load_low, // Load the low 8 bits of the jump destination
29     output mar_load_high, // Load the high 8 bits of the memory address into the
        MAR
30     output mar_load_low, // Load the low 8 bits of the memory address
31
32     output [3:0] alu_operation,
33     output [2:0] gp_input_select,
34     output [2:0] gp_output_select, // Register select for GP registers to data
        bus
35     output [2:0] gp_alu_output_select // Register select for GP registers
        directly to ALU
36 );
37
38 // Signals directly from the opcode
39 assign alu_operation = (opcode[15:11] === `MOVE) ? `ALU_PASSTHROUGH : opcode
    [14:11];
40 // The assignments below assume that the ALU operand directly from the GP
    registers
41 // is the "primary operand" used for unary operations.
42 assign gp_input_select[2:1] = opcode[4:3]; // GP registers from data bus
43
44 wire gp_address_force; // Bit used to force a particular address when using
    a multiply
45 assign gp_address_force = (state === `S_STORE_RESULT_2) ? 1'b1 : 1'b0;
46 assign gp_input_select[0] = (opcode[15:11] === `MULTIPLY) ? gp_address_force
    : opcode[2];
47 assign gp_output_select = opcode[7:5]; // GP registers to the data bus
48 assign gp_alu_output_select = opcode[4:2]; // GP registers to the ALU

```

```

49
50 // Signals from the state machine
51 assign ir_load_high = (state === `S_FETCH_1);
52 assign ir_load_low = (state === `S_FETCH_2);
53 // Read into the registers if we have a store, a copy, or are loading an
    immediate into a register
54 assign gp_read = (state === `S_STORE_RESULT_1 || state === `S_STORE_RESULT_2
    ||
55                 state === `S_COPY_REGISTER || state === `S_FETCH_MEMORY ||
56                 (state === `S_FETCH_IMMEDIATE && opcode[15:11] === `LOAD))
    ;
57 // Write from the registers if we have a register ALU operation or a store
58 assign gp_write = ((state === `S_ALU_OPERATION && opcode[10:9] === 2'b00) ||
59                  state === `S_STORE_MEMORY || state === `S_COPY_REGISTER);
60 // Jump only if the code from the ALU tells us to
61 assign pc_set = (state === `S_EXECUTE_JUMP &&
62                (opcode[1:0] === 2'b00 ||
63                 opcode[1:0] === 2'b01 && alu_flags[`CARRYFLAG] == 1'b1) ||
64                 opcode[1:0] === 2'b10 && alu_flags[`ZEROFLAG] == 1'b1) ||
65                 opcode[1:0] === 2'b11 && alu_flags[`NEGFLAG] == 1'b1));
66
67 assign pc_increment = (state === `S_FETCH_1 || state === `S_FETCH_2 ||
68                       state === `S_FETCH_IMMEDIATE || state ===
69                           `S_ALU_IMMEDIATE ||
70                           state === `S_FETCH_ADDRESS_1 || state ===
71                               `S_FETCH_ADDRESS_2 ||
72                               state === `S_FETCH_ADDRESS_3 || state ===
73                                   `S_FETCH_ADDRESS_4 ||
74                                   state === `S_LOAD_JUMP_1 || state === `S_LOAD_JUMP_2)
    ;
75 assign mem_read = (state === `S_FETCH_1 || state === `S_FETCH_2 ||
76                  state === `S_FETCH_IMMEDIATE || state ===
77                      `S_ALU_IMMEDIATE ||
78                      state === `S_FETCH_ADDRESS_1 || state ===
79                          `S_FETCH_ADDRESS_2 ||
80                          state === `S_FETCH_MEMORY || state === `S_TEMP_FETCH ||
81                          state === `S_FETCH_ADDRESS_3 || state ===
82                              `S_FETCH_ADDRESS_4 ||
83                              state === `S_LOAD_JUMP_1 || state === `S_LOAD_JUMP_2);
84
85 assign mem_write = (state === `S_STORE_MEMORY || state === `S_TEMP_STORE);
86 assign latch_alu = (state === `S_ALU_OPERATION || state === `S_ALU_IMMEDIATE
87                    ||
88                    state === `S_TEMP_FETCH);
89 // On store 1, store the lower half, unless there is a multiply.
90 assign alu_store_high = (state === `S_STORE_RESULT_1 && opcode[15:11] ===
    `MULTIPLY);
91 assign alu_store_low = ((state === `S_STORE_RESULT_1 && opcode[15:11] !==
    `MULTIPLY) ||
92                        state === `S_STORE_RESULT_2 || state ===
93                            `S_TEMP_STORE);
94 assign jr_load_high = (state === `S_LOAD_JUMP_1);
95 assign jr_load_low = (state === `S_LOAD_JUMP_2);

```

```
88     assign mar_load_high = (state === `S_FETCH_ADDRESS_1 || state ===
      `S_FETCH_ADDRESS_3);
89     assign mar_load_low = (state === `S_FETCH_ADDRESS_2 || state ===
      `S_FETCH_ADDRESS_4);
90
91
92 endmodule
```

## A.12 CPU

```
1  /* cpu.v Top level module for IC design project
2  * Author: Steven Bell <steven.bell@student.oc.edu>
3  * Date: 5 December 2010
4  * $LastChangedDate: 2010-12-13 18:04:13 -0600 (Mon, 13 Dec 2010) $
5  */
6
7  module cpu(
8      input clock,
9      input reset,
10     inout[7:0] external_data_bus,
11     output[15:0] address_bus,
12     output ram_enable,
13     output rom_enable,
14     output write,
15     output write_bar,
16     output[4:0] state
17 );
18
19     // Signals from control module to datapath
20     wire pc_increment;
21     wire pc_set;
22     wire gp_read;
23     wire gp_write;
24     wire[2:0] gp_input_select;
25     wire[2:0] gp_output_select;
26     wire[2:0] gp_alu_output_select;
27     wire[3:0] alu_operation;
28     wire latch_alu;
29     wire alu_store_high;
30     wire alu_store_low;
31     wire mar_set_high;
32     wire mar_set_low;
33     wire ir_set_high;
34     wire ir_set_low;
35     wire jr_set_high;
36     wire jr_set_low;
37
38     // Signals from datapath to control module
39     wire[15:0] ir_value; // Instruction register
40     wire[2:0] alu_flags;
41
42     // Other signals from datapath
43     wire[15:0] pc_count;
44     wire[15:0] mar_value;
```

```
45  wire[7:0] data_bus; // Internal data bus
46
47  //wire[4:0] state;
48  wire[15:0] address_mux_out;
49
50  datapath dp(.clock(clock),
51             .reset(reset),
52             .pc_increment(pc_increment),
53             .pc_set(pc_set),
54             .gp_read(gp_read),
55             .gp_write(gp_write),
56             .gp_input_select(gp_input_select),
57             .gp_output_select(gp_output_select),
58             .gp_alu_output_select(gp_alu_output_select),
59             .alu_operation(alu_operation),
60             .latch_alu(latch_alu),
61             .alu_store_high(alu_store_high),
62             .alu_store_low(alu_store_low),
63             .mar_set_high(mar_set_high),
64             .mar_set_low(mar_set_low),
65             .ir_set_high(ir_set_high),
66             .ir_set_low(ir_set_low),
67             .jr_set_high(jr_set_high),
68             .jr_set_low(jr_set_low),
69             .pc_count(pc_count),
70             .mar_value(mar_value),
71             .ir_value(ir_value),
72             .flags(alu_flags),
73             .data_bus(data_bus));
74
75  control cm(.clock(clock),
76            .reset(reset),
77            .instruction(ir_value),
78            .state(state));
79
80  control_signals cs(.state(state),
81                   .opcode(ir_value),
82                   .alu_flags(alu_flags),
83                   .ir_load_high(ir_set_high),
84                   .ir_load_low(ir_set_low),
85                   .gp_read(gp_read),
86                   .gp_write(gp_write),
87                   .pc_set(pc_set),
88                   .pc_increment(pc_increment),
89                   .mem_read(mem_read),
90                   .mem_write(mem_write),
91                   .latch_alu(latch_alu),
92                   .alu_store_high(alu_store_high),
93                   .alu_store_low(alu_store_low),
94                   .jr_load_high(jr_set_high),
95                   .jr_load_low(jr_set_low),
96                   .mar_load_high(mar_set_high),
97                   .mar_load_low(mar_set_low),
98                   .alu_operation(alu_operation),
```

```
99         .gp_input_select(gp_input_select),
100         .gp_output_select(gp_output_select),
101         .gp_alu_output_select(gp_alu_output_select));
102
103     address_mux am(.pc_value(pc_count),
104                  .mar_value(mar_value),
105                  .state(state),
106                  .address_bus(address_mux_out));
107
108
109     memio mem(.read_memory(mem_read),
110              .write_memory(mem_write),
111              .internal_data_path(data_bus),
112              .external_data_path(external_data_bus),
113              .address_in(address_mux_out),
114              .address_out(address_bus),
115              .ram_enable(ram_enable),
116              .rom_enable(rom_enable),
117              .write(write),
118              .write_bar(write_bar));
119
120 endmodule
```

## B Verilog testbench code

### B.1 Generic 16-bit register

```
1 // register_16bit_test.abv
2 // ABEL test vector file for register_16_bit.v
3 // Author: Steven Bell <steven.bell@student.oc.edu>
4 // Date: 16 September 2010
5 // $LastChangedDate: 2010-09-23 08:44:58 -0500 (Thu, 23 Sep 2010) $
6
7 MODULE register_16bit
8
9 // Inputs
10     clock pin;
11     reset pin;
12     setHigh pin;
13     setLow pin;
14     instructionByteIn_7_ pin;
15     instructionByteIn_6_ pin;
16     instructionByteIn_5_ pin;
17     instructionByteIn_4_ pin;
18     instructionByteIn_3_ pin;
19     instructionByteIn_2_ pin;
20     instructionByteIn_1_ pin;
21     instructionByteIn_0_ pin;
22
23
24 // Outputs
25     instructionOut_15_ pin;
26     instructionOut_14_ pin;
27     instructionOut_13_ pin;
28     instructionOut_12_ pin;
29     instructionOut_11_ pin;
30     instructionOut_10_ pin;
31     instructionOut_9_ pin;
32     instructionOut_8_ pin;
33     instructionOut_7_ pin;
34     instructionOut_6_ pin;
35     instructionOut_5_ pin;
36     instructionOut_4_ pin;
37     instructionOut_3_ pin;
38     instructionOut_2_ pin;
39     instructionOut_1_ pin;
40     instructionOut_0_ pin;
41
42 // Buses
43     instructionByteIn = [instructionByteIn_7_, instructionByteIn_6_,
44         instructionByteIn_5_, instructionByteIn_4_, instructionByteIn_3_,
45         instructionByteIn_2_, instructionByteIn_1_, instructionByteIn_0_];
46     instructionOut = [instructionOut_15_, instructionOut_14_, instructionOut_13_
47         , instructionOut_12_, instructionOut_11_, instructionOut_10_,
48         instructionOut_9_, instructionOut_8_, instructionOut_7_,
49         instructionOut_6_, instructionOut_5_, instructionOut_4_,
50         instructionOut_3_, instructionOut_2_, instructionOut_1_,
```

```

        instructionOut_0_];
45
46     u = .u.; // Rising edge
47     d = .d.; // Falling edge
48     x = .x.; // Don't care
49
50 Test_vectors
51 ([clock,reset,setHigh,setLow,instructionByteIn] -> [instructionOut])
52 [u, 0, 0, 0, ^hde] -> [^h0000]; // Reset
53 [u, 1, 1, 0, ^hde] -> [^hde00]; // Set high
54 [u, 1, 0, 1, ^had] -> [^hdead]; // Set low
55 [u, 0, 1, 0, ^hfa] -> [^h0000]; // Reset while setting high; reset should
    take precedence
56 [u, 1, 1, 1, ^hfa] -> [^hfa00]; // Set both high and low; high should should
    take precedence, low will not be set
57 [u, 1, 0, 1, ^hce] -> [^hface]; // Set low
58 END

```

## B.2 Program counter

```

1 /* program_counter_test.v
2  * Verilog testbench for program_counter.v
3  * Author: Steven Bell <steven.bell@student.oc.edu>
4  * Date: 11 September 2010
5  * $LastChangedDate: 2010-11-23 09:34:58 -0600 (Tue, 23 Nov 2010) $
6  */
7
8 // Expected clock frequency is ~10 MHz = 100ns
9 // Tick freq is 1 GHz => 100 ticks is one clock cycle; resolution is 10 ps
10 `timescale 1 ns / 10 ps
11
12 module program_counter_test();
13
14     reg reset = 1'b1; // Active low reset for the part
15     reg increment = 1'b0;
16     reg set = 1'b0; // Set the count from the external source
17     reg[15:0] counterIn = 16'hfffe; // Input value to the counter
18     wire[15:0] counterOut; // Counter output for accessing memory
19
20     // Clock
21     reg clock = 1'b0;
22     always #50 clock = !clock; // 100 time ticks per cycle = 10 MHz clock
23
24     program_counter pc_dut(clock, reset, increment, set, counterIn, counterOut);
25
26     // Test
27     initial begin
28         // Reset
29         @(negedge clock) begin
30             reset = 1'b0;
31         end
32         checkCount(16'b0, __LINE__);
33
34         // Check that counter doesn't increment without the signal being asserted

```

```

35     @(negedge clock) begin
36         reset = 1'b1;
37     end
38     checkCount(16'b0, __LINE__);
39
40     // Load a value
41     @(negedge clock) begin
42         set = 1'b1;
43     end
44     checkCount(16'hfffe, __LINE__);
45
46     // Load a value while incrementing, the new value should take precedence
47     @(negedge clock) begin
48         set = 1'b1;
49         increment = 1'b1;
50     end
51     checkCount(16'hfffe, __LINE__);
52
53     // Increment
54     @(negedge clock) begin
55         set = 1'b0;
56         increment = 1'b1;
57     end
58     checkCount(16'hffff, __LINE__);
59
60     // Test rollover
61     @(negedge clock) begin
62         increment = 1'b1;
63     end
64     checkCount(16'h0000, __LINE__);
65
66     $finish;
67 end
68
69
70 // Waveform log file and monitoring
71 initial begin
72     $dumpfile("program_counter.vcd");
73     $dumpvars();
74     //$monitor("time: %d, value: %h", $time, counterOut);
75 end
76
77 task checkCount(input [15:0] expected, input integer lineNum);
78 begin
79     @(posedge clock) #5 begin
80         if(counterOut == expected) begin
81             $display("%3d_-_Test_passed", lineNum);
82         end
83         else begin
84             $display("%3d_-_Test_failed,_expected_%h,_got_%h", lineNum, expected,
85                 counterOut);
86         end
87     end
88 end

```

```

88     endtask
89
90 endmodule

```

### B.3 ALU

```

1  /* alu_test.v
2  * Testbench for the arithmetic logic unit
3  * Author: Steven Bell <steven.bell@student.oc.edu>
4  * Date: 30 September 2010
5  * $LastChangedDate: 2010-12-13 18:04:13 -0600 (Mon, 13 Dec 2010) $
6  */
7
8  // Expected clock frequency is ~10 MHz = 100ns
9  // Tick freq is 1 GHz => 100 ticks is one clock cycle; resolution is 10 ps
10 `timescale 1 ns / 10 ps
11
12 module alu_test();
13
14     reg reset = 1;
15     reg[7:0] registerParam; // Parameter from the general purpose registers (
16         primary)
17     reg[7:0] databusParam; // Parameter from the data bus (secondary)
18     reg[3:0] operation; // Operation to perform
19     wire[15:0] result; // ALU result
20     wire[2:0] flags;
21
22     // Clock
23     reg clock = 0;
24     always #50 clock = !clock; //100 time ticks per cycle = 10 MHz clock
25
26     // DUT
27     alu aluDUT(clock, reset, registerParam, databusParam, operation, result,
28         flags);
29
30     // Waveform file and monitoring
31     initial begin
32         $dumpfile("alu.vcd");
33         $dumpvars; // Dump everything
34         //$monitor("%d + %d = %d ", registerParam, databusParam, result);
35     end
36
37     initial begin
38         perform_reset();
39         // Test PASSTHROUGH
40         // Passthrough uses the secondary input (data bus) rather than the primary
41         // (GP registers)
42         test_operation(8'd0, 8'd2, `ALU_PASSTHROUGH, 8'd2, 3'b000, __LINE__); // "
43             Normal" passthrough
44         test_operation(8'd1, 8'd0, `ALU_PASSTHROUGH, 8'd0, 3'b100, __LINE__); //
45             Test zero
46         test_operation(8'd0, -8'd1, `ALU_PASSTHROUGH, -8'd1, 3'b001, __LINE__); //
47             Test negative

```

```

43
44 // Test ADD
45 test_operation(8'd1, 8'd2, `ALU_ADD, 8'd3, 3'b000, __LINE__); // Normal
    add
46 test_operation(8'd255, 8'd2, `ALU_ADD, 8'd1, 3'b010, __LINE__); // Test
    carry
47 test_operation(8'd125, 8'd10, `ALU_ADD, 8'd135, 3'b001, __LINE__); // Test
    negative
48 test_operation(-8'd15, 8'd10, `ALU_ADD, -8'd5, 3'b001, __LINE__); // Test
    negative another way
49 test_operation(8'd250, 8'd6, `ALU_ADD, 8'd0, 3'b110, __LINE__); // Test
    carry and zero
50
51 // Test SUBTRACT
52 // Note that the subtraction of a negative number causes a borrow because
    of the sign bit. This
53 // is not exactly the desired behavior, but it is the simplest logically
    consistent behavior.
54 test_operation(8'd10, 8'd6, `ALU_SUBTRACT, 8'd4, 3'b000, __LINE__); //
    Normal subtract
55 test_operation(8'd10, 8'd10, `ALU_SUBTRACT, 8'd0, 3'b100, __LINE__); //
    Test zero
56 test_operation(8'd10, 8'd15, `ALU_SUBTRACT, -8'd5, 3'b011, __LINE__); //
    Test borrow and negative
57 test_operation(8'd2, -8'd6, `ALU_SUBTRACT, 8'd8, 3'b010, __LINE__); //
    Subtraction of negative number
58 test_operation(-8'd128, 8'd1, `ALU_SUBTRACT, 8'd127, 3'b000, __LINE__); //
    Test underflow (no borrow)
59
60 // Test MULTIPLY
61 test_operation16(8'd10, 8'd6, `ALU_MULTIPLY, 16'd60, 3'b000, __LINE__); //
    Small multiply
62 test_operation16(8'd100, 8'd60, `ALU_MULTIPLY, 16'd6000, 3'b000, __LINE__)
    ; // Large multiply
63 test_operation16(-8'd1, -8'd1, `ALU_MULTIPLY, 16'd1, 3'b000, __LINE__); //
    Huge multiply
64 test_operation16(-8'd10, 8'd6, `ALU_MULTIPLY, -16'd60, 3'b000, __LINE__);
    // Normal multiply
65 // BUG: Signed multiplication doesn't work exactly as I expect, due to the
    input values having only
66 // 8 bits but producing a 16-bit result.
67
68 // Test AND
69 test_operation(8'b10101010, 8'b01100110, `ALU_AND, 8'b00100010, 3'b000,
    __LINE__); // Normal AND
70 test_operation(8'b10101010, 8'b01010101, `ALU_AND, 8'b00000000, 3'b100,
    __LINE__); // Test zero
71 test_operation(8'b10101010, 8'b11100110, `ALU_AND, 8'b10100010, 3'b001,
    __LINE__); // Test negative
72
73 // Test OR
74 test_operation(8'b00101010, 8'b01100110, `ALU_OR, 8'b01101110, 3'b000,
    __LINE__); // Normal OR

```

```
75 test_operation(8'b10101010, 8'b01100110, 'ALU_OR, 8'b11101110, 3'b001,
   __LINE__); // Test negative
76 test_operation(8'b00000000, 8'b00000000, 'ALU_OR, 8'b00000000, 3'b100,
   __LINE__); // Test zero
77
78 // Test LOGICAL_SHIFT_RIGHT
79 test_operation(8'b10101010, 8'b11111111, 'ALU_LOGICAL_SHIFT_RIGHT, 8'
   b01010101, 3'b000, __LINE__); // Shift out a 0
80 test_operation(8'b10101011, 8'b11111111, 'ALU_LOGICAL_SHIFT_RIGHT, 8'
   b01010101, 3'b010, __LINE__); // Shift out a 1
81 test_operation(8'b00000000, 8'b11111111, 'ALU_LOGICAL_SHIFT_RIGHT, 8'
   b00000000, 3'b100, __LINE__); // Test zero
82
83 // Test LOGICAL_SHIFT_LEFT
84 test_operation(8'b00101010, 8'b11111111, 'ALU_LOGICAL_SHIFT_LEFT, 8'
   b01010100, 3'b000, __LINE__); // Shift out a 0
85 test_operation(8'b10101010, 8'b11111111, 'ALU_LOGICAL_SHIFT_LEFT, 8'
   b01010100, 3'b010, __LINE__); // Shift out a 1
86 test_operation(8'b00000000, 8'b11111111, 'ALU_LOGICAL_SHIFT_LEFT, 8'
   b00000000, 3'b100, __LINE__); // Test zero
87 test_operation(8'b11000000, 8'b11111111, 'ALU_LOGICAL_SHIFT_LEFT, 8'
   b10000000, 3'b011, __LINE__); // Test carry and negative
88
89 // Test ARITH_SHIFT_RIGHT
90 test_operation(8'b00101010, 8'b11111111, 'ALU_ARITH_SHIFT_RIGHT, 8'
   b00010101, 3'b000, __LINE__); // Shift out a 0
91 test_operation(8'b00101011, 8'b11111111, 'ALU_ARITH_SHIFT_RIGHT, 8'
   b00010101, 3'b010, __LINE__); // Shift out a 1
92 test_operation(8'b00000000, 8'b11111111, 'ALU_ARITH_SHIFT_RIGHT, 8'
   b00000000, 3'b100, __LINE__); // Test zero
93 test_operation(8'b10101010, 8'b11111111, 'ALU_ARITH_SHIFT_RIGHT, 8'
   b11010101, 3'b001, __LINE__); // Test sign bit shift
94 // BUG: The test above fails - apparently Verilog only does an arithmetic
   shift if the value is signed
95 // so it seems to just be doing a logical shift. I could hack an
   arithmetic shift, or maybe there is
96 // a way to treat the input as a signed value?
97
98 // Test ARITH_SHIFT_LEFT
99 test_operation(8'b00101010, 8'b11111111, 'ALU_ARITH_SHIFT_LEFT, 8'
   b01010100, 3'b000, __LINE__); // Shift out a 0
100 test_operation(8'b10101010, 8'b11111111, 'ALU_ARITH_SHIFT_LEFT, 8'
   b01010100, 3'b010, __LINE__); // Shift out a 1
101 test_operation(8'b01101010, 8'b11111111, 'ALU_ARITH_SHIFT_LEFT, 8'
   b11010100, 3'b001, __LINE__); // Test overflow
102 test_operation(8'b00000000, 8'b11111111, 'ALU_ARITH_SHIFT_LEFT, 8'
   b00000000, 3'b100, __LINE__); // Test zero
103
104 // Test TWOS_COMPLEMENT
105 test_operation(8'b00000001, 8'b11111111, 'ALU_TWOS_COMPLEMENT, 8'b11111111
   , 3'b001, __LINE__); // 1 -> -1
106 test_operation(8'b10000000, 8'b11111111, 'ALU_TWOS_COMPLEMENT, 8'b10000000
   , 3'b001, __LINE__); // 128 -> -128
```

```

107 test_operation(8'd50, 8'b11111111, `ALU_TWOS_COMPLEMENT, -8'd50, 3'b001,
    __LINE__); // 50 -> -50
108 test_operation(-8'd50, 8'b11111111, `ALU_TWOS_COMPLEMENT, 8'd50, 3'b000,
    __LINE__); // -50 -> 50
109 test_operation(8'd0, 8'b11111111, `ALU_TWOS_COMPLEMENT, 8'd0, 3'b100,
    __LINE__); // 50 -> -50
110
111 // Test COMPLEMENT
112 test_operation(8'b10000001, 8'b11111111, `ALU_COMPLEMENT, 8'b01111110, 3'
    b000, __LINE__); // Test positive
113 test_operation(8'b00000001, 8'b11111111, `ALU_COMPLEMENT, 8'b11111110, 3'
    b001, __LINE__); // Test negative
114 test_operation(8'b11111111, 8'b11111111, `ALU_COMPLEMENT, 8'b00000000, 3'
    b100, __LINE__); // Test zero
115
116
117 @ (negedge clock) begin
118     $finish;
119 end
120 end
121
122 /* Cycles the reset input to the ALU to put it into a known state.
123 * Takes no parameters. */
124 task perform_reset();
125     begin
126         @ (negedge clock) begin
127             reset = 0;
128         end
129         @ (negedge clock) begin
130             reset = 1;
131         end
132     end
133 endtask
134
135
136 /* Performs an ALU operation giving an 8 bit result and determines whether the
137 * result and flags match the expected results and flags.
138 *
139 * tParam1 - First operand for the ALU; typically from the general purpose
    registers
140 * tParam2 - Second operand for the ALU; typically from the data bus
141 * tOperation - Operation to perform on the two operands
142 * tExpectedValue - Expected result of the operation
143 * tExpectedFlags - Expected zero, carry, and negative flags from the
    operation
144 * lineNum - Line number the test is on. This should generally be a
    preprocessor macro.
145 */
146 task test_operation(input [7:0] tParam1, input [7:0] tParam2, input [3:0]
    tOperation,
147                     input [7:0] tExpectedValue, input [2:0] tExpectedFlags,
    input integer lineNum);
148     begin
149         @ (negedge clock) begin // Wait for falling edge

```

```

150     registerParam = tParam1; // Set the inputs
151     databusParam = tParam2;
152     operation = tOperation; // Select the operation
153     end
154     @(posedge clock) begin // Wait for rising edge, and delay so the values
155         are stable
156         #5;
157         if(result[7:0] != tExpectedValue) begin // Check the outputs
158             $display("%3d_-_Test_failed!_Expected_result_%d,_got_%d", lineNum,
159                 tExpectedValue, result[7:0]);
160         end
161         else if(flags != tExpectedFlags) begin
162             $display("%3d_-_Test_failed!_Expected_flags_%b,_got_%b", lineNum,
163                 tExpectedFlags, flags);
164         end
165         else begin
166             $display("%3d_-_Test_passed", lineNum);
167         end
168     end // @(posedge clock)
169 end // task
170 endtask
171
172 /* Performs an ALU operation giving a 16-bit result and determines whether the
173 * result and flags match the expected results and flags.
174 *
175 * tParam1 - First operand for the ALU; typically from the general purpose
176 * registers
177 * tParam2 - Second operand for the ALU; typically from the data bus
178 * tOperation - Operation to perform on the two operands
179 * tExpectedValue - Expected result of the operation
180 * tExpectedFlags - Expected zero, carry, and negative flags from the
181 * operation
182 * lineNum - Line number the test is on. This should generally be a
183 * preprocessor macro.
184 */
185 task test_operation16(input [7:0] tParam1, input [7:0] tParam2, input [3:0]
186     tOperation,
187     input [15:0] tExpectedValue, input [2:0] tExpectedFlags,
188     input integer lineNum);
189     begin
190         @(negedge clock) begin // Wait for falling edge
191             registerParam = tParam1; // Set the inputs
192             databusParam = tParam2;
193             operation = tOperation; // Select the operation
194         end
195         @(posedge clock) begin // Wait for rising edge, and delay so the values
196             are stable
197             #5;
198             if(result[15:0] != tExpectedValue) begin // Check the outputs
199                 $display("%3d_-_Test_failed!_Expected_result_%d,_got_%d", lineNum,
200                     tExpectedValue, result[7:0]);
201             end
202             else if(flags != tExpectedFlags) begin

```

```

194     $display("%3d_-_Test_failed!_Expected_flags_%b,_got_%b", lineNum,
            tExpectedFlags, flags);
195     end
196     else begin
197         $display("%3d_-_Test_passed", lineNum);
198     end
199     end
200     // Print the result
201     end
202 endtask
203
204
205 endmodule

```

## B.4 ALU latch

```

1  /* alu_latch_test.v
2  * Testbench for the ALU latch
3  * Author: Steven Bell <steven.bell@student.oc.edu>
4  * Date: 14 October 2010
5  * $LastChangedDate: 2010-12-13 18:04:13 -0600 (Mon, 13 Dec 2010) $
6  */
7
8  // Expected clock frequency is ~10 MHz = 100ns
9  // Tick freq is 1 GHz => 100 ticks is one clock cycle; resolution is 10 ps
10 `timescale 1 ns / 10 ps
11
12 module alu_latch_test();
13     reg reset = 1;
14     reg[15:0] alu_result;
15     reg grab;
16     reg store_high;
17     reg store_low;
18     wire[7:0] databus; // Output of ALU latch goes to data bus (for now)
19
20     // Clock
21     reg clock = 0;
22     always #50 clock = ~clock; // 100 time ticks per cycle = 10 MHz clock
23
24     // DUT
25     alu_latch latchDUT(.clock(clock),
26                       .reset(reset),
27                       .alu_result(alu_result),
28                       .grab(grab),
29                       .store_high(store_high),
30                       .store_low(store_low),
31                       .out(databus));
32
33     // Monitoring and dumpfile
34     initial begin
35         $dumpfile("alu_latch.vcd");
36         $dumpvars; // Dump everything
37     end
38

```

```

39 // Beginning of test
40 initial begin
41     @(negedge clock) begin
42         reset = 0;
43     end
44     @(negedge clock) begin
45         reset = 1;
46     end
47
48     //      input      grab  high  low  expected
49     do_test(16'hbeef, 1'b1, 1'b0, 1'b0, 8'hzz, __LINE__); // Latch a value,
50         but don't put it out
51     do_test(16'hdead, 1'b0, 1'b1, 1'b0, 8'hbe, __LINE__); // Hold the value
52         and put the high half out
53     do_test(16'hdead, 1'b0, 1'b0, 1'b1, 8'hbf, __LINE__); // Hold the value
54         and put the low half out
55     do_test(16'hface, 1'b1, 1'b0, 1'b1, 8'hce, __LINE__); // Load a new value
56         and put it's low half out
57
58     $finish; // End the simulation
59 end
60
61 task do_test(input [15:0] r, input g, input h, input l, input [7:0] expected,
62     input integer line);
63 begin
64     @(negedge clock) begin
65         alu_result = r;
66         grab = g;
67         store_high = h;
68         store_low = l;
69     end
70     @(posedge clock) begin
71         #5; // Wait for the bits to flip, then check them
72         if(databus === expected) begin
73             $display("%3d_ Test_passed", line);
74         end
75         else begin
76             $display("%3d_ Test_failed! Expected_%h, got_%h", line, expected,
77                 databus);
78         end
79     end
80 end
81 endtask
82
83 endmodule

```

## B.5 General-purpose register block

```

1 /* gp_registers_test.v
2  * Testbench for the general purpose register set.
3  * Author: Steven Bell <steven.bell@student.oc.edu>
4  * Date: 23 September 2010
5  * $LastChangedDate: 2010-12-13 18:04:13 -0600 (Mon, 13 Dec 2010) $

```

```

6  */
7
8  // Expected clock frequency is ~10 MHz = 100ns
9  // Tick freq is 1 GHz => 100 ticks is one clock cycle; resolution is 10 ps
10 `timescale 1 ns / 10 ps
11
12 module gp_registers_test;
13
14     reg reset;
15     reg read_data; // save data to register
16     reg write_data; // write data from register
17     reg[2:0] input_select; // Register to put the input value into
18     reg[2:0] output_select; // Index of the register we want to put on the data
        bus output
19     reg[2:0] alu_output_select; // Index of the register we want to put on the
        ALU output
20
21     wire[7:0] data_bus; // Contains the input value to store, or the ouput we
        write
22     wire[7:0] alu_output_value; // Output bus to the ALU
23
24     // Data bus
25     reg[7:0] data_bus_driver; // Simulation of another device driving the data
        bus
26     reg data_bus_driven; // Whether or not something else is driving the data
        bus
27     assign data_bus = data_bus_driven ? data_bus_driver : 8'hzz;
28
29     // Test variables
30     parameter NUM_REGISTERS = 8; // Number of registers we have
31     integer regidx; // Used to cycle through the registers
32
33     // Clock
34     reg clock = 0;
35     always #50 clock = !clock; //100 time ticks per cycle = 10 MHz clock
36
37     // DUT
38     gp_registers registers(clock, reset, read_data, write_data,
39                             input_select, output_select, alu_output_select,
40                             data_bus, alu_output_value);
41
42     // Waveform log file
43     initial begin
44         $dumpfile("gp_registers.vcd");
45         $dumpvars; // Dump everything
46         //$monitor("time: %5d, clock: %d, reset: %d ", $time, clock, reset);
47     end
48
49     // Test code
50     initial begin
51         // Reset the registers
52         @(negedge clock)
53             reset = 0;
54         @(negedge clock)

```

```

55     reset = 1;
56
57
58     // Iterate through the registers and check that they were reset
59     for(regidx = 0; regidx < NUM_REGISTERS; regidx = regidx + 1) begin
60         // On the negative edge of the clock, request the value out both ports
61         @(negedge clock) begin
62             write_data = 1'b1; // The registers will be writing data out to the
                bus
63             read_data = 1'b0;
64             output_select = regidx;
65             alu_output_select = regidx;
66         end
67         // On the positive edge, read the output and check it
68         @(posedge clock) begin
69             #30; // Wait for the value to change - is there a better way?
70             if(data_bus != 8'h00 || alu_output_value != 8'h00) begin
71                 $display("Register_%d_not_reset!_Value_is_%d/%d", regidx, data_bus,
                alu_output_value);
72             end // if
73         end // posedge clock
74     end // checking reset
75
76
77     // Iterate through the registers and assign values that we will read
    later
78     for(regidx = 0; regidx < NUM_REGISTERS; regidx = regidx + 1) begin
79         // On the negative edge of the clock, set up the inputs
80         @(negedge clock) begin
81             write_data = 1'b0;
82             read_data = 1'b1; // The registers will now be reading data in
83             data_bus_driven = 1'b1; // And we will be driving the data bus
84             data_bus_driver = regidx + 10; // Store a unique nonzero value for
                each register
85             input_select = regidx;
86             output_select = (regidx + 1) % NUM_REGISTERS; // Read out a different
                register than
87             alu_output_select = (regidx + 1) % NUM_REGISTERS; // the one we are
                writing to
88         end
89         // On the positive edge, read the output and check it
90         @(posedge clock) begin
91             #30; // Wait for the value to change - is there a better way?
92             // BUG: This will fail on the last loop
93             //if(data_bus != 8'hzz || alu_output_value != 8'h00) begin
94             //    $display("Register %d not writing properly! Output is %d/%d",
                regidx, data_bus, alu_output_value);
95             //end // if
96         end // posedge clock
97     end // assigning test values
98
99
100    // Iterate through the registers and check that we can read the test
    values out again

```

```

101   for(regidx = 0; regidx < NUM_REGISTERS; regidx = regidx + 1) begin
102       // On the negative edge of the clock, request the value out both ports
103       @(negedge clock) begin
104           write_data = 1'b1; // The registers will be writing data out to the
               bus
105           read_data = 1'b0;
106           data_bus_driven = 1'b0; // The registers will be driving the bus
107           output_select = regidx;
108           alu_output_select = regidx;
109       end
110       // On the positive edge, read the output and check it
111       @(posedge clock) begin
112           #30; // Wait for the value to change - is there a better way?
113           if(data_bus != regidx + 10 || alu_output_value != regidx + 10) begin
114               $display("Register_%d_failed!_Expected_%d,_got_%d_and_%d", regidx,
               regidx + 10, data_bus, alu_output_value);
115           end // if
116       end // posedge clock
117   end // checking reset
118
119   @(negedge clock)
120   $finish;
121
122   end
123
124 endmodule

```

## B.6 Datapath

```

1  /* datapath_test.v
2  * Testbench for the arithmetic logic unit
3  * Author: Steven Bell <steven.bell@student.oc.edu>
4  * Date: 6 October 2010
5  * $LastChangedDate: 2010-12-13 18:04:13 -0600 (Mon, 13 Dec 2010) $
6  */
7
8  // Expected clock frequency is ~10 MHz = 100ns
9  // Tick freq is 1 GHz => 100 ticks is one clock cycle; resolution is 10 ps
10 `timescale 1 ns / 10 ps
11
12 `include "constants.v"
13
14 module datapath_test();
15
16     parameter NAME_LEN = 239; // Length of the string in bytes, minus one (30 *
               8 - 1)
17
18     reg reset;
19     reg gp_read;
20     reg gp_write;
21     reg pc_increment;
22     reg pc_set;
23     reg[2:0] gp_input_select;
24     reg[2:0] gp_output_select;

```

```

25  reg[2:0] gp_alu_output_select;
26  reg[3:0] alu_operation;
27  reg latch_alu;
28  reg alu_store_high;
29  reg alu_store_low;
30  reg mar_set_high;
31  reg mar_set_low;
32  reg ir_set_high;
33  reg ir_set_low;
34  reg jr_set_high;
35  reg jr_set_low;
36
37  wire[15:0] pc_count; // Program counter output
38  wire[15:0] mar_value; // Memory address register output
39  wire[15:0] ir_value; // Instruction register output
40  wire[2:0] alu_flags; // ALU flags
41
42  /* Data bus
43   * Because the data bus is a bidirectional port, we can't treat it as a wire
44   * (which must have continuous assignment) or a reg (which can't be
45   * continuously
46   * assigned). We have to create a reg which holds the value, and then use
47   * an
48   * assign statement to pick what we want. */
49  reg[7:0] data_bus_driver; // Holds the value we want to put onto the bus
50  reg data_bus_input; // When true, the data bus reads from data_bus_driver,
51  // when false, it's an output
52  wire[7:0] data_bus; // Wire connected to the DUT
53
54  assign data_bus = data_bus_input ? data_bus_driver : 8'hzz;
55
56  // Testbench variables
57  integer regIdx; // Index variable for the general purpose registers
58  reg[NAME_LEN:0] testName; // Holds the printed name for some tests
59
60  // Clock
61  reg clock = 0;
62  always #50 clock = !clock; // 100 time ticks per cycle = 10 MHz clock
63
64  // DUT
65  datapath dp(
66      .clock(clock),
67      .reset(reset),
68      .pc_increment(pc_increment),
69      .pc_set(pc_set),
70      .gp_read(gp_read),
71      .gp_write(gp_write),
72      .gp_input_select(gp_input_select),
73      .gp_output_select(gp_output_select),
74      .gp_alu_output_select(gp_alu_output_select),
75      .alu_operation(alu_operation),
76      .latch_alu(latch_alu),
77      .alu_store_high(alu_store_high),
78      .alu_store_low(alu_store_low),

```

```

76         .mar_set_high(mar_set_high),
77         .mar_set_low(mar_set_low),
78         .ir_set_high(ir_set_high),
79         .ir_set_low(ir_set_low),
80         .jr_set_high(jr_set_high),
81         .jr_set_low(jr_set_low),
82         .pc_count(pc_count),
83         .mar_value(mar_value),
84         .ir_value(ir_value),
85         .flags(alu_flags),
86         .data_bus(data_bus));
87
88     // Waveform log file and monitoring
89     initial begin
90         $dumpfile("datapath.vcd");
91         $dumpvars;
92         $monitor();
93     end
94
95     // Beginning of the test
96     initial begin
97         // Reset
98         @(negedge clock)
99         resetSignals();
100        reset = 1'b0;
101        @(posedge clock) #5
102        check16("Program_counter_reset", pc_count, 16'b0, __LINE__);
103        check16("MAR_reset", mar_value, 16'b0, __LINE__);
104        check16("Instruction_register_reset", ir_value, 16'b0, __LINE__);
105
106        // Load instruction register from the data bus
107        @(negedge clock)
108        resetSignals();
109        data_bus_driver = 8'hef;
110        data_bus_input = 1'b1;
111        ir_set_low = 1'b1;
112        @(posedge clock) #5
113        check16("Instruction_register_load_low", ir_value, 16'h00ef, __LINE__);
114        @(negedge clock)
115        resetSignals();
116        data_bus_driver = 8'hbe;
117        data_bus_input = 1'b1;
118        ir_set_high = 1'b1;
119        @(posedge clock) #5
120        check16("Instruction_register_load_high", ir_value, 16'hbeef, __LINE__);
121
122        // Load the jump register from the data bus
123        // The jump register doesn't have an output except to the program counter
124        @(negedge clock)
125        resetSignals();
126        data_bus_driver = 8'hce;
127        data_bus_input = 1'b1;
128        jr_set_low = 1'b1;
129        @(negedge clock)

```

```

130     resetSignals();
131     data_bus_driver = 8'hfa;
132     data_bus_input = 1'b1;
133     jr_set_high = 1'b1;
134     // Put the jump register value into the program counter and check it
135     @(negedge clock)
136     resetSignals();
137     pc_set = 1'b1;
138     @(posedge clock) #5
139     check16("Jump_register", pc_count, 16'hface, __LINE__);
140
141     // Increment the program counter
142     @(negedge clock)
143     resetSignals();
144     pc_increment = 1'b1;
145     @(posedge clock) #5
146     check16("Program_counter_increment", pc_count, 16'hfacf, __LINE__);
147
148     // Load each general purpose register from the data bus
149     resetSignals();
150     for(regIdx = 0; regIdx < 8; regIdx = regIdx + 1) begin
151         @(negedge clock)
152         data_bus_driver = regIdx; // Load each register with its index
153         data_bus_input = 1'b1; // We are driving the data bus
154         gp_input_select = regIdx; // regIdx will never be more than 7, so this
            is safe
155         gp_read = 1'b1; // Read from the data bus into the register
156     end
157
158     // Put each general purpose register back on the data bus and check it
159     for(regIdx = 0; regIdx < 8; regIdx = regIdx + 1) begin
160         @(negedge clock)
161         resetSignals();
162         gp_output_select = regIdx;
163         gp_write = 1'b1;
164         @(posedge clock) #5
165         $sformat(testName, "General-purpose_reg_#%1d", regIdx);
166         checkDatabus(testName, regIdx, __LINE__);
167     end
168
169     // Load memory address register from the data bus
170     @(negedge clock)
171     resetSignals();
172     data_bus_driver = 8'had;
173     data_bus_input = 1'b1;
174     mar_set_low = 1'b1;
175     @(posedge clock) #5
176     check16("MAR_load_low", mar_value, 16'h00ad, __LINE__);
177     @(negedge clock)
178     resetSignals();
179     data_bus_driver = 8'hde;
180     data_bus_input = 1'b1;
181     mar_set_high = 1'b1;
182     @(posedge clock) #5

```

```
183     check16("MAR_load_high", mar_value, 16'hdead, __LINE__);
184
185     // Load memory address register from the GP registers
186     // This loads the high byte and then the low byte, to ensure that that
187     // works just
188     // as well as loading low and then high.
189     // This code assumes that the register values are equal to their addresses
190     @(negedge clock)
191     resetSignals();
192     gp_output_select = 3'h4;
193     gp_write = 1'b1;
194     mar_set_high = 1'b1;
195     @(negedge clock)
196     resetSignals();
197     gp_output_select = 3'h5;
198     gp_write = 1'b1;
199     mar_set_low = 1'b1;
200     @(posedge clock) #5
201     check16("MAR_load_from_GPR", mar_value, 16'h0405, __LINE__);
202
203     // Perform 2-operand ALU operation using GP registers
204     // Add registers 5 and 6, and check that the result is 11
205     @(negedge clock)
206     resetSignals();
207     gp_alu_output_select = 3'h6; // Register 6 directly to ALU
208     gp_output_select = 3'h5; // Register 5 on data bus
209     gp_write = 1'b1;
210     alu_operation = `ALU_ADD;
211     latch_alu = 1'b1; // Latch this result so we can put it on the data bus
212     // There's no way to see the ALU output except to put it on the data bus,
213     // so that's what we do next.
214     @(negedge clock)
215     resetSignals();
216     alu_store_low = 1'b1;
217     @(posedge clock)
218     checkDatabus("ALU_binary_from_GP", 8'd11, __LINE__);
219
220     // Perform 2-operand ALU operation using an immediate
221     // Multiply 6 by 100 and check that the result is 0x0258 (600)
222     @(negedge clock)
223     resetSignals();
224     gp_alu_output_select = 3'h6; // Register 6 directly to ALU
225     data_bus_driver = 8'd100; // Immediate value of 100
226     data_bus_input = 1'b1;
227     alu_operation = `ALU_MULTIPLY;
228     latch_alu = 1'b1; // Latch this result so we can put it on the data bus
229     @(negedge clock)
230     resetSignals();
231     alu_store_high = 1'b1;
232     @(negedge clock)
233     checkDatabus("ALU_binary_with_immediate_high", 8'h02, __LINE__);
234     @(negedge clock)
235     resetSignals();
236     alu_store_low = 1'b1;
```

```
236 @ (posedge clock)
237 checkDatabus("ALU_binary_with_immediate_low", 8'h58, __LINE__);
238
239
240 // Perform 1-operand ALU operation using GP register
241 // Invert register 1 to get 0b11111110 (0xFE)
242 @ (negedge clock)
243 resetSignals();
244 gp_alu_output_select = 3'h1; // Register 6 directly to ALU
245 alu_operation = `ALU_COMPLEMENT;
246 latch_alu = 1'b1; // Latch this result so we can put it on the data bus
247 @ (negedge clock)
248 resetSignals();
249 alu_store_low = 1'b1;
250 @ (posedge clock)
251 checkDatabus("ALU_unary_from_GP", 8'hfe, __LINE__);
252
253 // Perform ALU passthrough
254 // Copy the value 125 into the ALU latch and write it back out to the
    databus
255 @ (negedge clock)
256 resetSignals();
257 data_bus_driver = 8'd125;
258 data_bus_input = 1'b1;
259 alu_operation = `ALU_PASSTHROUGH;
260 latch_alu = 1'b1; // Latch this result so we can put it on the data bus
261 @ (negedge clock)
262 resetSignals();
263 alu_store_low = 1'b1;
264 @ (negedge clock)
265 checkDatabus("ALU_passthrough", 8'd125, __LINE__);
266
267
268
269 @ (negedge clock)
270 $finish;
271 end // End of the test
272
273 // Sets all of the signals to their inactive state.
274 task resetSignals();
275 begin
276     reset = 1;
277     data_bus_input = 1'b0;
278     pc_increment = 1'b0;
279     pc_set = 1'b0;
280     gp_read = 0;
281     gp_write = 0;
282     gp_input_select = 0;
283     gp_output_select = 0;
284     gp_alu_output_select = 0;
285     alu_operation = 0;
286     latch_alu = 1'b0;
287     alu_store_high = 1'b0;
288     alu_store_low = 1'b0;
```

```

289     mar_set_high = 0;
290     mar_set_low = 0;
291     ir_set_high = 0;
292     ir_set_low = 0;
293     jr_set_high = 0;
294     jr_set_low = 0;
295 end
296 endtask
297
298 task check16(input [NAME_LEN:0] name, input [15:0] value, input [15:0] expected
    , input integer lineNum);
299 begin
300     if(value === expected) begin
301         $display("%3d_ Test_passed: %s", lineNum, name);
302     end
303     else begin
304         $display("%3d_ Test_failed, expected %h, got %h: %s", lineNum,
            expected, value, name);
305     end
306 end
307 endtask
308
309 task checkDatabus(input [NAME_LEN:0] name, input [7:0] expected, input
    integer lineNum);
310 begin
311     if(data_bus === expected) begin
312         $display("%3d_ Test_passed: %s", lineNum, name);
313     end
314     else begin
315         $display("%3d_ Test_failed, expected %h, got %h: %s", lineNum,
            expected, data_bus, name);
316     end
317 end
318 endtask
319
320 endmodule

```

## B.7 Control module state machine

```

1 /* control_test.v
2  * Testbench for the control module. This code uses an external text file,
3  * control_vectors, which contains a series of control module states and
4  * opcodes
5  * where necessary. After performing a reset, the code reads a line from the
6  * file, executes a clock cycle, and checks that the control module entered
7  * the
8  * specified state.
9  * This code does not check that the state vectors are outputted properly. It
10  * only
11  * checks that the state sequence is correct.
12  * Author: Steven Bell <steven.bell@student.oc.edu>
13  * Date: 23 November 2010
14  * $LastChangedDate: 2010-12-13 18:04:13 -0600 (Mon, 13 Dec 2010) $
15  */

```

```

13
14 // Expected clock frequency is ~10 MHz = 100ns
15 // Tick freq is 1 GHz => 100 ticks is one clock cycle; resolution is 10 ps
16 `timescale 1 ns / 10 ps
17
18 module control_test();
19     parameter STRING_LEN = 239; // Maximum length of the strings we read in bits
        (minus one)
20     reg reset = 1; // Reset signal for the part
21     reg[15:0] instruction_register; // Dummy instruction register which we load
        from the file
22
23     wire[4:0] control_state; // Current state of the control module
24
25     // Clock
26     reg clock = 0;
27     always #50 clock = ~clock;
28
29     // DUT
30     control controlDUT(.clock(clock),
31                       .reset(reset),
32                       .instruction(instruction_register),
33                       .state(control_state));
34
35     // Logging
36     initial begin
37         $dumpfile("control.vcd");
38         $dumpvars(); // Dump everything
39     end
40
41     // Variables for reading the file
42     integer file;
43     integer r; // Result from scanf; required by verilog spec, but not used
44     reg[STRING_LEN:0] stateString;
45     reg[7:0] opcodeByte;
46
47
48     // Beginning of test
49     initial begin
50         // Perform reset
51         @(negedge clock)
52         reset = 1'b0;
53         @(negedge clock)
54         reset = 1'b1;
55
56         // Start reading the file
57         file = $fopen("control_vectors", "r");
58
59         // Loop until we get to the end of the file
60         while(!$feof(file)) begin
61             // Read a line with a desired state
62             r = $fscanf(file, "%s", stateString);
63             while(isComment(stateString)) begin // Keep reading and printing
                comments until we get an instruction

```

```

64     $display("%s", stateString);
65     r = $fscanf(file, "%s", stateString);
66     end
67
68     // On the rising edge of the clock, the module should be in that state
69     @(posedge clock) #5
70     testOpcode(stateString, control_state);
71
72     // If the state is a fetch, then read the opcode byte from the file and
73     // set the instruction register appropriately
74     if(stateString == "FETCH_1") begin
75         r = $fscanf(file, "%2h", opcodeByte);
76         instruction_register[15:8] = opcodeByte;
77     end
78     else if(stateString == "FETCH_2") begin
79         r = $fscanf(file, "%2h", opcodeByte);
80         instruction_register[7:0] = opcodeByte;
81     end
82
83     end
84
85
86     // How do we test jumps??
87
88     // End of the test
89     $display("Test_completed_successfully!");
90     $finish;
91     end
92
93     /* Compares a state string (e.g, "FETCH_2") to a state number (e.g, 2) to
94     * see if they match.  If so, it prints the state string.  If not, it prints
95     * an error message and quits the simulation */
96     task testOpcode(input [20*8-1:0] string, input [4:0] state);
97         reg [4:0] desiredState; // State value determined from the string
98     begin
99
100         case(string)
101             "RESET":
102                 desiredState = `S_RESET;
103             "FETCH_1":
104                 desiredState = `S_FETCH_1;
105             "FETCH_2":
106                 desiredState = `S_FETCH_2;
107             "ALU_OPERATION":
108                 desiredState = `S_ALU_OPERATION;
109             "ALU_IMMEDIATE":
110                 desiredState = `S_ALU_IMMEDIATE;
111             "STORE_RESULT_1":
112                 desiredState = `S_STORE_RESULT_1;
113             "STORE_RESULT_2":
114                 desiredState = `S_STORE_RESULT_2;
115             "FETCH_IMMEDIATE":
116                 desiredState = `S_FETCH_IMMEDIATE;
117             "COPY_REGISTER":

```

```

118     desiredState = `S_COPY_REGISTER;
119     "FETCH_ADDRESS_1":
120     desiredState = `S_FETCH_ADDRESS_1;
121     "FETCH_ADDRESS_2":
122     desiredState = `S_FETCH_ADDRESS_2;
123     "FETCH_MEMORY":
124     desiredState = `S_FETCH_MEMORY;
125     "STORE_MEMORY":
126     desiredState = `S_STORE_MEMORY;
127     "TEMP_FETCH":
128     desiredState = `S_TEMP_FETCH;
129     "FETCH_ADDRESS_3":
130     desiredState = `S_FETCH_ADDRESS_3;
131     "FETCH_ADDRESS_4":
132     desiredState = `S_FETCH_ADDRESS_4;
133     "TEMP_STORE":
134     desiredState = `S_TEMP_STORE;
135     "LOAD_JUMP_1":
136     desiredState = `S_LOAD_JUMP_1;
137     "LOAD_JUMP_2":
138     desiredState = `S_LOAD_JUMP_2;
139     "EXECUTE_JUMP":
140     desiredState = `S_EXECUTE_JUMP;
141     "HALT":
142     desiredState = `S_HALT;
143     default:
144     desiredState = 5'b00000;
145     endcase
146
147     if(state == desiredState) begin
148         $display("%h, %s", instruction_register, string);
149     end
150     else begin
151         $display("Test_failed!_Expected_state_%s,_but_got_%d", string, state);
152         $display("Instruction_register_contained_%h_at_failure",
153             instruction_register);
154         $finish; // quit the simulation
155     end
156 end
157 endtask
158
159 function isComment(input [STRING_LEN:0] str);
160 begin
161     if(str[7:0] == "#") begin
162         isComment = 1'b1;
163     end
164     else begin
165         isComment = 1'b0;
166     end
167 end
168 endfunction
169
170 endmodule

```

## B.8 Control signals translator

```
1 /* control_signals_test.v Testbench for control signals module, which
2 * translates the control module state into the vector of control signals.
3 * Author: Steven Bell <steven.bell@student.oc.edu>
4 * Date: 2 December 2010
5 * $LastChangedDate$
6 */
7
8 // Expected clock frequency is ~10 MHz = 100ns
9 // Tick freq is 1 GHz => 100 ticks is one clock cycle; resolution is 10 ps
10 `timescale 1 ns / 10 ps
11
12 // Need to see if there's something like C's #ifndef for headers
13 //`include "constants.vh"
14
15 module control_signals_test();
16     // Inputs to the DUT
17     reg[4:0] state;
18     reg[15:0] opcode;
19     reg[2:0] alu_flags;
20
21     // Outputs from the DUT
22     wire ir_load_high;
23     wire ir_load_low;
24     wire gp_read;
25     wire gp_write;
26     wire pc_set;
27     wire pc_increment;
28     wire mem_read;
29     wire mem_write;
30     wire latch_alu;
31     wire alu_store_high;
32     wire alu_store_low;
33     wire jr_load_high;
34     wire jr_load_low;
35     wire mar_load_high;
36     wire mar_load_low;
37     wire[3:0] alu_operation;
38     wire[2:0] gp_input_select;
39     wire[2:0] gp_output_select;
40     wire[2:0] gp_alu_output_select;
41
42     // Variables representing the desired values for the outputs
43     reg d_ir_load_high;
44     reg d_ir_load_low;
45     reg d_gp_primary_addr;
46     reg d_gp_alu_addr;
47     reg d_gp_read;
48     reg d_gp_write;
49     reg d_pc_set;
50     reg d_pc_increment;
51     reg d_mem_read;
52     reg d_mem_write;
```

```

53  reg d_latch_alu;
54  reg d_alu_store_high;
55  reg d_alu_store_low;
56  reg d_jr_load_high;
57  reg d_jr_load_low;
58  reg d_mar_load_high;
59  reg d_mar_load_low;
60  reg[3:0] d_alu_operation;
61  reg[2:0] d_gp_input_select;
62  reg[2:0] d_gp_output_select;
63  reg[2:0] d_gp_alu_output_select;
64
65  // Variables for module tasks
66  reg passed = 1'b1;
67
68  // DUT
69  control_signals csDUT(.state(state),
70                        .opcode(opcode),
71                        .alu_flags(alu_flags),
72                        .ir_load_high(ir_load_high),
73                        .ir_load_low(ir_load_low),
74                        .gp_read(gp_read),
75                        .gp_write(gp_write),
76                        .pc_set(pc_set),
77                        .pc_increment(pc_increment),
78                        .mem_read(mem_read),
79                        .mem_write(mem_write),
80                        .latch_alu(latch_alu),
81                        .alu_store_high(alu_store_high),
82                        .alu_store_low(alu_store_low),
83                        .jr_load_high(jr_load_high),
84                        .jr_load_low(jr_load_low),
85                        .mar_load_high(mar_load_high),
86                        .mar_load_low(mar_load_low),
87                        .alu_operation(alu_operation),
88                        .gp_input_select(gp_input_select),
89                        .gp_output_select(gp_output_select),
90                        .gp_alu_output_select(gp_alu_output_select));
91
92  // Monitoring and dumpfile
93  initial begin
94      $dumpfile("control_signals.vcd");
95      $dumpvars; // Dump everything
96  end
97
98  // Beginning of test
99  initial begin
100     // Reset
101     state = `S_RESET;
102     opcode = 16'hbeef;
103     zeroDesired();
104
105     checkSignals(__LINE__);
106

```

```
107
108 // Fetch 1
109 state = `S_FETCH_1;
110 opcode = 16'hface;
111 zeroDesired();
112 d_dir_load_high = 1;
113 d_mem_read = 1;
114 d_pc_increment = 1;
115
116 checkSignals(__LINE__);
117
118
119 // Fetch 2
120 state = `S_FETCH_2;
121 opcode = 16'hface;
122 zeroDesired();
123 d_dir_load_low = 1;
124 d_mem_read = 1;
125 d_pc_increment = 1;
126
127 checkSignals(__LINE__);
128
129
130 // ALU operation with immediate, no write from register
131 state = `S_ALU_OPERATION;
132 opcode = 16'h0c00;
133 zeroDesired();
134 d_latch_alu = 1;
135
136 checkSignals(__LINE__);
137
138
139 // ALU operation with two registers
140 state = `S_ALU_OPERATION;
141 opcode = 16'h0800;
142 zeroDesired();
143 d_gp_write = 1;
144 d_latch_alu = 1;
145
146 checkSignals(__LINE__);
147
148
149 // Store result
150 state = `S_STORE_RESULT_1;
151 opcode = 16'h0c00;
152 zeroDesired();
153 d_gp_read = 1;
154 d_alu_store_low = 1;
155
156 checkSignals(__LINE__);
157
158
159 // Store a multiply
160 state = `S_STORE_RESULT_1;
```

```
161 opcode = 16'h1820;
162 zeroDesired();
163 d_gp_read = 1;
164 d_alu_store_high = 1;
165 checkSignals(__LINE__);
166
167 // Store the second half of the multiply
168 state = `S_STORE_RESULT_2;
169 opcode = 16'h1820;
170 zeroDesired();
171 d_gp_read = 1;
172 d_alu_store_low = 1;
173 checkSignals(__LINE__);
174
175 // Do a copy
176 state = `S_COPY_REGISTER;
177 opcode = 16'h0000;
178 zeroDesired();
179 d_gp_read = 1;
180 d_gp_write = 1;
181 checkSignals(__LINE__);
182
183 // Do an address fetch
184 state = `S_FETCH_ADDRESS_1;
185 opcode = 16'h0000;
186 zeroDesired();
187 d_pc_increment = 1;
188 d_mem_read = 1;
189 d_mar_load_high = 1;
190 checkSignals(__LINE__);
191
192 state = `S_FETCH_ADDRESS_2;
193 opcode = 16'h0000;
194 zeroDesired();
195 d_pc_increment = 1;
196 d_mem_read = 1;
197 d_mar_load_low = 1;
198 checkSignals(__LINE__);
199
200 // Memory fetch
201 state = `S_FETCH_MEMORY;
202 opcode = 16'h0000;
203 zeroDesired();
204 d_gp_read = 1;
205 d_mem_read = 1;
206 checkSignals(__LINE__);
207
208 // Memory store
209 state = `S_STORE_MEMORY;
210 opcode = 16'h0000;
211 zeroDesired();
212 d_gp_write = 1;
213 d_mem_write = 1;
214 checkSignals(__LINE__);
```

```
215
216 // Do a move starting with the temp fetch
217 state = `S_TEMP_FETCH;
218 opcode = 16'h9000;
219 zeroDesired();
220 d_latch_alu = 1;
221 d_mem_read = 1;
222 checkSignals(__LINE__);
223
224 // Continue move; get the destination address
225 state = `S_FETCH_ADDRESS_3;
226 opcode = 16'h9000;
227 zeroDesired();
228 d_pc_increment = 1;
229 d_mem_read = 1;
230 d_mar_load_high = 1;
231 checkSignals(__LINE__);
232
233 state = `S_FETCH_ADDRESS_4;
234 opcode = 16'h9000;
235 zeroDesired();
236 d_pc_increment = 1;
237 d_mem_read = 1;
238 d_mar_load_low = 1;
239 checkSignals(__LINE__);
240
241 // Do the temp store
242 state = `S_TEMP_STORE;
243 opcode = 16'h9000;
244 zeroDesired();
245 d_alu_store_low = 1;
246 d_mem_write = 1;
247 checkSignals(__LINE__);
248
249 // Load the jump register
250 state = `S_LOAD_JUMP_1;
251 opcode = 16'h7800;
252 zeroDesired();
253 d_pc_increment = 1;
254 d_mem_read = 1;
255 d_jr_load_high = 1;
256 checkSignals(__LINE__);
257
258 state = `S_LOAD_JUMP_2;
259 opcode = 16'h7800;
260 zeroDesired();
261 d_pc_increment = 1;
262 d_mem_read = 1;
263 d_jr_load_low = 1;
264 checkSignals(__LINE__);
265
266 // Test jump always
267 state = `S_EXECUTE_JUMP;
268 opcode = 16'h7800;
```

```
269 zeroDesired();
270 d_pc_set = 1;
271 checkSignals(__LINE__);
272
273 // Test jump if carry - don't jump
274 state = `S_EXECUTE_JUMP;
275 opcode = 16'h7801;
276 alu_flags = 3'b000;
277 zeroDesired();
278 d_pc_set = 0;
279 checkSignals(__LINE__);
280
281 // Jump if carry - do jump
282 state = `S_EXECUTE_JUMP;
283 opcode = 16'h7801;
284 alu_flags = 3'b010;
285 zeroDesired();
286 d_pc_set = 1;
287 checkSignals(__LINE__);
288
289 // Test jump if zero - don't jump
290 state = `S_EXECUTE_JUMP;
291 opcode = 16'h7802;
292 alu_flags = 3'b000;
293 zeroDesired();
294 d_pc_set = 0;
295 checkSignals(__LINE__);
296
297 // Jump if zero - do jump
298 state = `S_EXECUTE_JUMP;
299 opcode = 16'h7802;
300 alu_flags = 3'b100;
301 zeroDesired();
302 d_pc_set = 1;
303 checkSignals(__LINE__);
304
305 // Test jump if negative - don't jump
306 state = `S_EXECUTE_JUMP;
307 opcode = 16'h7803;
308 alu_flags = 3'b000;
309 zeroDesired();
310 d_pc_set = 0;
311 checkSignals(__LINE__);
312
313 // Jump if negative - do jump
314 state = `S_EXECUTE_JUMP;
315 opcode = 16'h7803;
316 alu_flags = 3'b001;
317 zeroDesired();
318 d_pc_set = 1;
319 checkSignals(__LINE__);
320
321
322 // Check halt
```

```
323     state = `HALT;
324     opcode = 16'hff00;
325     zeroDesired();
326     checkSignals(__LINE__);
327
328
329
330
331
332
333     $finish;
334 end
335
336 task checkSignals(input integer lineNum);
337 begin
338     #100
339     if(ir_load_high != d_ir_load_high) begin
340         $display("%4d:_IR_high_test_failed;_expected_%h_but_got_%h", lineNum,
341             d_ir_load_high, ir_load_high);
342         passed = 1'b0;
343     end
344     if(ir_load_low != d_ir_load_low) begin
345         $display("%4d:_IR_low_test_failed;_expected_%h_but_got_%h", lineNum,
346             d_ir_load_low, ir_load_low);
347         passed = 1'b0;
348     end
349     if(gp_read != d_gp_read) begin
350         $display("%4d:_GP_read_test_failed;_expected_%h_but_got_%h", lineNum,
351             d_gp_read, gp_read);
352         passed = 1'b0;
353     end
354     if(gp_write != d_gp_write) begin
355         $display("%4d:_GP_write_test_failed;_expected_%h_but_got_%h", lineNum,
356             d_gp_write, gp_write);
357         passed = 1'b0;
358     end
359     if(pc_set != d_pc_set) begin
360         $display("%4d:_PC_set_test_failed;_expected_%h_but_got_%h", lineNum,
361             d_pc_set, pc_set);
362         passed = 1'b0;
363     end
364     if(pc_increment != d_pc_increment) begin
365         $display("%4d:_PC_increment_test_failed;_expected_%h_but_got_%h",
366             lineNum, d_pc_increment, pc_increment);
367         passed = 1'b0;
368     end
369     if(mem_read != d_mem_read) begin
370         $display("%4d:_Memory_read_test_failed;_expected_%h_but_got_%h",
371             lineNum, d_mem_read, mem_read);
372         passed = 1'b0;
373     end
374     if(mem_write != d_mem_write) begin
375         $display("%4d:_Memory_write_test_failed;_expected_%h_but_got_%h",
376             lineNum, d_mem_write, mem_write);
```

```

369     passed = 1'b0;
370 end
371 if(latch_alu != d_latch_alu) begin
372     $display("%4d: ALU_latch_test_failed;_expected_%h_but_got_%h", lineNum,
373             d_latch_alu, latch_alu);
374     passed = 1'b0;
375 end
376 if(alu_store_high != d_alu_store_high) begin
377     $display("%4d: ALU_store_high_test_failed;_expected_%h_but_got_%h",
378             lineNum, d_alu_store_high, alu_store_high);
379     passed = 1'b0;
380 end
381 if(alu_store_low != d_alu_store_low) begin
382     $display("%4d: ALU_store_low_test_failed;_expected_%h_but_got_%h",
383             lineNum, d_alu_store_low, alu_store_low);
384     passed = 1'b0;
385 end
386 if(jr_load_high != d_jr_load_high) begin
387     $display("%4d: JR_high_test_failed;_expected_%h_but_got_%h", lineNum,
388             d_jr_load_high, jr_load_high);
389     passed = 1'b0;
390 end
391 if(ir_load_low != d_ir_load_low) begin
392     $display("%4d: JR_low_test_failed;_expected_%h_but_got_%h", lineNum,
393             d_jr_load_low, jr_load_low);
394     passed = 1'b0;
395 end
396 if(mar_load_high != d_mar_load_high) begin
397     $display("%4d: MAR_high_test_failed;_expected_%h_but_got_%h", lineNum,
398             d_mar_load_high, mar_load_high);
399     passed = 1'b0;
400 end
401 if(mar_load_low != d_mar_load_low) begin
402     $display("%4d: MAR_low_test_failed;_expected_%h_but_got_%h", lineNum,
403             d_mar_load_low, mar_load_low);
404     passed = 1'b0;
405 end
406 if(passed === 1'b1) begin
407     $display("%4d: Test_passed", lineNum);
408 end
409 end
410 endtask
411
412 task zeroDesired();
413 begin
414     d_ir_load_high = 1'b0;
415     d_ir_load_low = 1'b0;
416     d_gp_primary_addr = 1'b0;
417     d_gp_alu_addr = 1'b0;
418     d_gp_read = 1'b0;
419     d_gp_write = 1'b0;
420     d_pc_set = 1'b0;

```

```

416     d_pc_increment = 1'b0;
417     d_mem_read = 1'b0;
418     d_mem_write = 1'b0;
419     d_latch_alu = 1'b0;
420     d_alu_store_high = 1'b0;
421     d_alu_store_low = 1'b0;
422     d_jr_load_high = 1'b0;
423     d_jr_load_low = 1'b0;
424     d_mar_load_high = 1'b0;
425     d_mar_load_low = 1'b0;
426     d_alu_operation = 4'd0;
427     d_gp_input_select = 3'd0;
428     d_gp_output_select = 3'd0;
429     d_gp_alu_output_select = 3'd0;
430     end
431     endtask
432
433
434 endmodule

```

## B.9 CPU

```

1  /* cpu_test.v Testbench for full CPU
2  * Author: Steven Bell <steven.bell@student.oc.edu>
3  * Date: 6 December 2010
4  * $LastChangedDate: 2010-12-16 17:48:32 -0600 (Thu, 16 Dec 2010) $
5  */
6
7  // Expected clock frequency is ~10 MHz = 100ns
8  // Tick freq is 1 GHz => 100 ticks is one clock cycle; resolution is 10 ps
9  `timescale 1 ns / 10 ps
10
11 module cpu_test();
12     parameter RAM_SIZE = 16; // Size of the RAM in bytes
13     parameter RAM_ADDRESS_BITS = 4; // How many address bits the RAM needs
14     parameter ROM_SIZE = 1024; // Size of the ROM in bytes
15     parameter ROM_ADDRESS_BITS = 10; // How many address bits the ROM needs
16     parameter MAX_LINE_LENGTH = 200; // Maximum length of line in input file
17
18     // Signals for DUT
19     reg reset;
20     reg[7:0] data_bus_driver; // Value from the memory to drive the bus
21     reg memory_drives_bus; // True if the memory is driving the data bus
22     wire[7:0] data_bus; // Bidirectional data bus
23     wire[15:0] address_bus; // Address bus coming out of the chip
24     wire ram_enable;
25     wire rom_enable;
26     wire write;
27
28     // Variables for file processing and simulation
29     integer clock_cycles; // Number of clock cycles we've stepped through
30     reg[7:0] rom[0:ROM_SIZE-1]; // Simulated EEPROM
31     reg[7:0] ram[0:RAM_SIZE-1]; // Simulated RAM
32     integer i; // Index for ROM

```

```

33 integer file; // File handle
34 integer c; // Test character for seeing if the a line begins with a comment
35 integer r; // Unused placeholder for return values from file parsing
36 reg[8*MAX_LINE_LENGTH:0] line; // Unused placeholder for string comments
    that get read
37
38 // Clock
39 reg clock = 0;
40 always #50 clock = ~clock;
41
42 // DUT
43 cpu cpuDUT(.clock(clock),
44           .reset(reset),
45           .external_data_bus(data_bus),
46           .address_bus(address_bus),
47           .ram_enable(ram_enable),
48           .rom_enable(rom_enable),
49           .write(write));
50
51 // Waveform log file
52 initial begin
53     $dumpfile("cpu.vcd");
54     $dumpvars;
55 end
56
57 // Bidirectional driver code for data bus
58 assign data_bus = memory_drives_bus ? data_bus_driver : 16'hzz;
59
60 // Beginning of test
61 initial begin
62     // Initialize ROM to all zeros
63     for(i = 0; i < ROM_SIZE; i = i + 1) begin
64         rom[i] = 8'd0;
65     end
66
67     $display("Loading_instructions_into_simulated_ROM_from_\`cpu_code\`");
68     // Start reading the file
69     file = $fopen("cpu_code", "r");
70
71     // Loop until we get to the end of the file and initialize ROM with the
        contents
72     i = 0; // Start back at the beginning of ROM
73     while(! $feof(file)) begin
74         // Check if the first character is a '#', signifying a comment
75         c = $fgetc(file);
76         if(c == "#") begin
77             // If we've got a comment, skip to the next line
78             r = $fgets(line, file);
79         end
80         else begin
81             // Otherwise, assume that it's a hex value and try to read it
82             r = $ungetc(c, file);
83             r = $scanf(file, "%h\n", rom[i]);
84             i = i + 1;

```

```

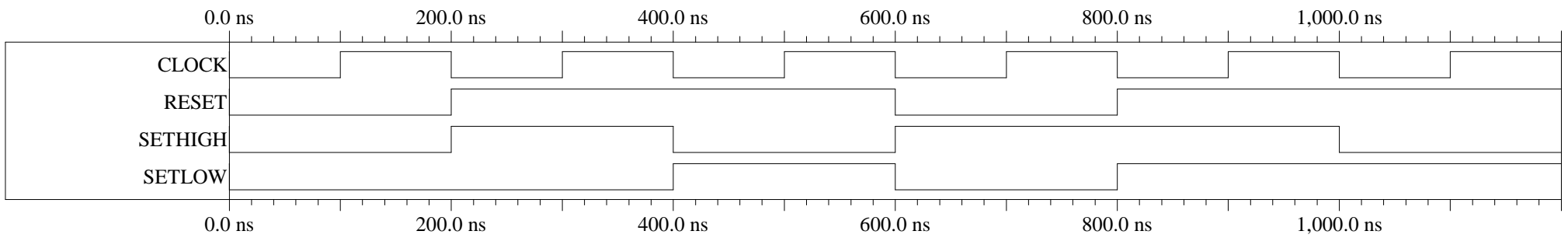
85     end
86 end
87
88 // Test: print out the contents of ROM
89 for(i = 0; i < 20; i = i + 1) begin
90     $write("%h_", rom[i]);
91     if(i % 8 == 7) begin
92         $write("\n");
93     end
94 end
95
96 // Perform a reset
97 @(negedge clock)
98     reset = 1'b0;
99 @(negedge clock)
100    reset = 1'b1;
101
102 // And off we go!
103 for(clock_cycles = 0; clock_cycles < 500; clock_cycles = clock_cycles + 1)
104     begin
105         @(posedge clock)
106         memory_drives_bus = 1'b0; // Relinquish control of the data bus until we
107             're told to keep using it
108         #10 // Wait for the memory address to be set before we try to give back
109             data!
110         if(rom_enable == 1'b0) begin // ROM output-enable is active low
111             data_bus_driver = rom[address_bus];
112         end
113         else if(ram_enable == 1'b0) begin // RAM output-enable is also active
114             low
115             if(write == 1'b1) begin // CPU is writing to RAM
116                 ram[address_bus[RAM_ADDRESS_BITS-1:0]] = data_bus;
117             end
118             else begin // CPU is reading from RAM
119                 data_bus_driver = ram[address_bus[RAM_ADDRESS_BITS-1:0]];
120             end
121         end // if(rom/ram enable)
122         memory_drives_bus = !write;
123
124         // $display("%h %h", address_bus, data_bus);
125     end // for(clock_cycles)
126 $finish;
127 end // initial
128 endmodule

```

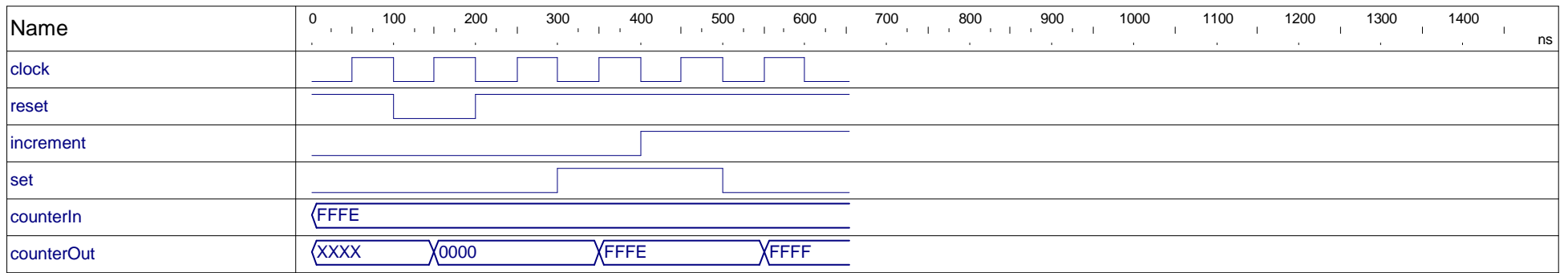
## C Output waveforms

Waveforms were created by dumping the signals to a VCD file directly from Verilog, converting them to Aldec's .abf file format using the Aldec command-line utilities `vcd2asdb` and `awfconv`, and viewing them in ActiveHDL's waveform viewer.

1. Generic 16-bit register .....	77
2. Program counter .....	78
3. ALU .....	79
4. ALU latch .....	81
5. General-purpose register block .....	82
6. Datapath .....	83
7. Control module state machine .....	86
8. Control signals translator .....	87
9. CPU final test, part A .....	89
10. CPU final test, part B .....	102



### Program Counter



## ALU

Name	0	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	ns		
clock																		
reset																		
primaryOperand	XX				00	01	00	01	FF	7D	F1	FA	0A				02	
secondaryOperand	XX				02	00	FF	02	0A				06	0A		0F	FA	
operation	X				C				1				2					
result	XXXX				0002	0000	00FF	0003	0101	0087	00FB	0100	0004	0000	01FB	0108		
flags	XXX				000	100	001	000	010	001	110				000	100	011	010

Name	1490	1590	1690	1790	1890	1990	2090	2190	2290	2390	2490	2590	2690	2790	2890	ns		
clock																		
reset																		
primaryOperand	80	0A	64	FF	F6	AA				2A	AA	00	AA	AB	00	2A		
secondaryOperand	01	06	3C	FF	06	66	55	E6	66	00				FF				
operation	3				4				5				6				7	
result	007F	003C	1770	FE01	05C4	0022	0000	00A2	006E	00EE	0000	0055	0000		0054			
flags	000				001	000	100				001	000	001	100	000	010	100	000

Name	2980	3080	3180	3280	3380	3480	3580	3680	3780	3880	3980	4080	4180	4280	4380	ns					
clock																					
reset																					
primaryOperand	AA	00	C0	2A	2B	00	AA	2A	AA	6A	00	01	80	32	CE						
secondaryOperand																					
operation	9								A								B				
result	0000				0080	0015	0000				0055	0054	00D4				0000	FFFF	FF80	FFCE	FF32
flags	010	100	011	000	010	100	000				010	001	100	001				000			

# ALU

Name	4470	4570	4670	4770	4870	4970	5070	5170	5270	5370	5470	5570	5670	5770	5870	ns
clock																
reset																
primaryOperand																
secondaryOperand																
operation																
result																
flags																

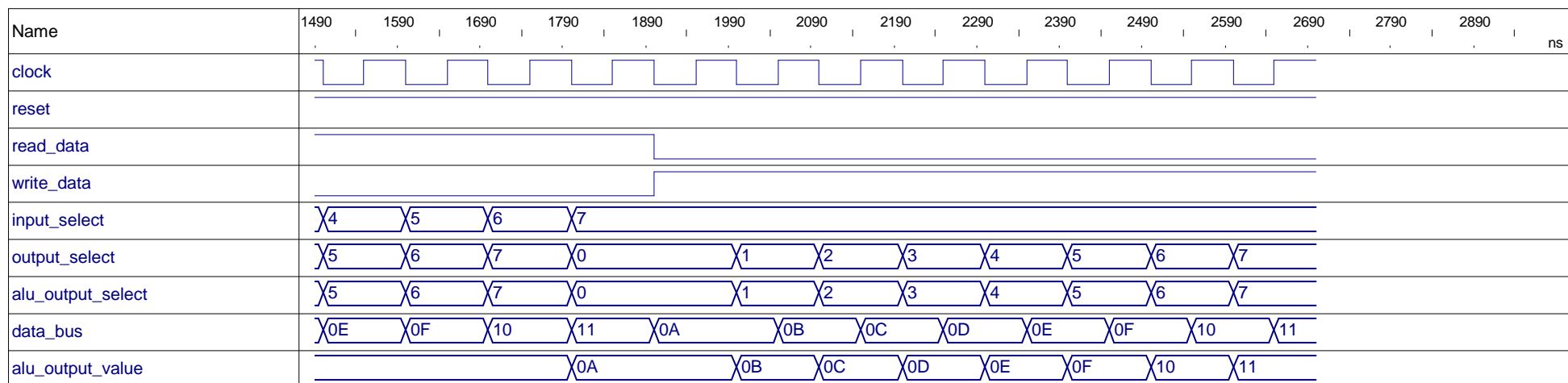
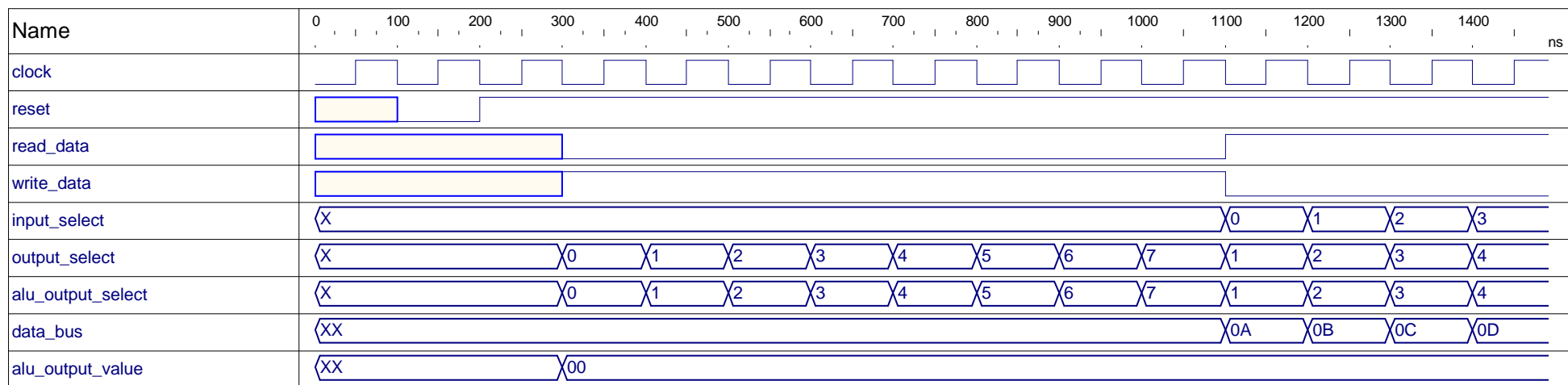
### ALU Latch

Name	0	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	ns
clock																
reset																
grab																
alu_result																
store_high																
store_low																
databus																

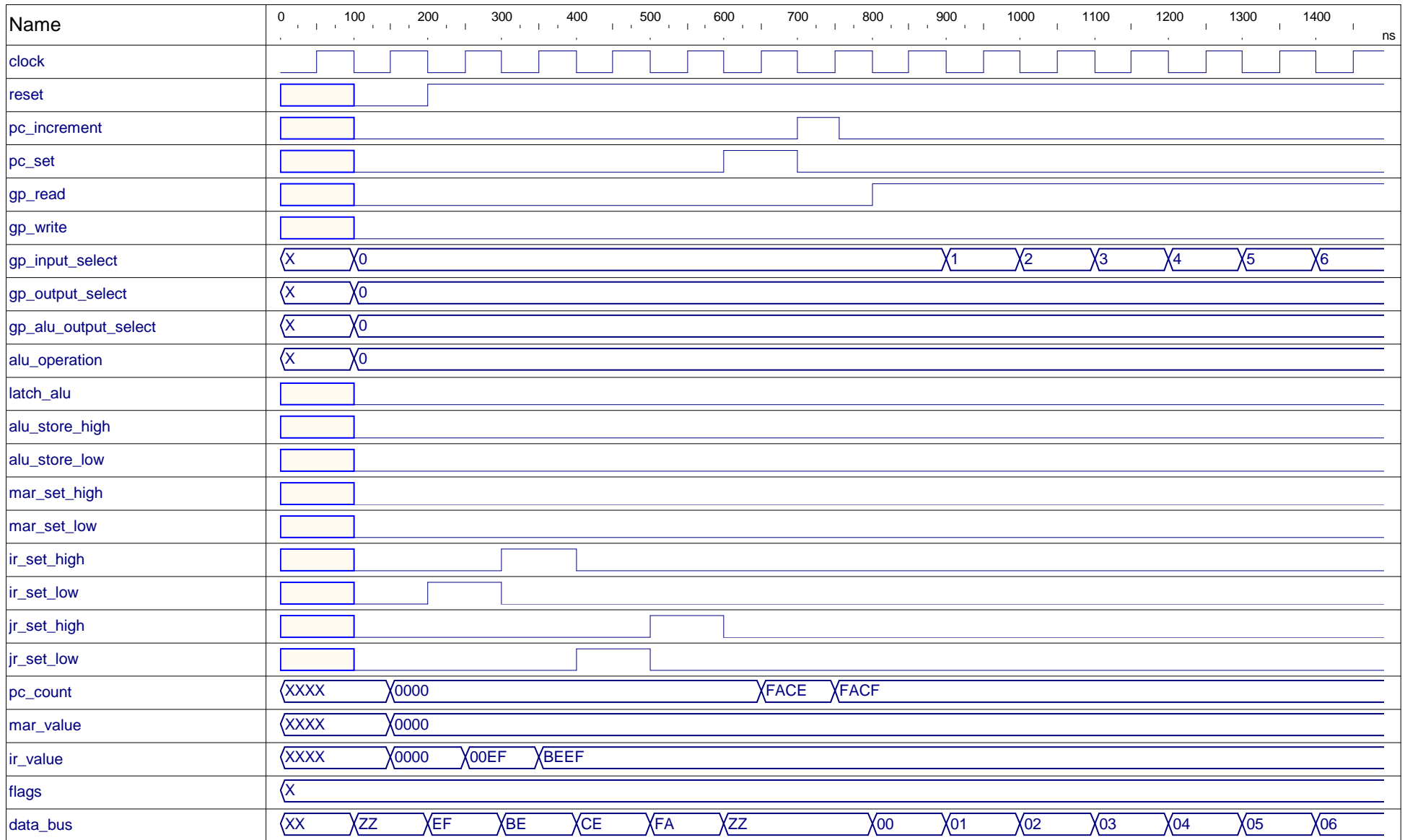
Name	1490	1590	1690	1790	1890	1990	2090	2190	2290	2390	2490	2590	2690	2790	2890	ns
clock																
reset																
grab																
alu_result																
store_high																
store_low																
databus																

Name	2980	3080	3180	3280	3380	3480	3580	3680	3780	3880	3980	4080	4180	4280	4380	ns
clock																
reset																
grab																
alu_result																
store_high																
store_low																
databus																

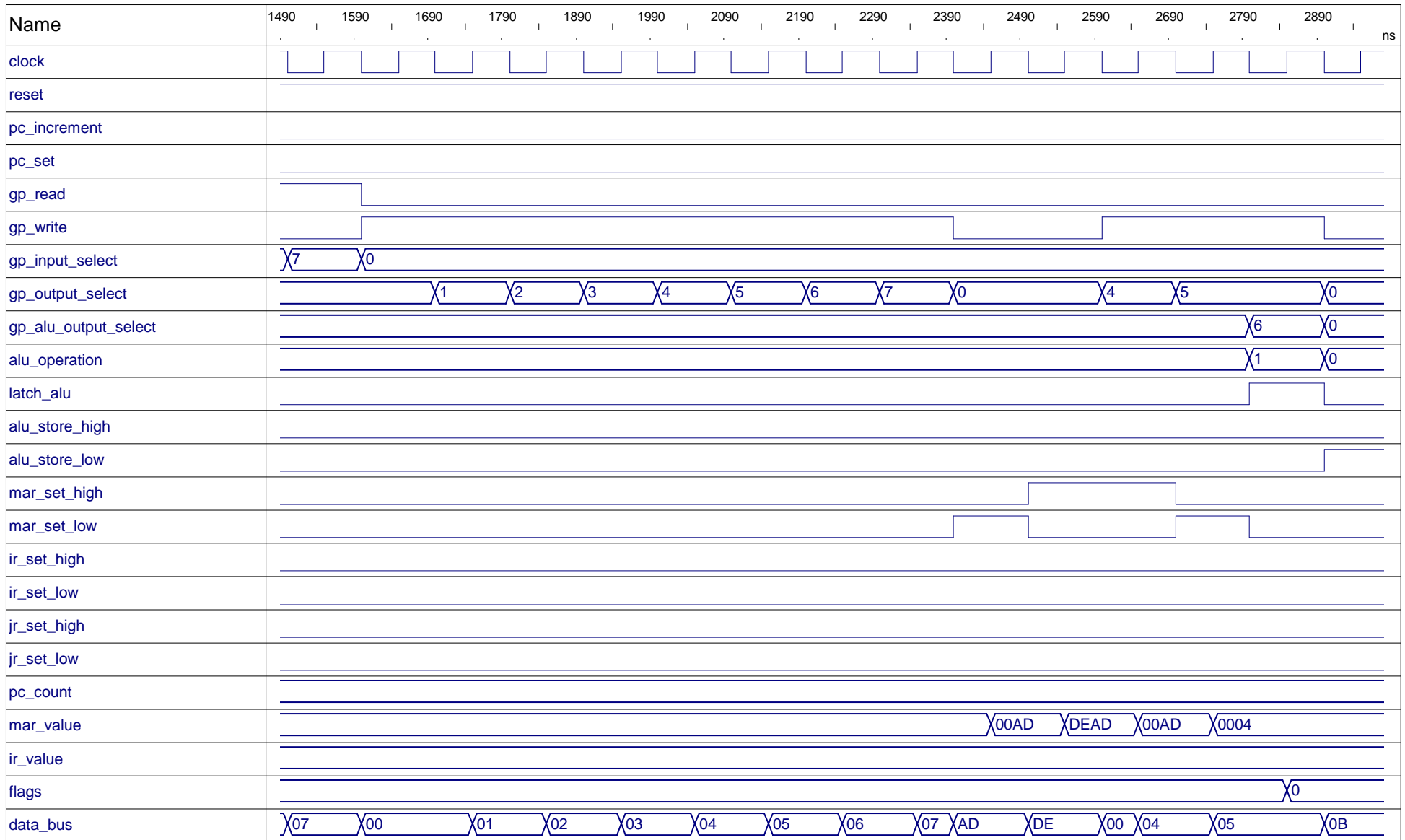
### General purpose registers



## Datapath



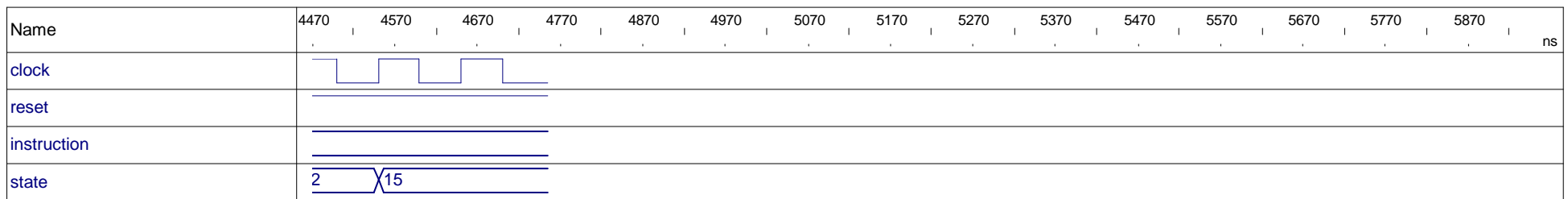
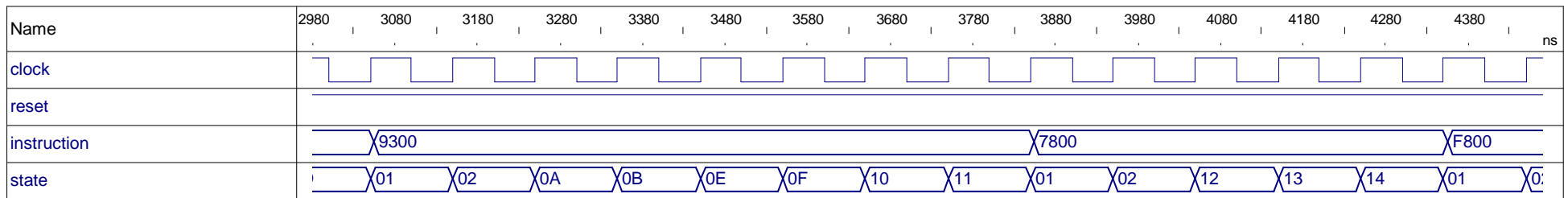
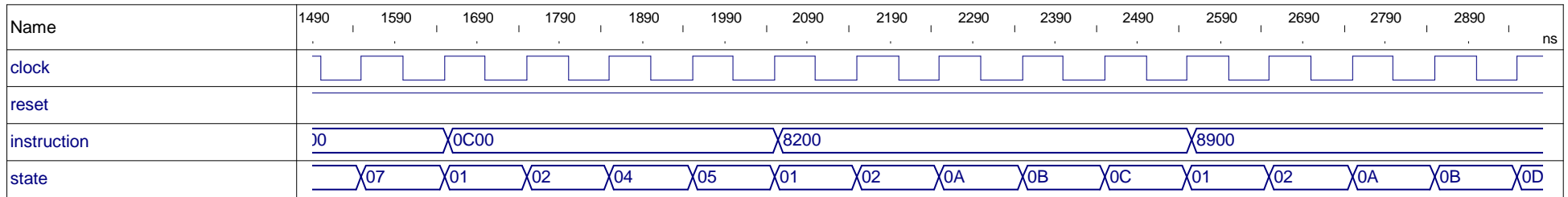
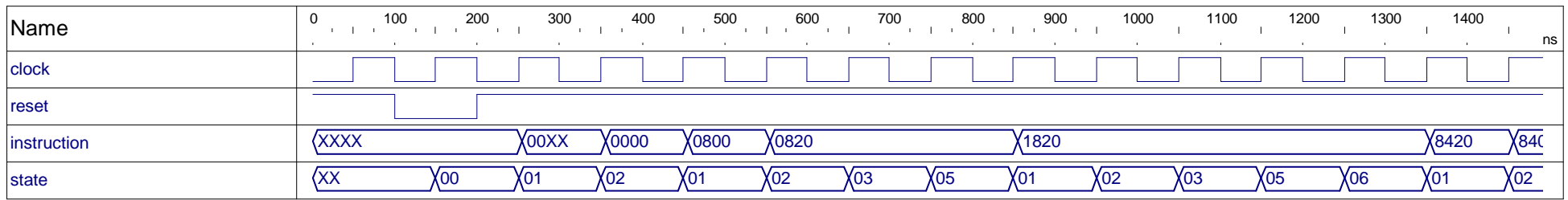
## Datapath



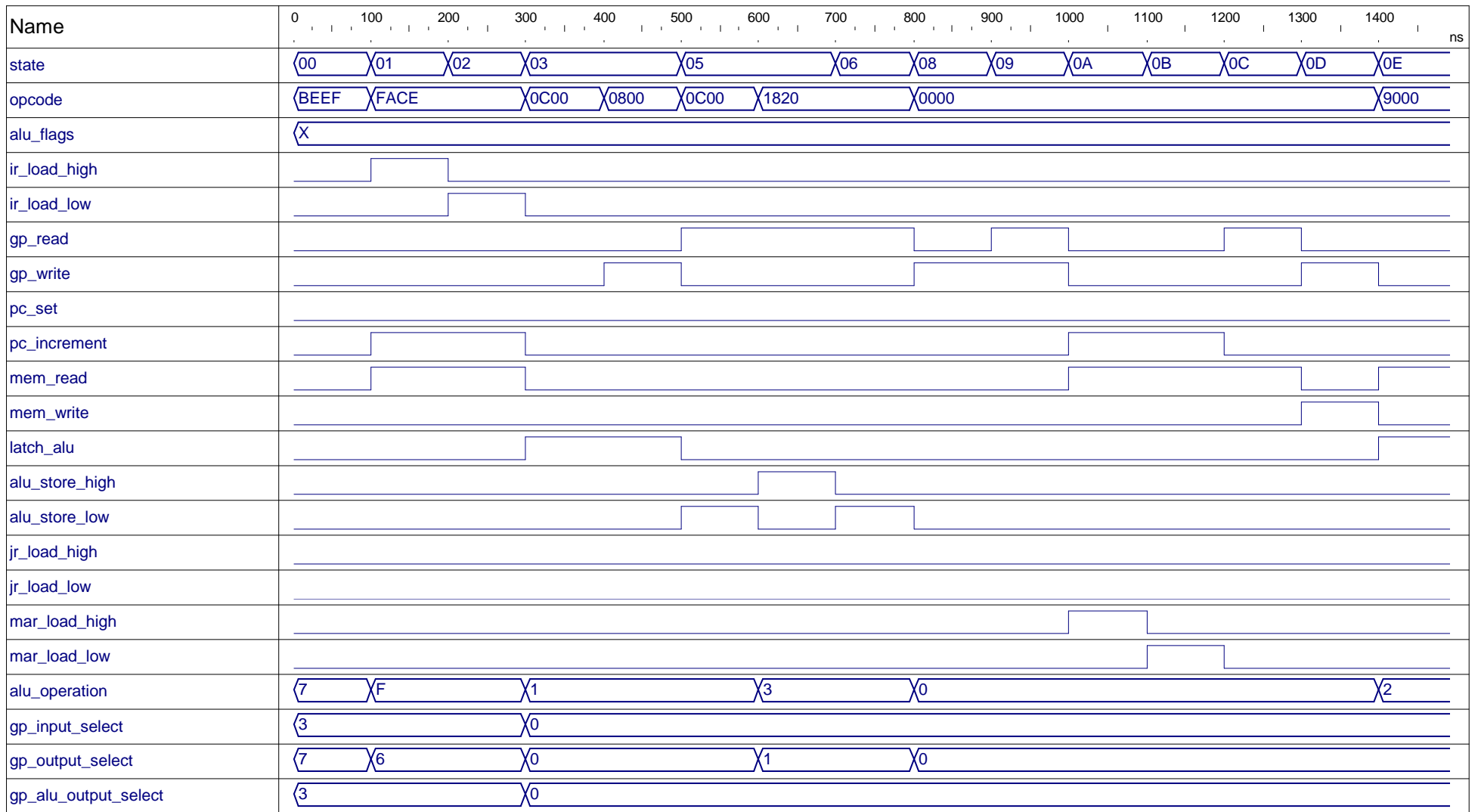
### Datapath

Name	2980	3080	3180	3280	3380	3480	3580	3680	3780	3880	3980	4080	4180	4280	4380	ns
clock																
reset																
pc_increment																
pc_set																
gp_read																
gp_write																
gp_input_select																
gp_output_select																
gp_alu_output_select																
alu_operation																
latch_alu																
alu_store_high																
alu_store_low																
mar_set_high																
mar_set_low																
ir_set_high																
ir_set_low																
jr_set_high																
jr_set_low																
pc_count																
mar_value																
ir_value																
flags																
data_bus																

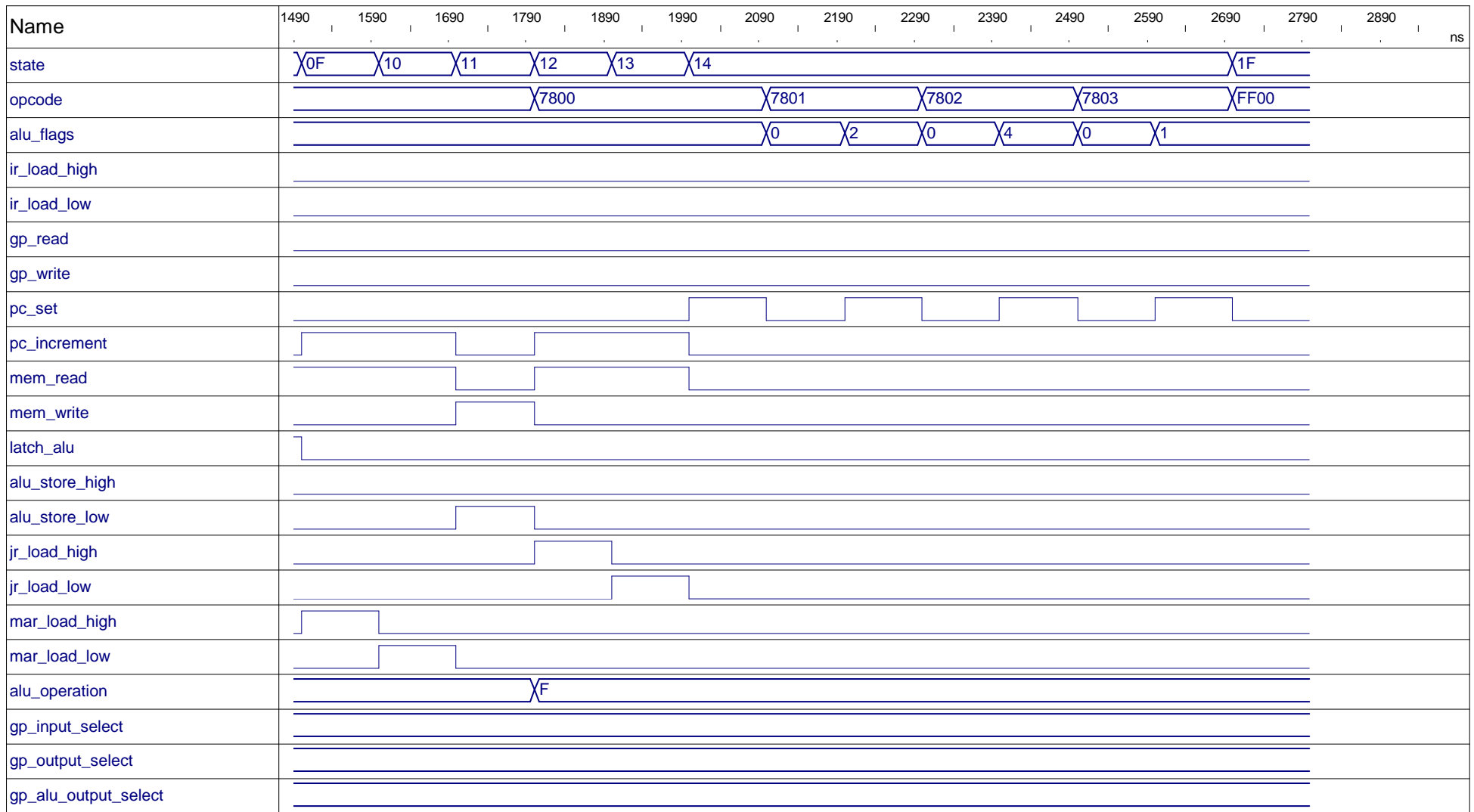
### Control state machine



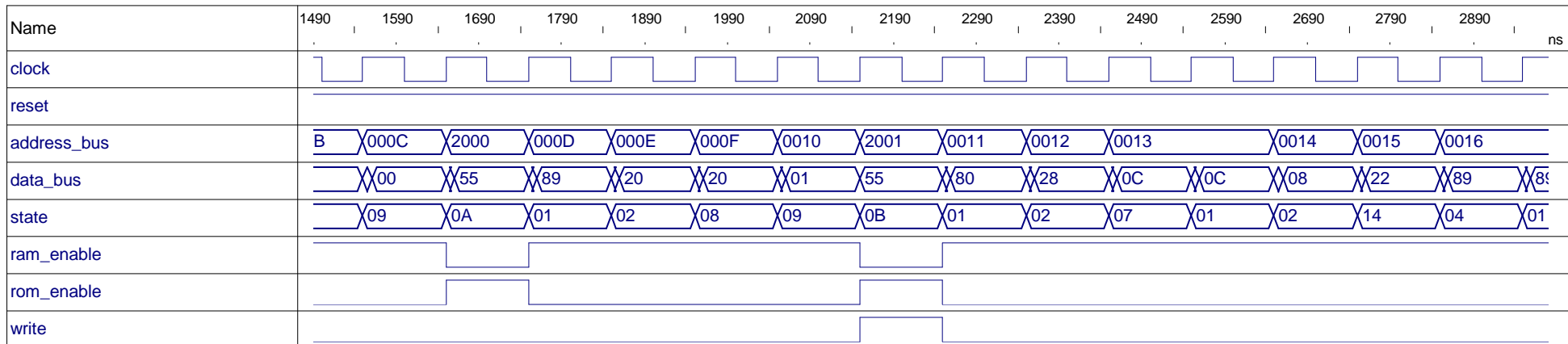
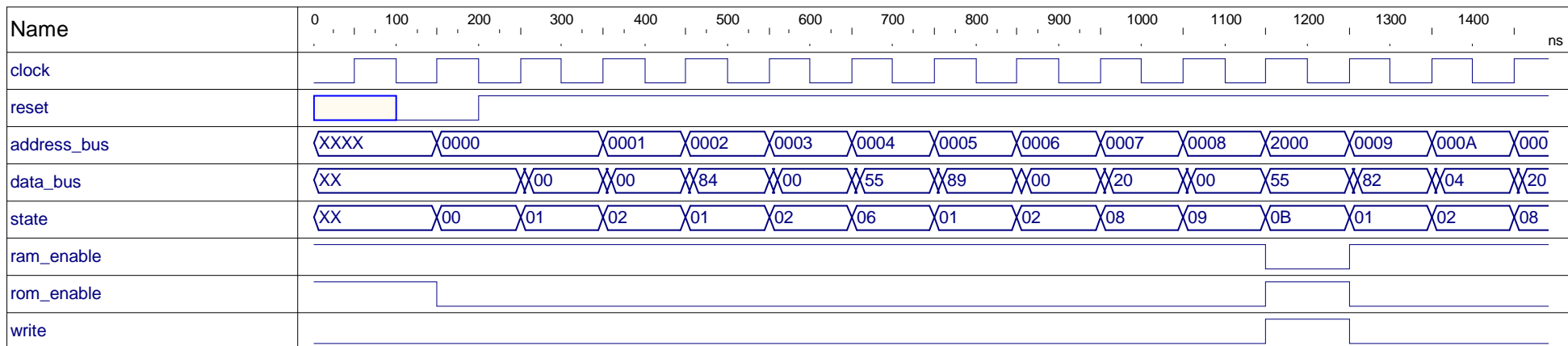
### Control signal translator



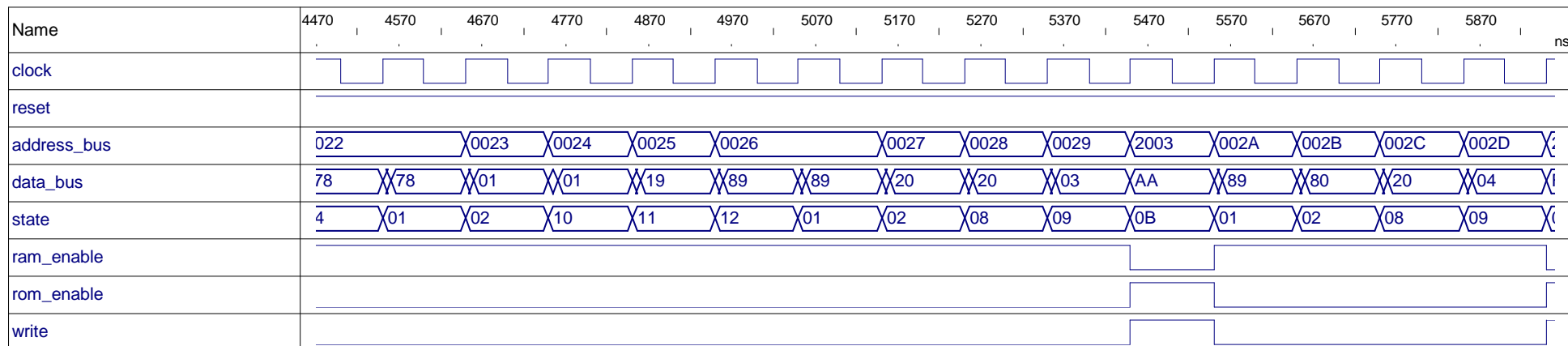
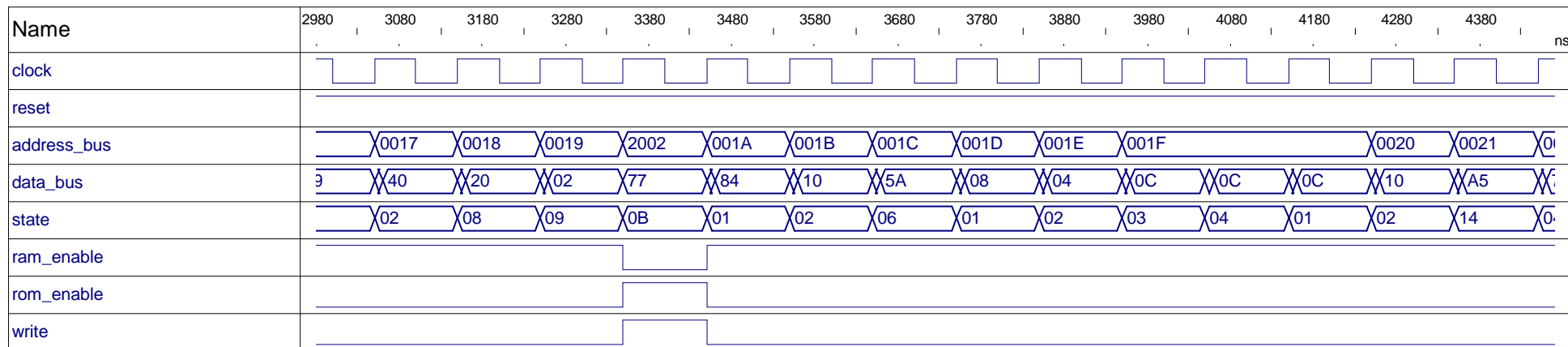
### Control signal translator



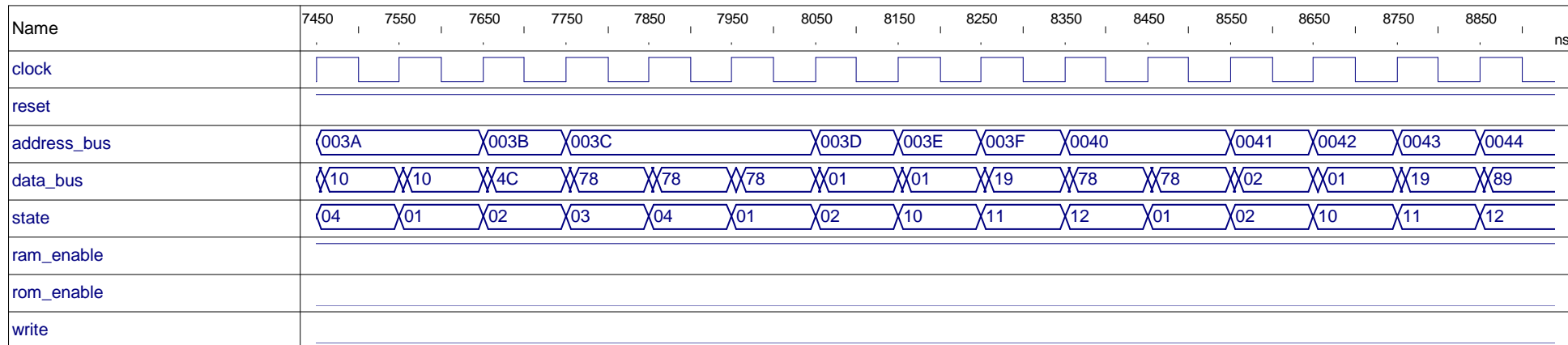
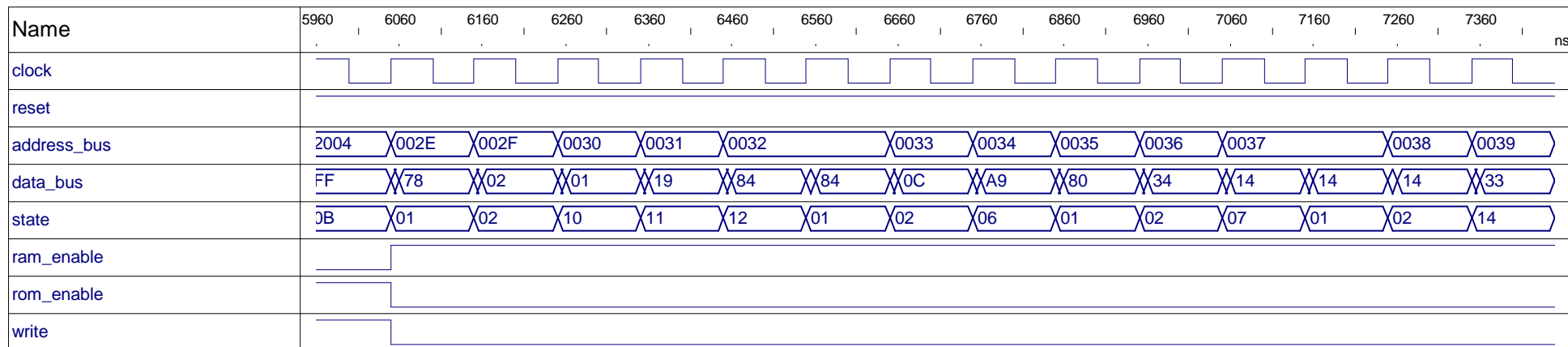
CPU test program, part A



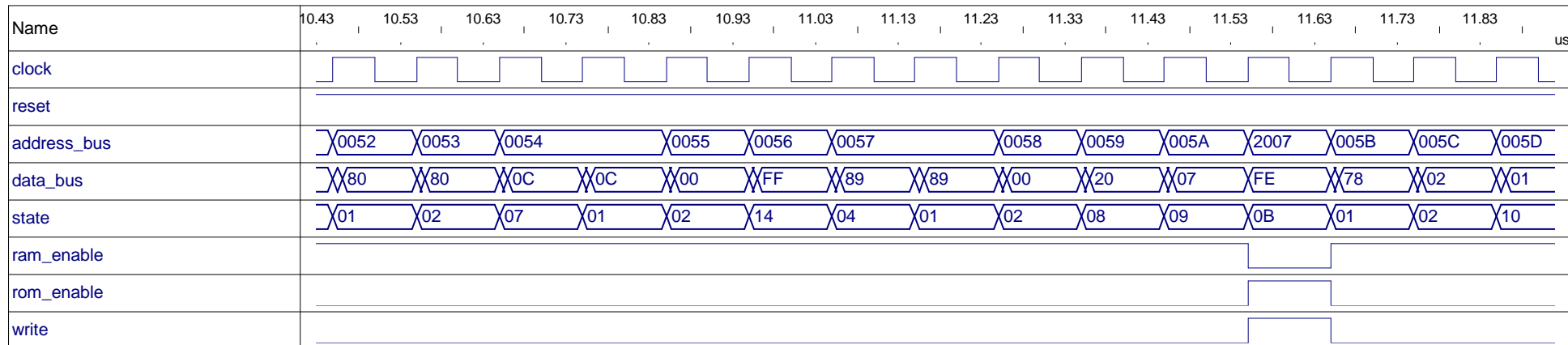
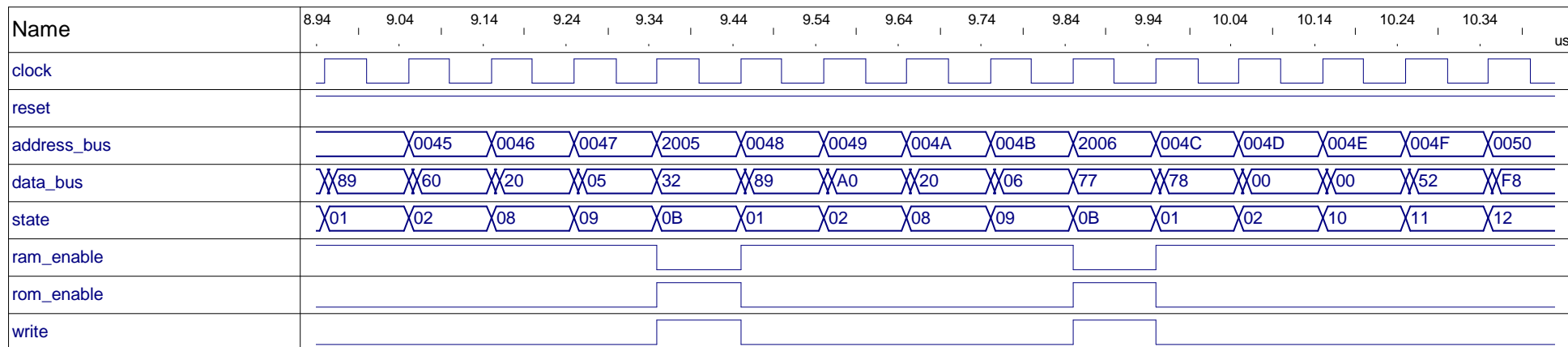
CPU test program, part A



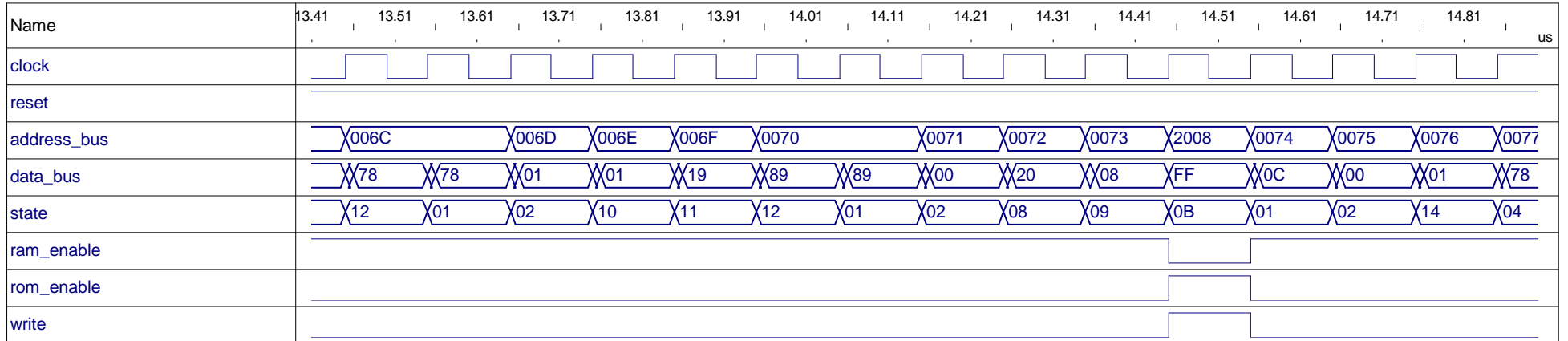
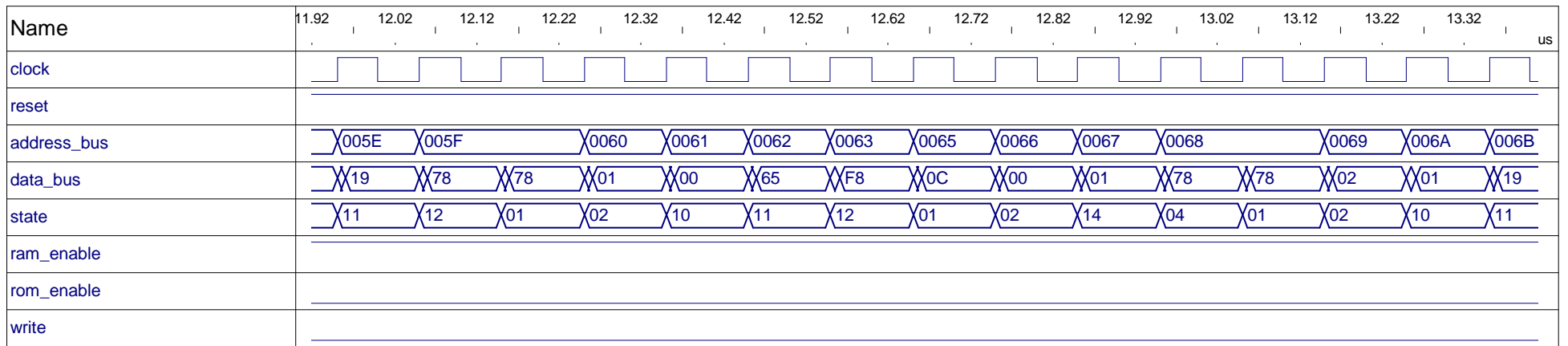
CPU test program, part A



CPU test program, part A

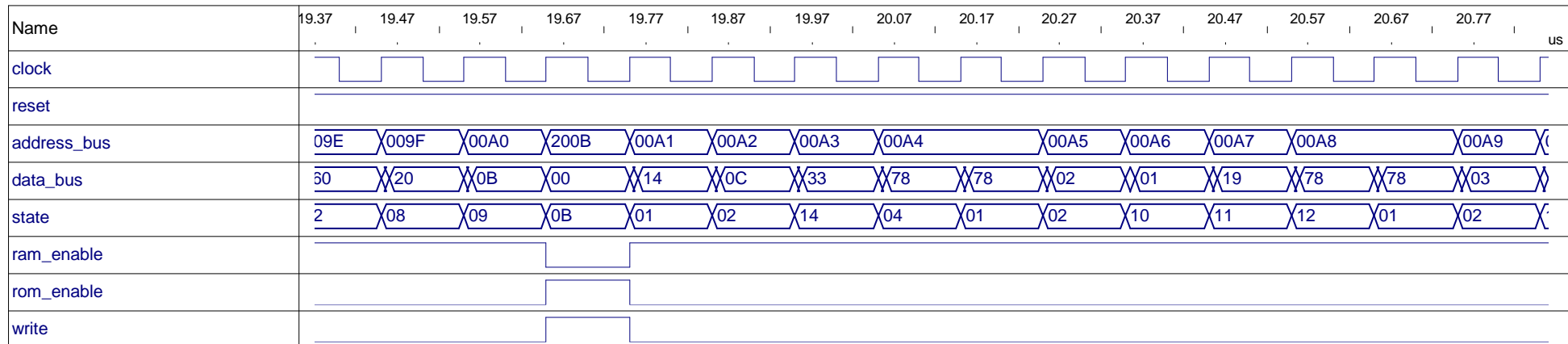
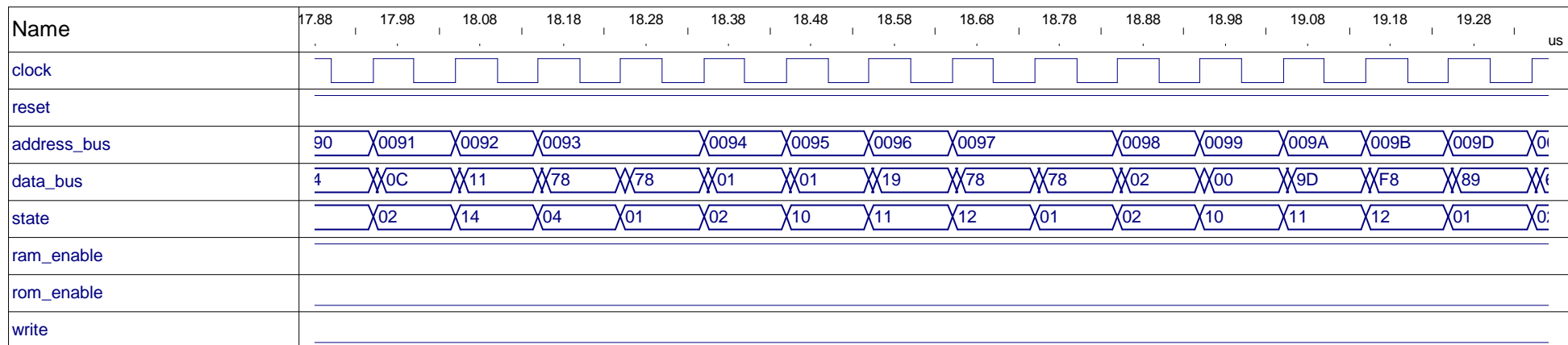


CPU test program, part A

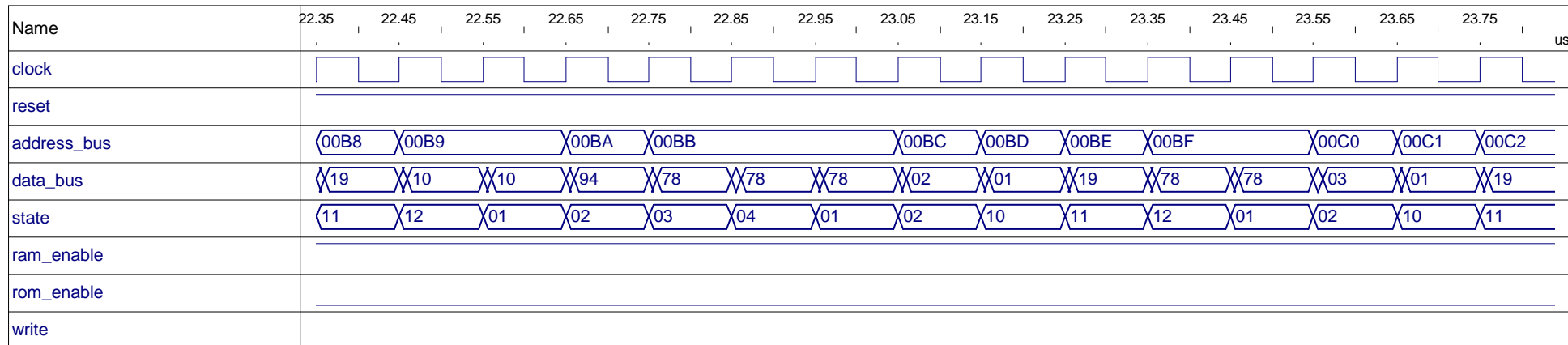
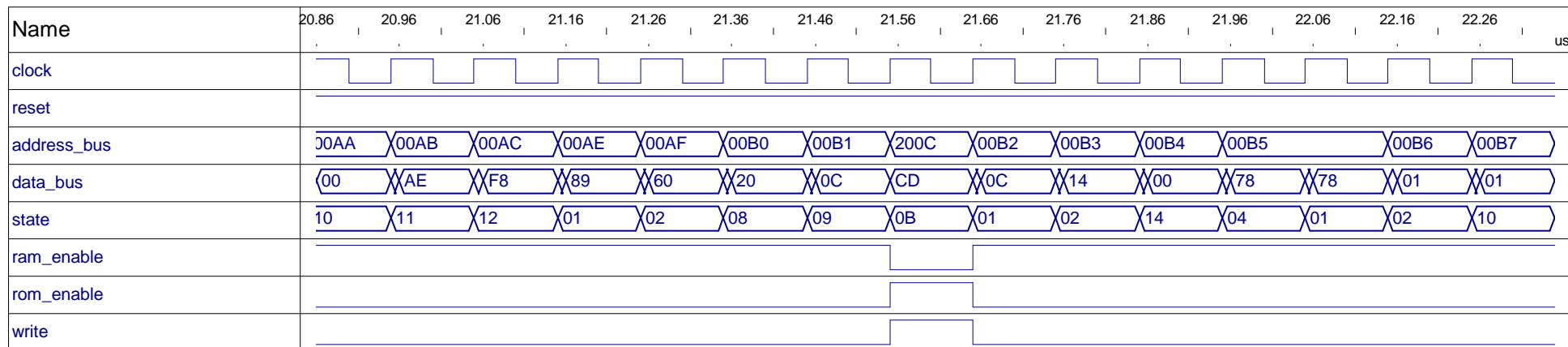




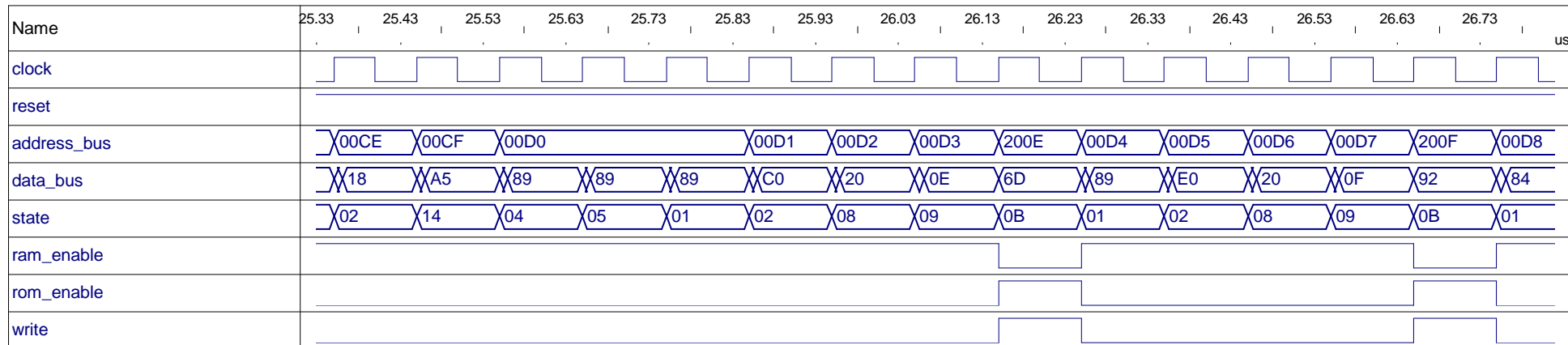
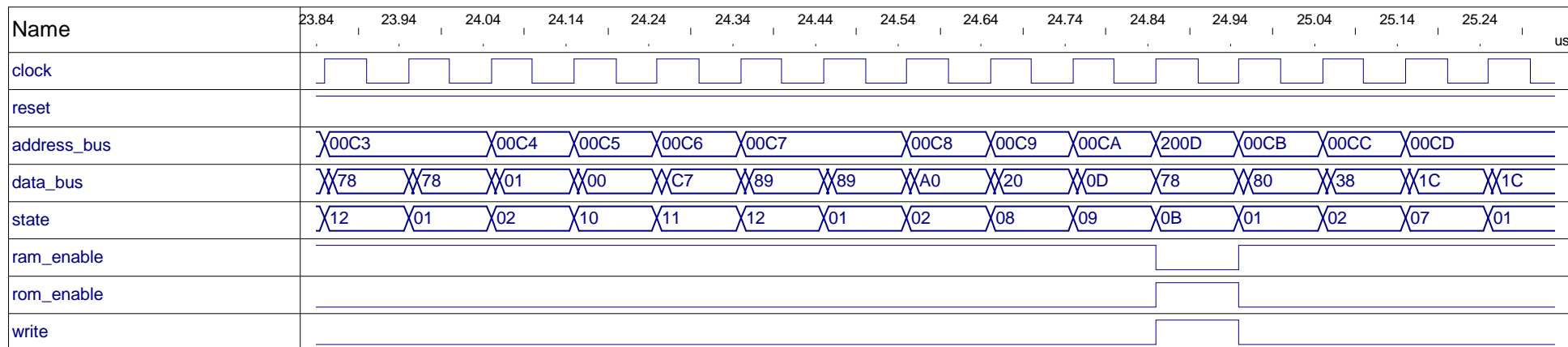
CPU test program, part A



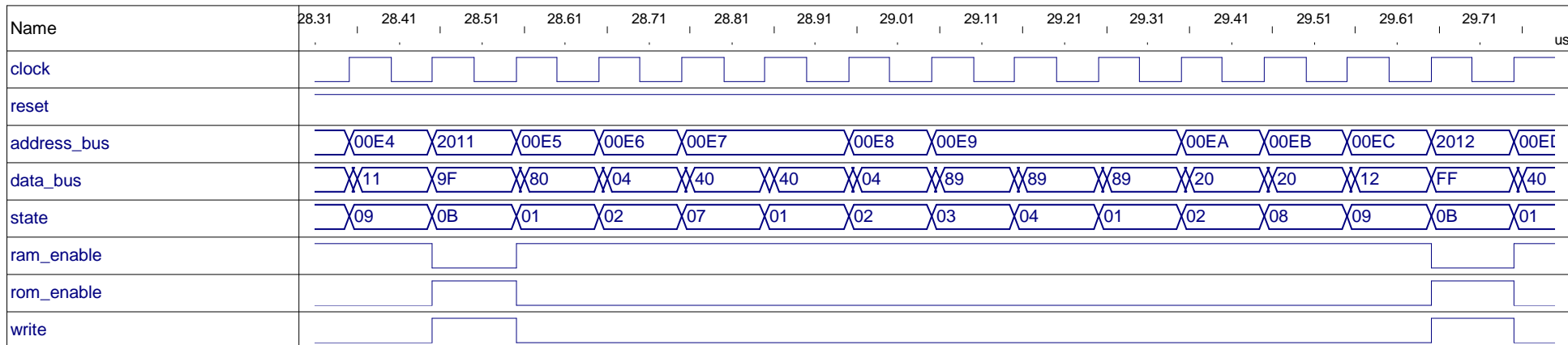
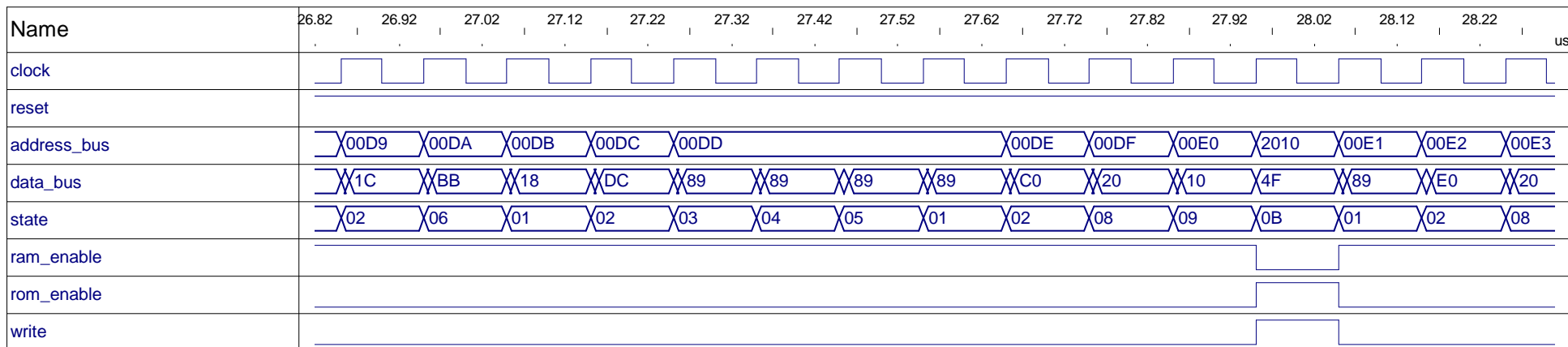
CPU test program, part A



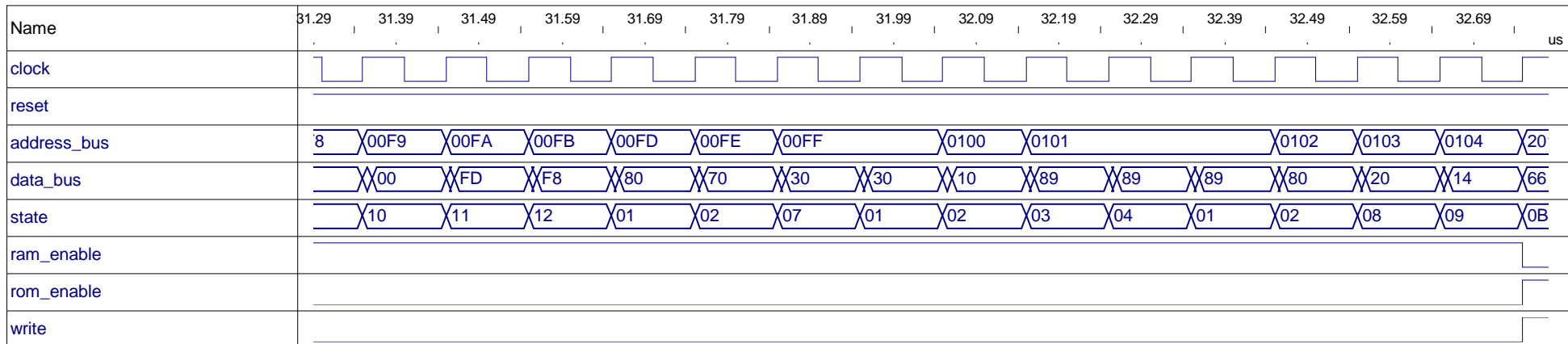
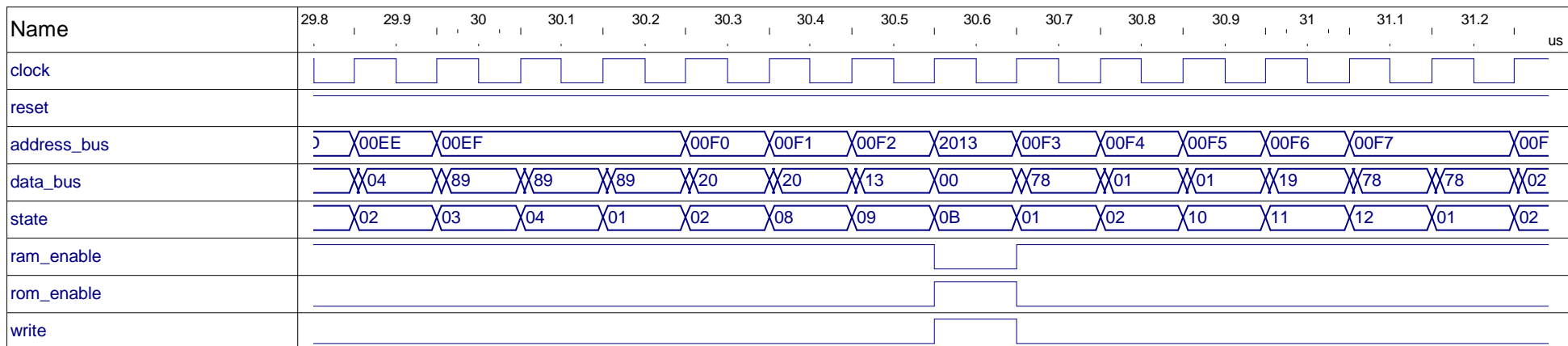
CPU test program, part A



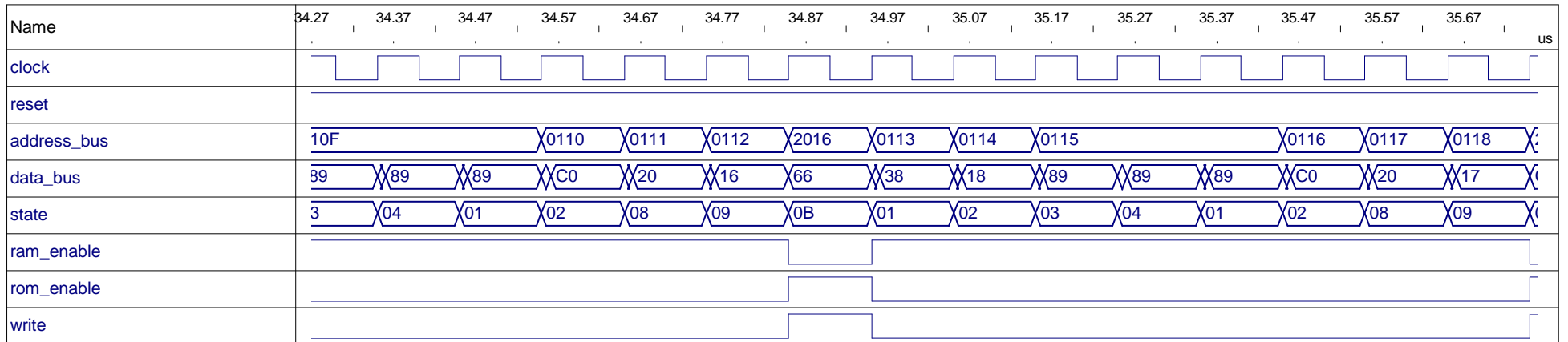
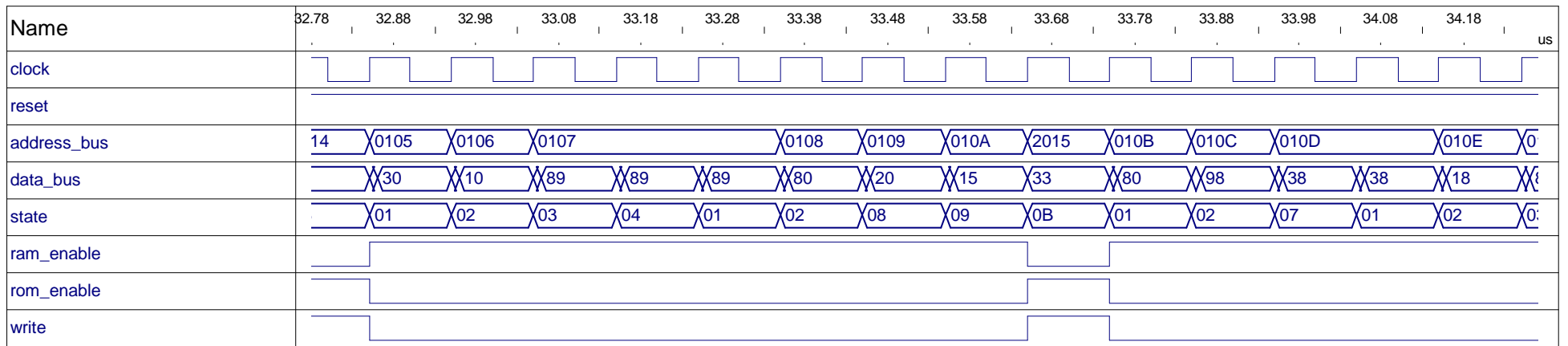
CPU test program, part A



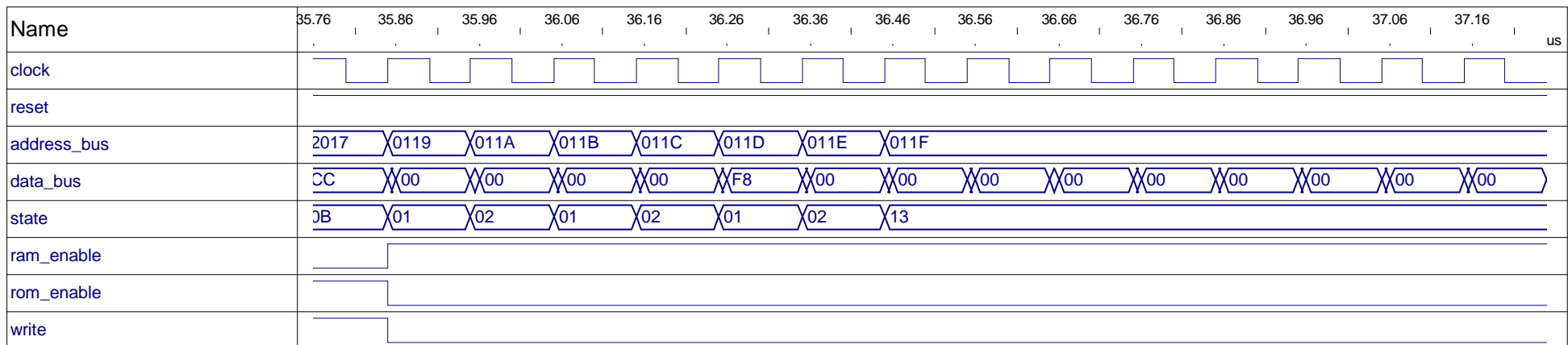
CPU test program, part A



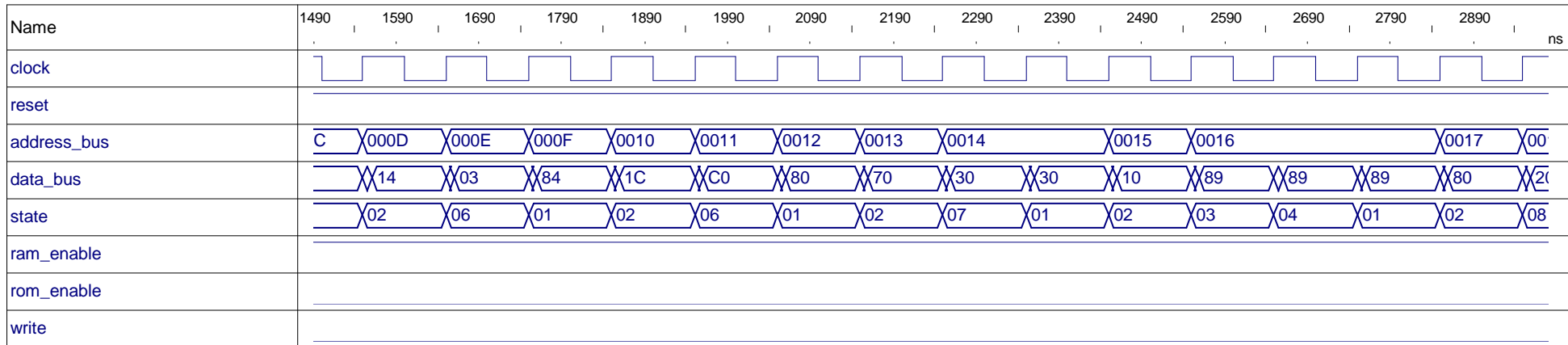
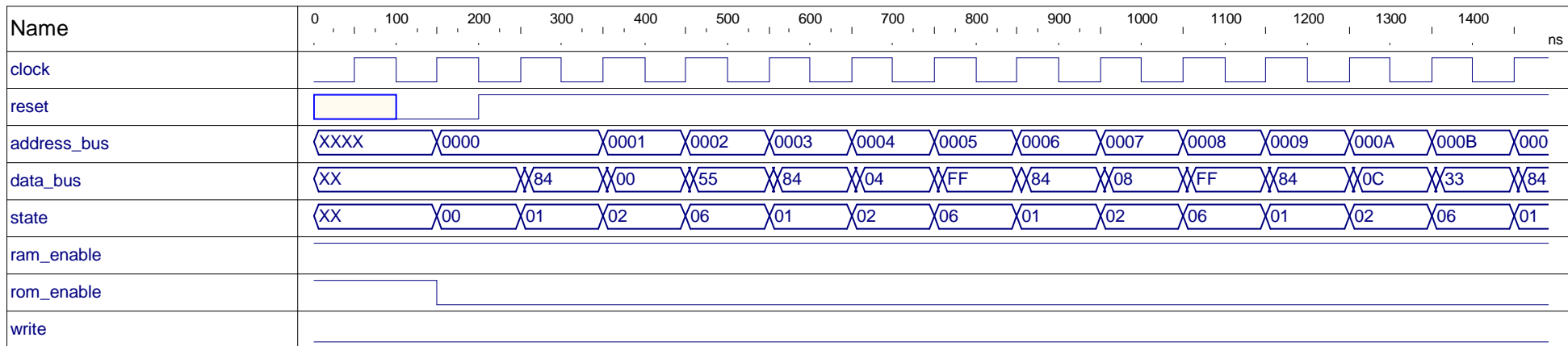
CPU test program, part A



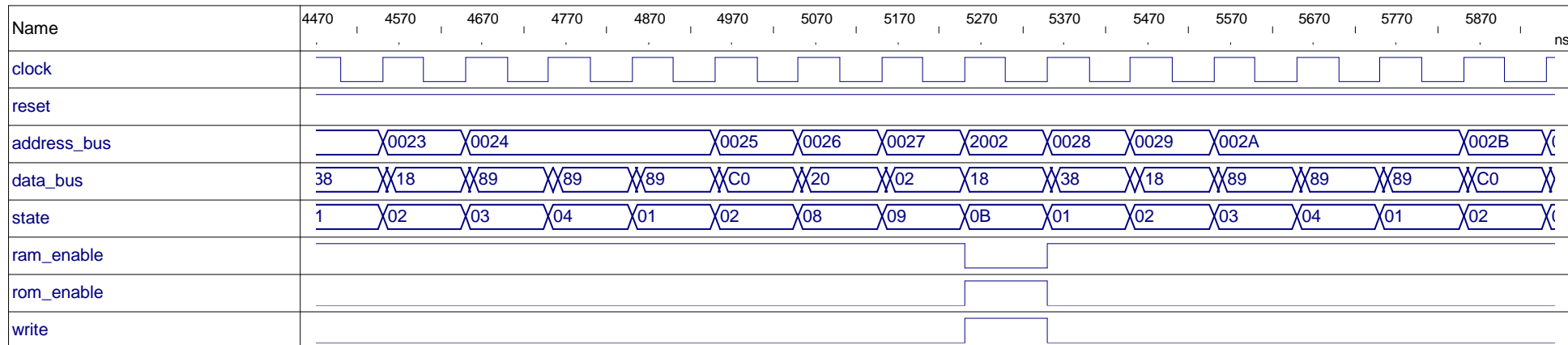
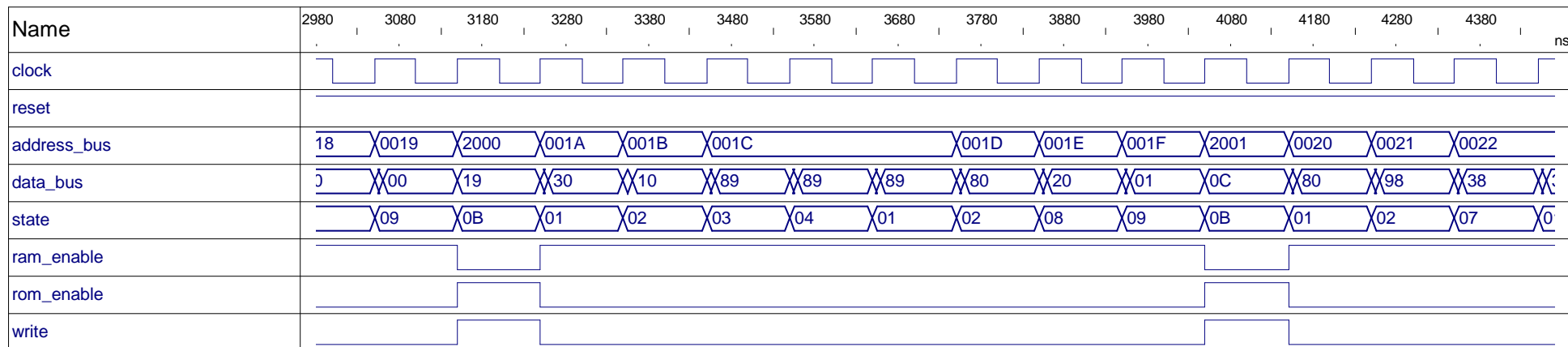
CPU test program, part A



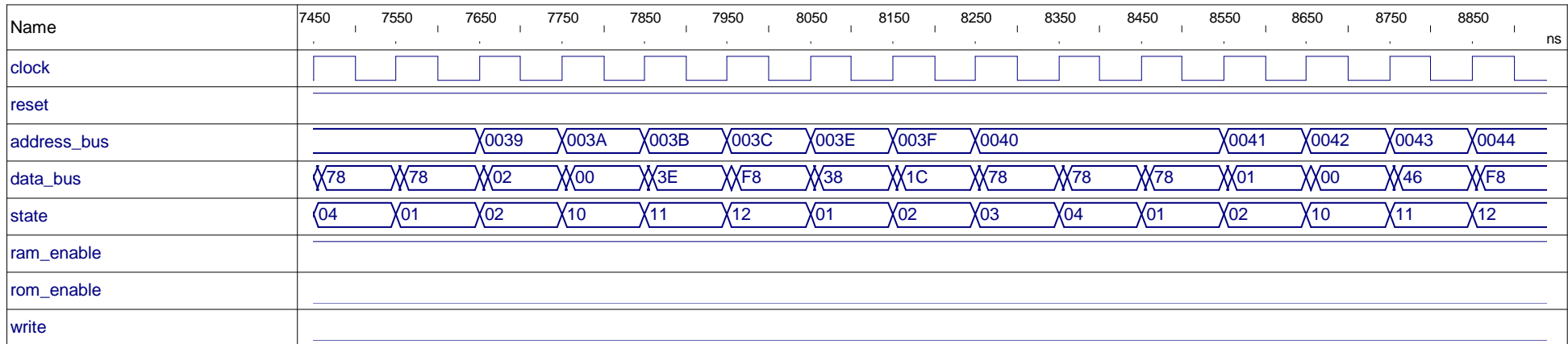
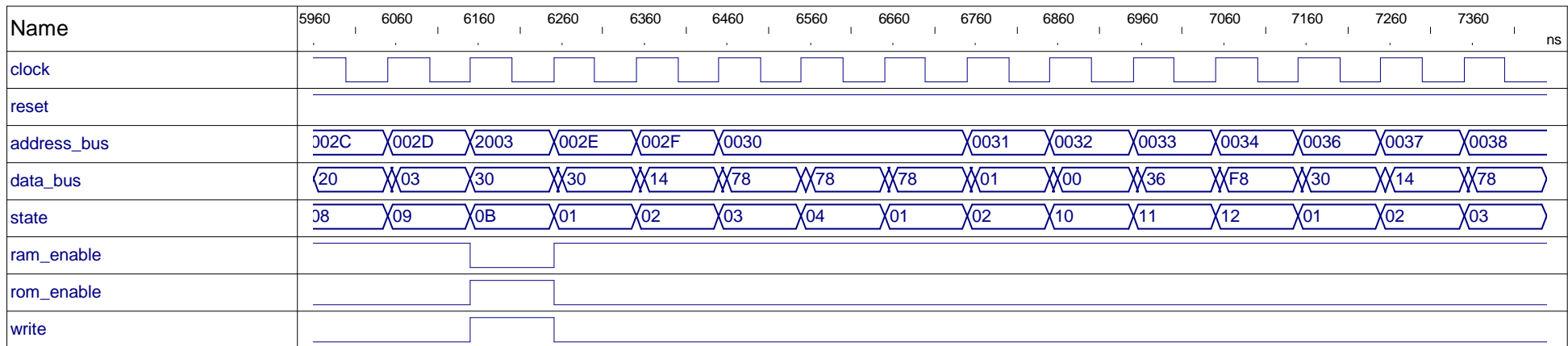
CPU test program B



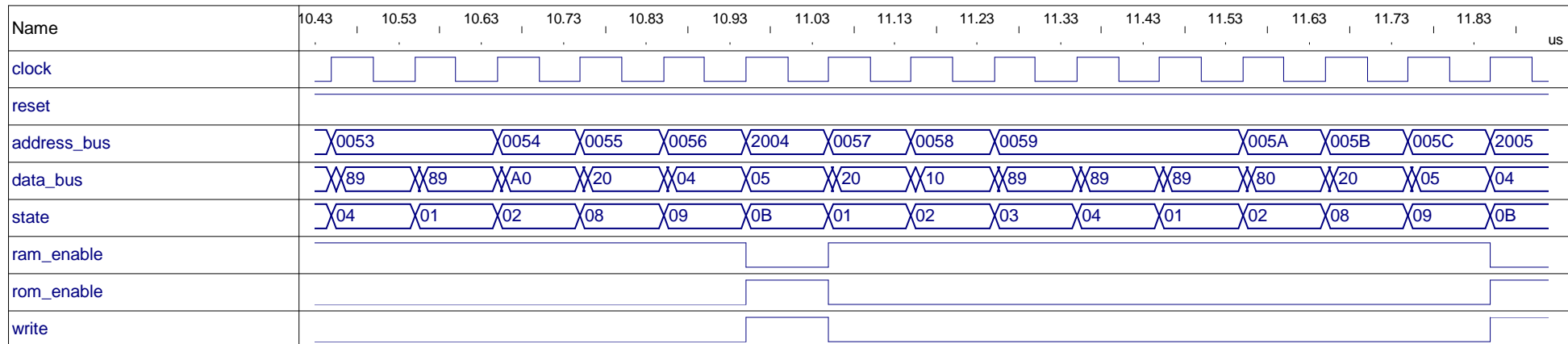
CPU test program B



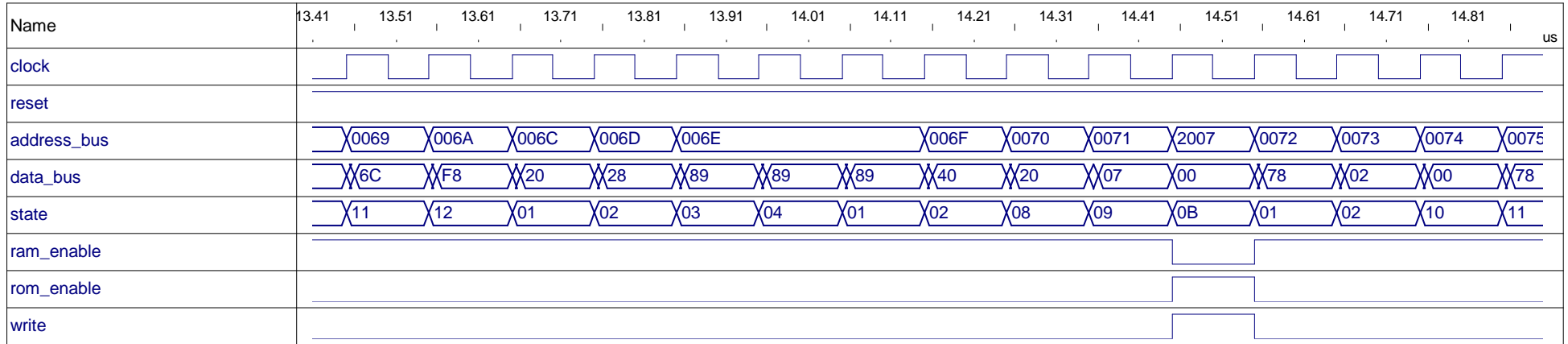
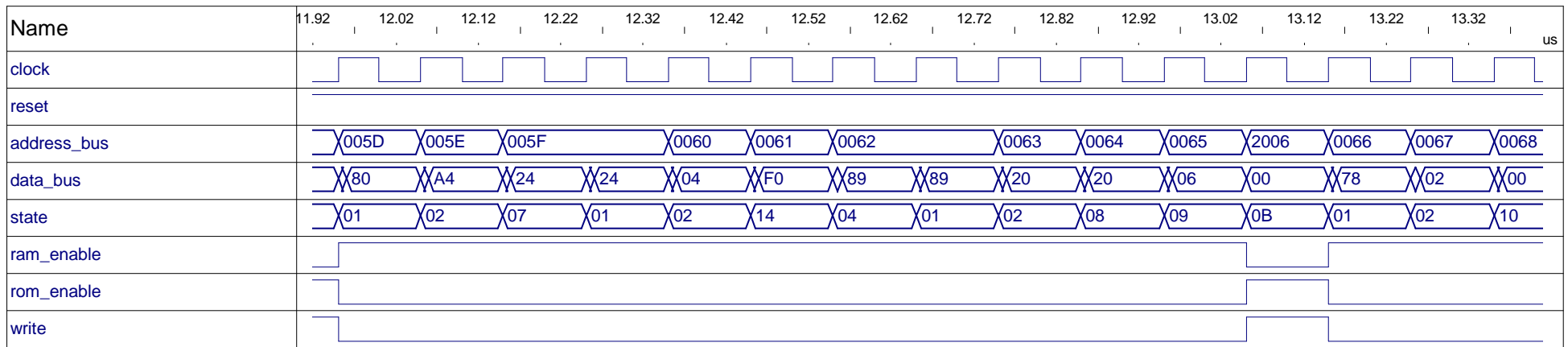
CPU test program B



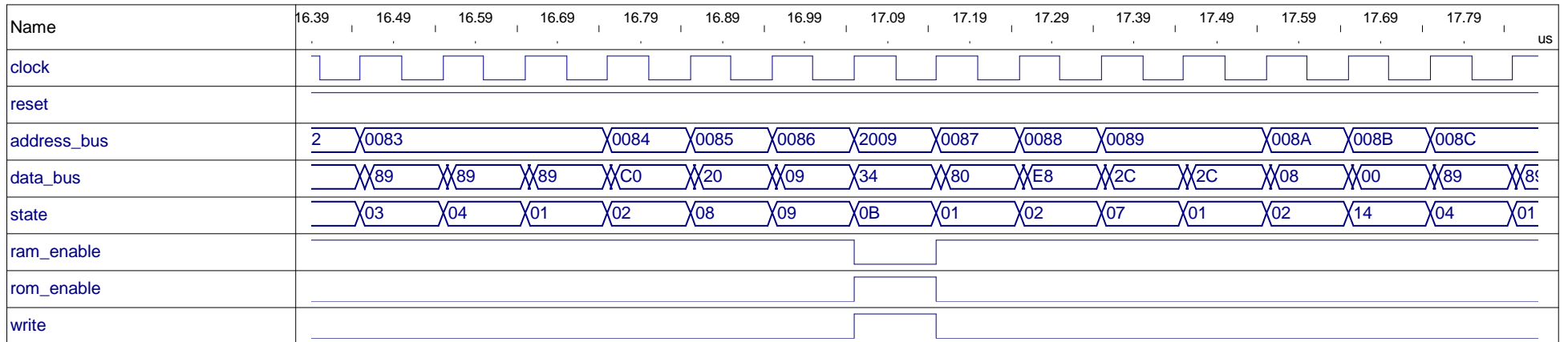
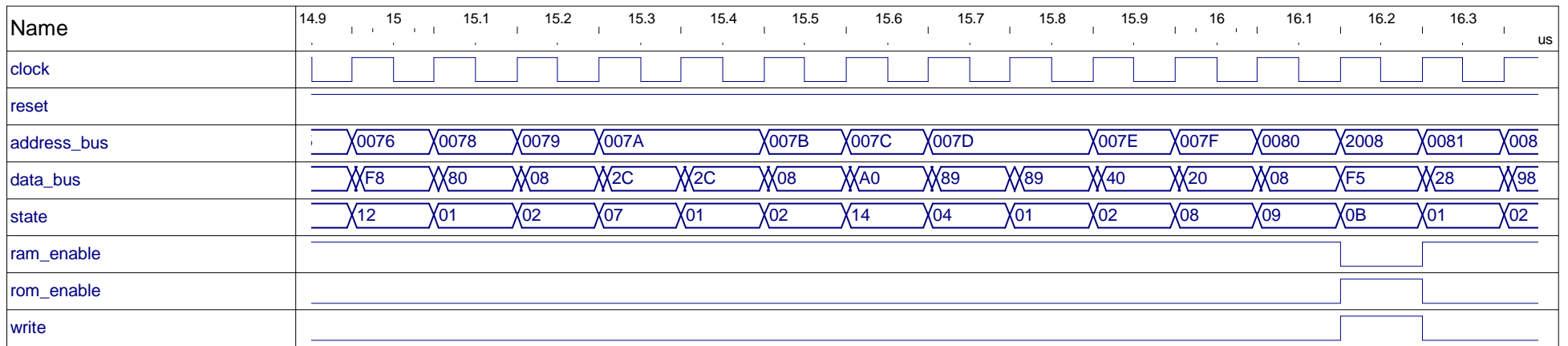
CPU test program B



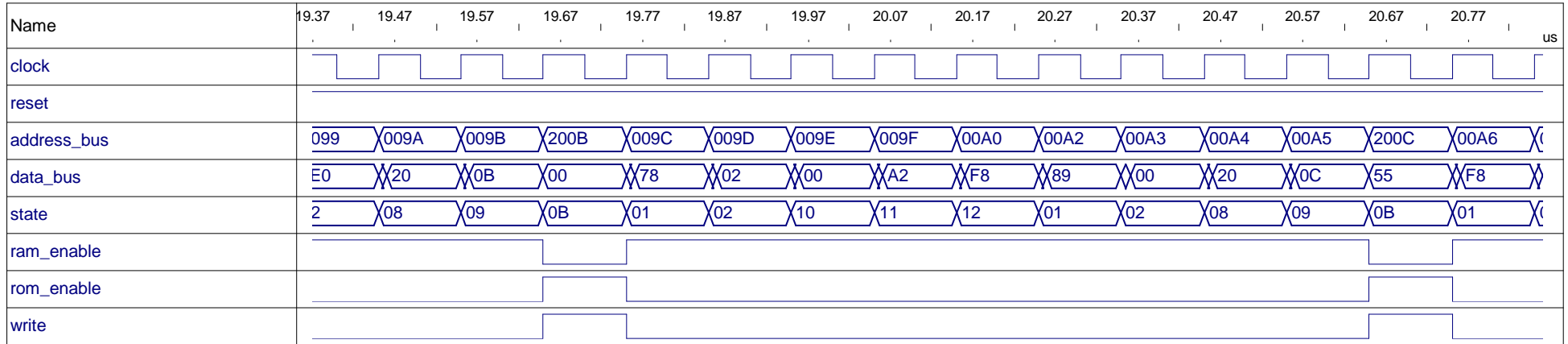
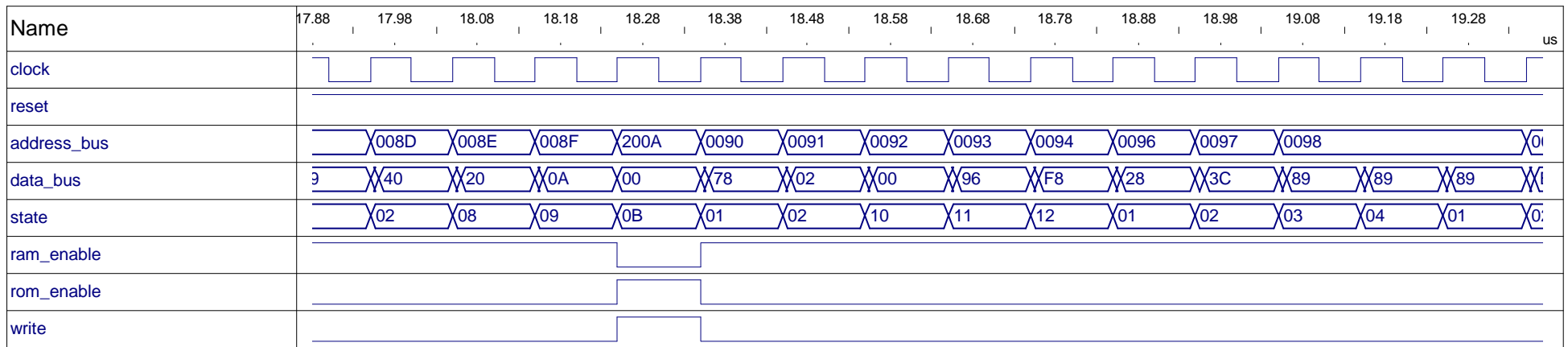
CPU test program B



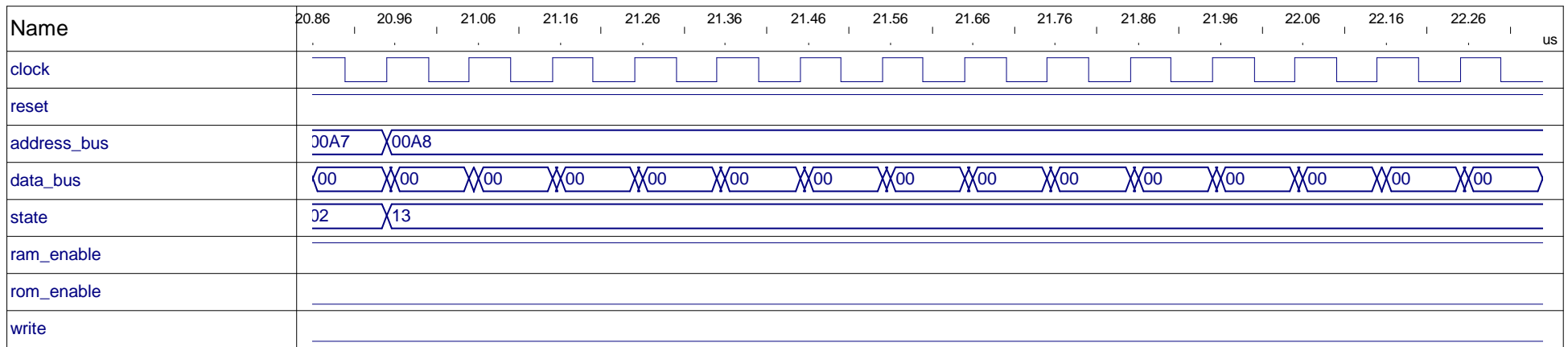
CPU test program B



CPU test program B

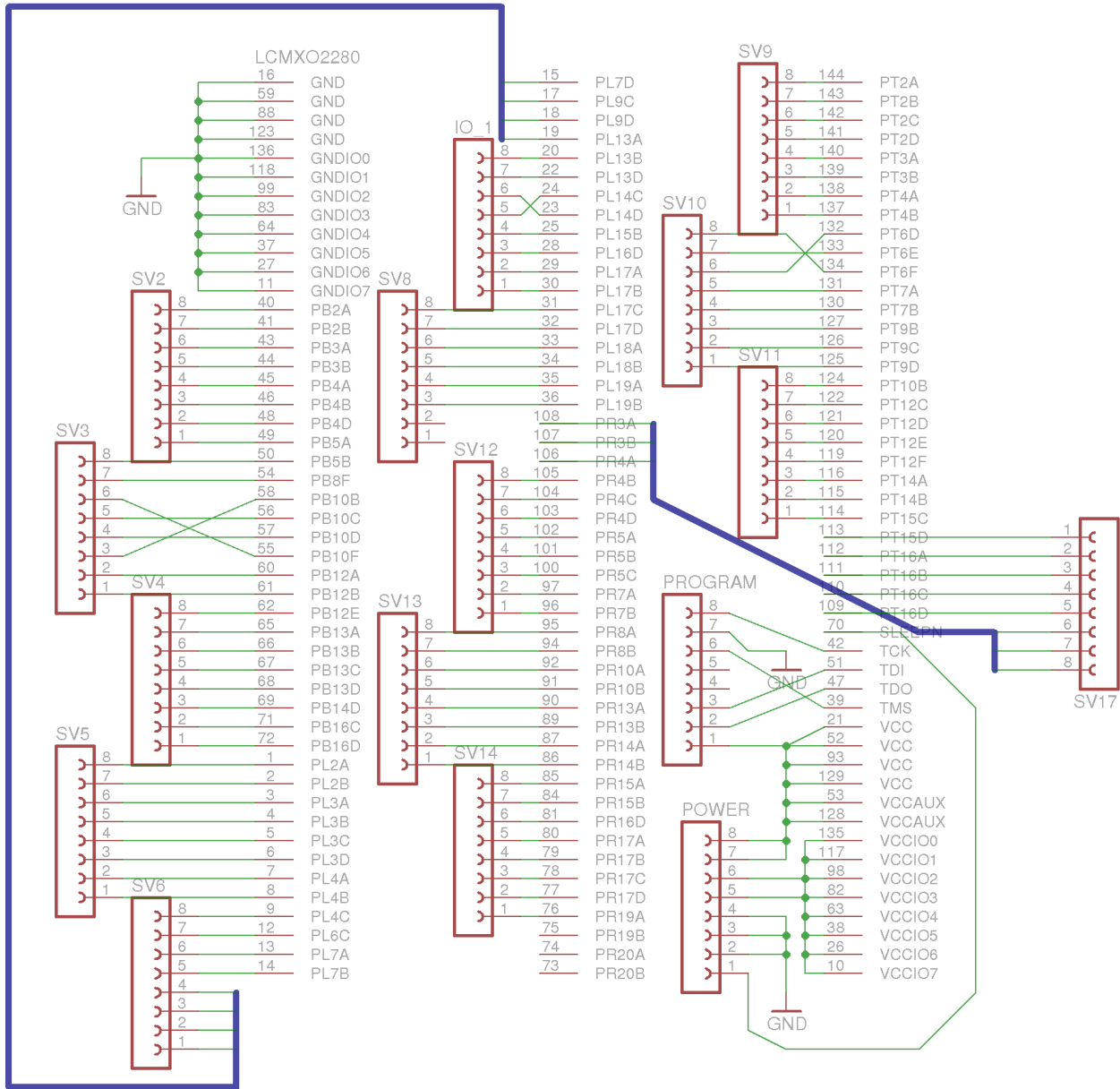


CPU test program B

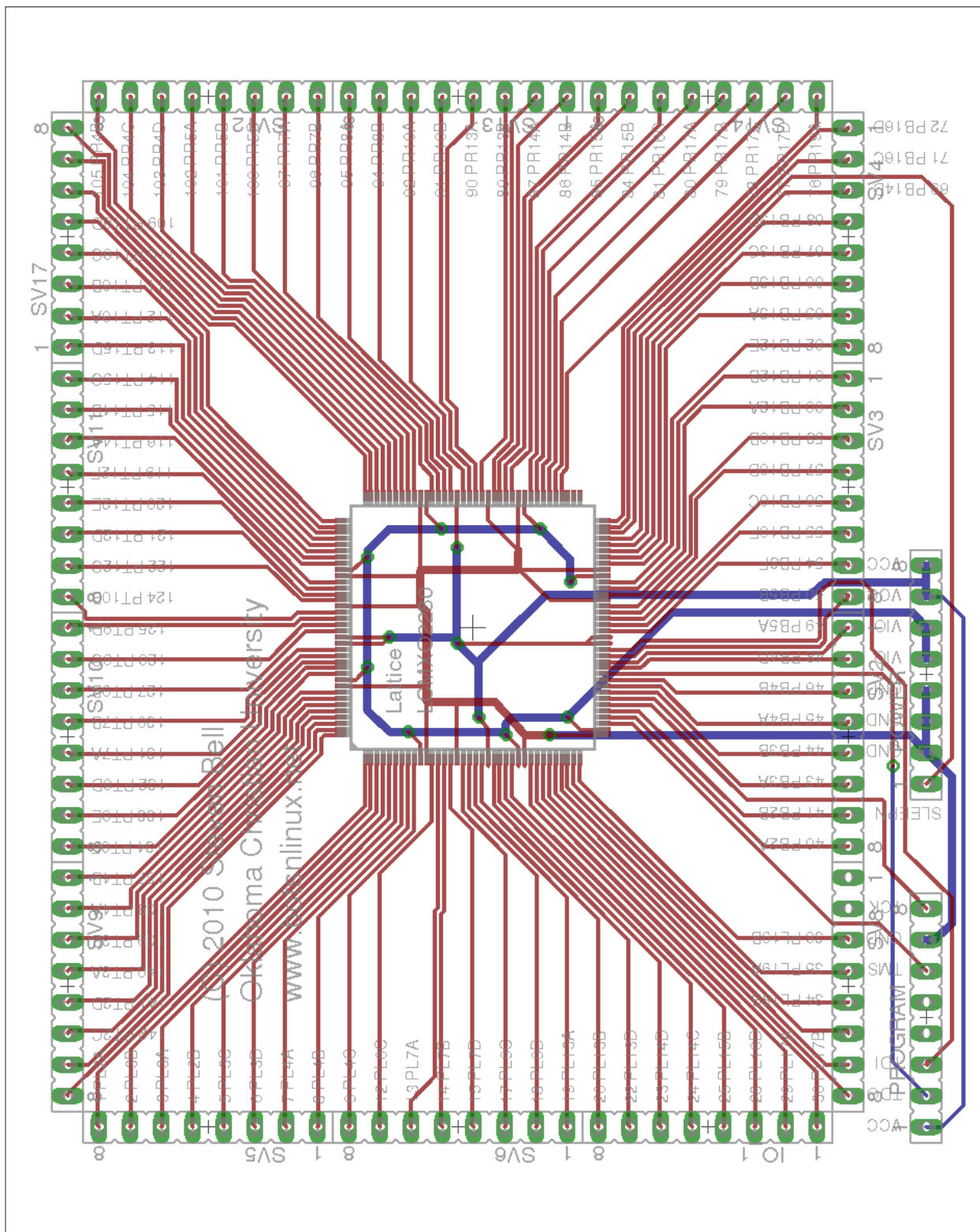


# D PCB diagrams

## D.1 Eagle schematic



## D.2 Eagle layout



## E Assembler code

```
1 /**
2  * \file main.cpp
3  * Main code file for Steven's (purposeful|powerful|pimpin'|pernicious)
4  *   assembler (SPASM)
5  * \author Steven Bell <steven.bell@student.oc.edu>
6  * \date 4 November 2010
7  * $LastChangedDate: 2010-12-14 23:24:10 -0600 (Tue, 14 Dec 2010) $
8  */
9
10 /*
11 * Command-line arguments
12 * Usage:
13 *   spasm [args] inputfile
14 *
15 * -o           Output file
16 * -v, --verbose Verbose mode, print out lots of information about what's
17 *               going on
18 * -d, --dump   Do a hex dump of the program after it is assembled
19 * -h, --help   Print this help message
20 */
21
22 #include <QCoreApplication>
23 #include <QStringList>
24 #include <QFile>
25 #include "cstdio"
26 #include "console.h"
27 #include "parser.h"
28
29 int main(int argc, char* argv[])
30 {
31     // Read the input arguments
32     // We don't really need QCoreApplication for anything except the arguments,
33     // so maybe there's
34     // a better way to do this?
35     QCoreApplication app(argc, argv);
36     QStringList argList = app.arguments();
37
38     // If the help flag was specified, then just print the help message and quit
39     if(argList.contains("-h") || argList.contains("--help")){
40         printf("SPASM - Steven's special-purpose assembler\n");
41         printf("Copyright (c) 2010 Steven Bell <steven.bell@student.oc.edu>\n\n");
42         printf("Usage:\n");
43         printf("  spasm [args] inputfile\n\n");
44         printf("  -o, --output      Output file\n");
45         printf("  -v, --verbose     Verbose mode, print out lots of information
46                     about what's going on\n");
47         printf("  -s, --simulator  Do a hex dump of the program for the Verilog
48                     testbench\n");
49         printf("  -l, --listing    Print a command listing useful for debugging\n");
50         ;
51         printf("  -h, --help       Print this help message\n");
52     }
```

```

46     printf(" Note that the -s, -l, and -v arguments are mutually exclusive.\n\n");
47     return(0);
48 }
49
50 // Get the input file name
51 QString inputFilePath(argList.last()); // Assume input file name is the last
    argument
52
53 // Determine the output file name
54 // If the -o flag was specified, use that name regardless of extension
55 QString outputFilePath;
56 if(argList.contains("-o")){
57     outputFilePath = argList.at(argList.indexOf("-o") + 1);
58 }
59 else if(argList.contains("--output")){
60     outputFilePath = argList.at(argList.indexOf("--output") + 1);
61 }
62 else{
63     // If no output file was specified, then use the input file name with the
        .S19 file extension
64     outputFilePath = inputFilePath + ".s19"; // Do something to strip off the
        extension
65 }
66
67 // Determine what kind of output to send to the terminal
68 bool verbose = false;
69 bool simDump = false;
70 bool listing = false;
71
72 if(argList.contains("-v") || argList.contains("--verbose")){
73     verbose = true;
74 }
75 else if(argList.contains("-s") || argList.contains("--simulator")){
76     simDump = true;
77 }
78 else if(argList.contains("-l") || argList.contains("--listing")){
79     listing = true;
80 }
81 Console::instance().setVerbose(verbose);
82
83 // Open the input file
84 QFile inputFile(inputFilePath);
85 if(!inputFile.open(QIODevice::ReadOnly | QIODevice::Text)){
86     printf("Failed to open file %s\n", inputFilePath.toAscii().constData());
87     return(1);
88 }
89
90 Parser p;
91 QByteArray programBytes;
92
93 // Run the first pass of the parser
94 // When undefined labels are encountered, the parser inserts 0x0000
95 // If undefined aliases are encountered, the parser throws an error

```

```

96  while(!inputFile.atEnd())
97  {
98      QString command(inputFile.readLine()); // Get the command string to parse
99      VERBOSE(p.linesRead(), command.trimmed().toAscii().constData()); // Print
      the command
100
101      // TODO: for each line, store the starting address so we can create a
      listing-style output
102      QByteArray commandBytes = p.parseLine(command);
103      programBytes.append(commandBytes);
104  }
105
106      // Fill in the labels which were undefined in the first pass
107      p.assignUnknownLabels(programBytes);
108
109      if(simDump){
110          for(int i = 0; i < programBytes.length(); i++){
111              printf("%02x\n", (unsigned char) programBytes[i]);
112          }
113      }
114
115      /* Create the output file (Motorola SREC format)
116      All values are are hexadecimal, but are written in pairs of ASCII
      characters.
117      The format follows this pattern:
118      | "S1" | Byte Count | Start address | Data Data Data Data | Checksum |
119      |      | 1 byte   | 2 bytes   | Up to 32 bytes   | 1 byte   |
120
121      */
122
123      // Open the file
124      QFile outputFile(outputFilePath);
125      outputFile.open(QIODevice::WriteOnly | QIODevice::Text);
126
127      int i = 0;
128      while(i < programBytes.length())
129      {
130          unsigned char checksum; // The checksum is the ones' complement of the sum
      of the byte count, address, and data
131
132          // Create the line header
133          int lineByteCount = programBytes.length() - i; // Number of bytes
      remaining
134          lineByteCount = (lineByteCount < 32) ? lineByteCount : 32; // If there are
      more than 32 bytes remaining, only print 32 of them
135          QString lineString = QString("S1%1%2").arg(lineByteCount + 3, 2, 16, QChar
      ('0')).arg(i, 4, 16, QChar('0'));
136          checksum += lineByteCount; // Add the data byte count to the checksum
137          checksum += i; // Add the starting address to the checksum
138
139          for(int j = 0; j < lineByteCount; j++)
140          {
141              // Add the bytes

```

```

142     lineString.append(QString("%1").arg((unsigned char) (programBytes[i]), 2,
143         16, QChar('0')));
143     checksum += programBytes[i];
144     i++;
145 }
146 // Write the checksum
147 checksum = ~checksum;
148 lineString.append(QString("%1").arg(checksum, 2, 16, QChar('0')));
149 outputFile.write(lineString.toAscii().constData());
150 outputFile.write("\n"); // Write a newline
151 }
152
153
154 return(0);
155 }

```

```

1  /** parser.cpp
2   * Class declaration for the parser class, which parses sets of instructions
3   * passed to it.
4   * Author: Steven Bell <steven.bell@student.oc.edu>
5   * Date: 24 November 2010
6   * $LastChangedDate: 2010-12-14 23:24:10 -0600 (Tue, 14 Dec 2010) $
7   */
8  #ifndef PARSER_H
9  #define PARSER_H
10
11  #include <QString>
12  #include <QHash>
13  #include <QByteArray>
14
15  class Parser
16  {
17
18  public:
19      typedef enum
20      {
21          NOP = 0x00,
22          ADD = 0x01,
23          SUB = 0x02,
24          MUL = 0x03,
25          AND = 0x04,
26          OR = 0x05,
27          LSR = 0x06,
28          LSL = 0x07,
29          COMP = 0x08,
30          ASR = 0x09,
31          ASL = 0x0a,
32          NEG = 0x0b, // Two's complement
33          // PASSTHROUGH (0x0c) is for internal use only
34          JUMP = 0x0f,
35          LOAD = 0x10,
36          STORE = 0x11,
37          MOVE = 0x12,

```

```

38     LSTACK = 0x13,
39     JSR = 0x14,
40     RSUB = 0x15,
41     RINT = 0x16,
42     HALT = 0x1F,
43     INVALID = 0xFF
44 } Opcode;
45
46 typedef enum
47 {
48     A = 0,
49     B = 1,
50     C = 2,
51     D = 3,
52     E = 4,
53     F = 5,
54     G = 6,
55     H = 7
56 } RegisterCode;
57
58 typedef enum
59 {
60     NONE = 0,
61     REGISTER = 2,
62     IMMEDIATE = 4,
63     ADDRESS = 8
64 } TokenType;
65
66 Parser();
67 QByteArray parseLine(QString line);
68 void assignUnknownLabels(QByteArray& code);
69 int linesRead(void) {return(mLineCount);}
70
71 private:
72 bool parseRegisterDest(QStringList* tokens, unsigned short* opcodeBytes);
73 Opcode stringToOpcode(QString str);
74 TokenType parseParameterToken(QString token, int allowable, unsigned short &
    value);
75
76 int mLineCount; ///< Counts how many lines have been parsed
77
78 ///< Address where the next instruction will be stored in memory, used for
    labels
79 unsigned int mByteLocation;
80
81 ///< Map of addresses to unknown labels
82 QHash<unsigned short, QString> mUnknownLabels;
83
84 ///< Map of defines and labels to addresses
85 QHash<QString, QString> mDefineTable;
86
87 };
88
89 #endif // PARSER_H

```

```

1  /** parser.cpp
2   * Code to parse instructions given one at a time. Labels are held
3   * persistently and handled appropriately.
4   * Author: Steven Bell <steven.bell@student.oc.edu>
5   * Date: 24 November 2010
6   * $LastChangedDate: 2010-12-14 23:24:10 -0600 (Tue, 14 Dec 2010) $
7   */
8  #include <cstdio>
9  #include <QStringList>
10 #include "console.h"
11 #include "parser.h"
12
13
14 /** Constructor
15  */
16 Parser::Parser()
17 {
18     mByteLocation = 0;
19     mLineCount = 1; // Line numbers start at 1, not 0!
20 }
21
22 /** Parses a single line of code, which is assumed to contain a single
23     instruction at most.
24     * \param line Text string to parse.
25     */
26 QByteArray Parser::parseLine(QString command)
27 {
28     // Strip off any comment
29     int commentPos = command.indexOf('*');
30     if(commentPos != -1){ // If a comment delimiter was found,
31         command = command.left(commentPos);
32     }
33     QStringList tokens = command.split(QRegExp(",|\\s"), QString::SkipEmptyParts
34         );
35     // TODO: should use a regex here to make sure that we don't end up with
36     // funky characters in the labels.
37     while(!tokens.isEmpty() && tokens.first().contains(QRegExp("[a-zA-Z]\\w+:$"
38         )))
39     {
40         // Parse the label(s) from the front, if there are any
41         QString label = tokens.first();
42         label = label.remove(QChar(':')); // Take the colon off the end
43         VERBOSE(mLineCount, "Label_found_at_0x%04x:_%s\\n", mByteLocation, label.
44             toAscii().constData());
45         // Create a string with the memory address so we can use it like any other
46         // define
47         mDefineTable[label] = QString("0x%1").arg(mByteLocation, 4, 16, QChar('0'
48             ));
49         tokens.removeFirst();
50     }
51 }

```

```

48
49 if(tokens.isEmpty()){ // If there's nothing, or only labels,
50     VERBOSE(mLineCount, "Empty_line!"); // Then move on
51     mLineCount++;
52     return(QByteArray());
53 }
54
55 QByteArray commandBytes; // Complete set of bytes, including immediates and
    addresses
56 unsigned short opcodeBytes = 0; // Pair of bytes used to assemble the opcode
57 unsigned short parsedBytes; // Values returned from each token parsing
58 Parser::TokenType parseResult; // Return type of token that was parsed
59
60 QString codeString = tokens.first(); // Save this in a separate variable so
    we can use it to differentiate the jumps
61 Parser::Opcode code = stringToOpcode(codeString);
62 tokens.removeFirst();
63 opcodeBytes = code << 11; // Move the code into the 5 highest bytes
64 mByteLocation += 2;
65
66 switch(code) {
67     case Parser::NOP:
68     case Parser::HALT:
69     case Parser::RSUB:
70     case Parser::RINT:
71         VERBOSE(mLineCount, "No_args_required");
72         break;
73     case Parser::ADD:
74     case Parser::SUB:
75     case Parser::MUL:
76     case Parser::AND:
77     case Parser::OR:
78         VERBOSE(mLineCount, "Two_args_required,_register_and_register/immediate"
93         );
94
95     parseRegisterDest(&tokens, &opcodeBytes);
96
97     parseResult = parseParameterToken(tokens.first(), Parser::REGISTER |
98     Parser::IMMEDIATE, parsedBytes);
99     tokens.removeFirst();
100     if(parseResult == REGISTER){
101         opcodeBytes = opcodeBytes | (parsedBytes << 5); // Put the source
102         register into bytes 7-5
103         // Don't have to set the source type, since it's already 00
104     }
105     else if(parseResult == IMMEDIATE){
106         // Append the immediate to the extra bytes
107         commandBytes.append((char)parsedBytes);
108         mByteLocation += 1;
109         opcodeBytes = opcodeBytes | (0x0400); // Set the source type bits to
110         immediate
111     }
112     else{

```

```

95     ERROR(mLineCount, "Expected_register_or_immediate_as_second_argument")
96     ;
97     }
98     break;
99     case Parser::LSR:
100    case Parser::LSL:
101    case Parser::ASR:
102    case Parser::ASL:
103    case Parser::COMP:
104    case Parser::NEG:
105        VERBOSE(mLineCount, "One_register_argument_required");
106        parseRegisterDest(&tokens, &opcodeBytes); // The first argument must
107        always be a the destination register
108        break;
109    case Parser::JUMP:
110        if(codeString.toUpper() == "JCAR"){
111            opcodeBytes = opcodeBytes | (0x0001); // Carry code is binary 01
112        }
113        else if(codeString.toUpper() == "JZERO"){
114            opcodeBytes = opcodeBytes | (0x0002); // Zero code is binary 10
115        }
116        else if(codeString.toUpper() == "JNEG"){
117            opcodeBytes = opcodeBytes | (0x0003); // Negative code is binary 11
118        }
119        // Jump always code is binary 00, so we don't have to do anything
120        // Fall through the case so that we can get the memory argument
121    case Parser::LSTACK:
122    case Parser::JSR:
123        VERBOSE(mLineCount, "One_memory_argument_required");
124        parseResult = parseParameterToken(tokens.first(), Parser::ADDRESS,
125        parsedBytes);
126        tokens.removeFirst();
127        if(parseResult == ADDRESS){
128            commandBytes.append((char)(parsedBytes >> 8));
129            commandBytes.append((char)parsedBytes);
130            mByteLocation += 2;
131        }
132        else{
133            ERROR(mLineCount, "Expected_address_or_label_as_first_argument");
134        }
135        break;
136    case Parser::LOAD:
137        VERBOSE(mLineCount, "Two_args_required,_register_and_register/immediate/
138        memory");
139        parseRegisterDest(&tokens, &opcodeBytes); // The first argument must
140        always be a the destination register
141        parseResult = parseParameterToken(tokens.first(), Parser::REGISTER |
142        Parser::IMMEDIATE | Parser::ADDRESS, parsedBytes);
143        tokens.removeFirst();
144        if(parseResult == REGISTER){ // COPY
145            opcodeBytes = opcodeBytes | (parsedBytes << 5); // Put the source
146            register as
147            // Don't have to set the source type, since it's already 00

```

```

142     }
143     else if(parseResult == IMMEDIATE){
144         commandBytes.append((char)parsedBytes);
145         mByteLocation += 1;
146         opcodeBytes = opcodeBytes | (0x0400); // Set the source type bits to
            immediate
147     }
148     else if(parseResult == ADDRESS){
149         commandBytes.append((char) (parsedBytes >> 8));
150         commandBytes.append((char)parsedBytes);
151         mByteLocation += 2;
152         opcodeBytes = opcodeBytes | (0x0200); // Set the source type bits to
            memory
153     }
154
155     break;
156 case Parser::STORE:
157     VERBOSE(mLineCount, "Two_args_required,_register_and_memory");
158
159     parseResult = parseParameterToken(tokens.first(), Parser::REGISTER,
        parsedBytes);
160     tokens.removeFirst();
161     if(parseResult == REGISTER){
162         opcodeBytes = opcodeBytes | (parsedBytes << 5);
163     }
164     else{
165         ERROR(mLineCount, "Expected_register_as_first_argument");
166     }
167
168     parseResult = parseParameterToken(tokens.first(), Parser::ADDRESS,
        parsedBytes);
169     tokens.removeFirst();
170     if(parseResult == ADDRESS){
171         commandBytes.append((char) (parsedBytes >> 8));
172         commandBytes.append((char)parsedBytes);
173         mByteLocation += 2;
174         opcodeBytes = opcodeBytes | (0x0100); // Set the destination type bits
            to memory
175     }
176     else{
177         ERROR(mLineCount, "Expected_address_as_second_argument");
178     }
179
180     break;
181 case Parser::MOVE:
182     // This code is repeated several times essentially verbatim. Time for a
        function?
183     VERBOSE(mLineCount, "Two_addresses_required\n");
184     parseResult = parseParameterToken(tokens.first(), Parser::ADDRESS,
        parsedBytes);
185     tokens.removeFirst();
186     if(parseResult == ADDRESS){
187         commandBytes.append((char) (parsedBytes >> 8));
188         commandBytes.append((char)parsedBytes);

```

```

189     mByteLocation += 2;
190 }
191 else{
192     ERROR(mLineCount, "Expected_address_as_first_argument");
193 }
194 parseResult = parseParameterToken(tokens.first(), Parser::ADDRESS,
    parsedBytes);
195 tokens.removeFirst();
196 if(parseResult == ADDRESS){
197     commandBytes.append((char)(parsedBytes >> 8));
198     commandBytes.append((char)parsedBytes);
199     mByteLocation += 2;
200 }
201 else{
202     ERROR(mLineCount, "Expected_address_as_second_argument");
203 }
204
205
206     break;
207 default:
208     ERROR(mLineCount, "Unrecognized_opcode");
209     break;
210 }
211
212 // Make sure that we've used up all of the arguments.  If not, print a
    warning
213 if(!tokens.isEmpty()){
214     WARNING(mLineCount, "Unused_arguments");
215 }
216
217 commandBytes.push_front((char)(opcodeBytes & 0x00ff));
218 commandBytes.push_front((char)(opcodeBytes >> 8));
219
220 mLineCount++;
221 // Byte location was already handled in-line
222
223 return(commandBytes);
224 }
225
226 /** This function goes through all of the locations in the unknown labels list
    and
227 * replaces the placeholder address with the correct value.  If the label is
    not
228 * found, then it prints an error.
229 */
230 void Parser::assignUnknownLabels(QByteArray& code)
231 {
232     QHash<unsigned short, QString>::iterator i;
233     for(i = mUnknownLabels.begin(); i != mUnknownLabels.end(); i++)
234     {
235         if(mDefineTable.contains(i.value())){
236             QString replacementString = mDefineTable[i.value()];
237             bool conversionOk; // Assume that it works; since we created the value!

```

```

238     unsigned int replacementValue = replacementString.mid(3).toUShort(&
        conversionOk, 16);
239
240     // TODO: This will fail badly if the code has multiple segments or doesn
        't start at 0.
241     code[i.key()] = (replacementValue >> 8);
242     code[i.key() + 1] = (replacementValue & 0x00FF);
243
244
245     VERBOSE(mLineCount, "Replacing_label_\">%s\">%with_0x%04x", i.value().
        toAscii().constData(), replacementValue);
246 }
247 else{
248     ERROR(mLineCount, "Unknown_label_\">%s\%">", i.value().toAscii().constData
        ());
249 }
250 }
251 }
252
253
254 /* This function removes one of the tokens from the list
255 */
256 bool Parser::parseRegisterDest(QStringList* tokens, unsigned short*
    opcodeBytes)
257 {
258     unsigned short parsedBytes;
259     Parser::TokenType parseResult = parseParameterToken(tokens->first(), Parser
        ::REGISTER, parsedBytes);
260     tokens->removeFirst();
261     if(parseResult == REGISTER){
262         *opcodeBytes = *opcodeBytes | (parsedBytes << 2); // Put the destination
            register into bytes 4-2
263         return(true);
264     }
265     else{
266         ERROR(mLineCount, "Expected_register_as_first_argument");
267         return(false);
268     }
269 }
270
271
272 /** Converts a string token (e.g, ADD or HALT) into an enum constant
    representing
273 * the opcode.
274 */
275 Parser::Opcode Parser::stringToOpcode(QString str)
276 {
277     str = str.toUpper(); // Force everything to uppercase
278
279     if(str == "NOP"){
280         return(NOP);
281     }
282     else if(str == "ADD"){
283         return(ADD);

```

```

284 }
285 else if(str == "SUB"){
286     return(SUB);
287 }
288 else if(str == "MUL"){
289     return(MUL);
290 }
291 else if(str == "AND"){
292     return(AND);
293 }
294 else if(str == "OR"){
295     return(OR);
296 }
297 else if(str == "LSR"){
298     return(LSR);
299 }
300 else if(str == "LSL"){
301     return(LSL);
302 }
303 else if(str == "COMP"){
304     return(COMP);
305 }
306 else if(str == "ASR"){
307     return(ASR);
308 }
309 else if(str == "ASL"){
310     return(ASL);
311 }
312 else if(str == "NEG"){
313     return(NEG);
314 }
315 else if(str == "JMP" || str == "JCAR" || str == "JZERO" || str == "JNEG"){
316     return(JUMP);
317 }
318 else if(str == "LOAD" || str == "COPY"){
319     return(LOAD);
320 }
321 else if(str == "STORE"){
322     return(STORE);
323 }
324 else if(str == "MOVE"){
325     return(MOVE);
326 }
327 else if(str == "LSTACK"){
328     return(LSTACK);
329 }
330 else if(str == "JSR"){
331     return(JSR);
332 }
333 else if(str == "RSUB"){
334     return(RSUB);
335 }
336 else if(str == "RINT"){
337     return(RINT);

```

```

338     }
339     else if(str == "HALT"){
340         return(HALT);
341     }
342     else{
343         return(INVALID);
344     }
345 }
346
347
348 /** Takes a single token, determines what type of token it is, and parses its
349     * value.
350     * If the token is a label or macro, it looks it up in the appropriate table
351     * and
352     * substitutes that value.
353     * \param token The token to parse.
354     * \param allowable Acceptable types of tokens. Multiple types can be
355     * selected using bitwise OR. If
356     * a token of the selected type is not found, this function will return
357     * failure.
358     * \param value 2-byte variable which will hold the parsed value.
359     * \return The type of token that was parsed. This could be a 3-bit register
360     * (registerCode),
361     * a one byte immediate value, a two byte immediate address, or a failure.
362     */
363 Parser::TokenType Parser::parseParameterToken(QString token, int allowable,
364     unsigned short &value)
365 {
366     bool conversionOk;
367
368     // Use a set of regular expressions to sort the tokens
369     // If the token is a single letter, A-H (a-h) then it is a register
370     if((allowable & REGISTER) != 0) && token.contains(QRegExp("^[a-hA-H]$"))){
371         VERBOSE(mLineCount, "Token_is_register");
372         char regChar = token.at(0).toLowerCase().toAscii();
373         value = regChar - 97; // 97 is ascii for 'a'
374         return(REGISTER);
375     }
376
377     // If the token begins with 0x, then it is a hexadecimal address
378     else if((allowable & ADDRESS) != 0) && token.contains(QRegExp("^0x[0-9a-fA-
379         F_]+$"))){
380         VERBOSE(mLineCount, "Token_is_hex_address");
381         value = token.toUShort(&conversionOk, 16); // Qt doesn't mind the leading
382             0x
383         if(conversionOk){
384             return(ADDRESS);
385         }
386         else{
387             return(NONE);
388         }
389     }
390     // If the token begins with 0b, then it is a binary address

```

```

383     else if(((allowable & ADDRESS) != 0) && token.contains(QRegExp("^0b[01_]+$")
    )){
384         VERBOSE(mLineCount, "Token_is_binary_address");
385         value = token.mid(2).toUShort(&conversionOk, 2); // Drop the "0b" and
            convert
386         if(conversionOk){
387             return(ADDRESS);
388         }
389         else{
390             return(NONE);
391         }
392     }
393     // If the token begins with any digit, then it is a decimal address
394     else if(((allowable & ADDRESS) != 0) && token.contains(QRegExp("[+-]?\\d+$"
    )))){
395         VERBOSE(mLineCount, "Token_is_decimal_address");
396         value = token.toUShort(&conversionOk, 10); // Qt automatically handles
            negative values
397         if(conversionOk){
398             return(ADDRESS);
399         }
400         else{
401             return(NONE);
402         }
403     }
404
405     // If the token begins with #0x, then it is a hexadecimal immediate
406     else if(((allowable & IMMEDIATE) != 0) && token.contains(QRegExp("#0x[0-9a-
    fA-F_]+$"))){
407         VERBOSE(mLineCount, "Token_is_hex_immediate");
408         value = token.mid(3).toUShort(&conversionOk, 16); // Drop the #0x
409         if(conversionOk){
410             return(IMMEDIATE);
411         }
412         else{
413             return(NONE);
414         }
415     }
416     // If the token begins with #0b, then it is a binary address
417     else if(((allowable & IMMEDIATE) != 0) && token.contains(QRegExp("#0b[01_]+
    $"))){
418         VERBOSE(mLineCount, "Token_is_binary_immediate");
419         value = token.mid(3).toUShort(&conversionOk, 2); // Drop the "#0b" and
            convert
420         if(conversionOk){
421             return(IMMEDIATE);
422         }
423         else{
424             return(NONE);
425         }
426     }
427     // If the token begins with '#' and then any digit, then it is a decimal
        address

```

```

428     else if(((allowable & IMMEDIATE) != 0) && token.contains(QRegExp("^#[+-]?\\d
         +$"))){
429         VERBOSE(mLineCount, "Token_is_decimal_immediate");
430         value = token.mid(1).toUShort(&conversionOk, 10); // Drop the "#"
431         if(conversionOk){
432             return(IMMEDIATE);
433         }
434         else{
435             return(NONE);
436         }
437     }
438     else if(token.contains(QRegExp("^([a-zA-Z]\\w+$"))){
439         // If the token starts with a letter, contains only alphanumeric
           characters and
440         // underscores, and is at least 2 characters long, then see if it's a
           label or macro
441
442         if(mDefineTable.contains(token)){
443             // Replace the macro or label with the appropriate token
444             // Call this function recursively on the replaced token
445             return(parseParameterToken(mDefineTable[token], allowable, value));
446         }
447         else{
448             // If the table doesn't contain what we're looking for, just substitute
           a placeholder
449             // address and put this location into the unknown labels table.
450             if(allowable & ADDRESS){
451                 VERBOSE(mLineCount, "Encountered_unknown_label_%s_at_0x%04x", token.
           toAscii().constData(), mByteLocation);
452                 mUnknownLabels[mByteLocation] = token;
453                 value = 0;
454                 return(ADDRESS);
455             }
456             else{
457                 // If an address isn't allowed here, then it's an error
458                 ERROR(mLineCount, "Unexpected_token_\"%s\"", token.toAscii().constData
           ());
459                 return(NONE);
460             }
461         }
462     }
463     else{
464         ERROR(mLineCount, "Unexpected_token_\"%s\"", token.toAscii().constData());
465         return(NONE);
466     }
467 }

```

```

1  /** \file console.h
2  * Singleton class which handles printing of messages, errors and warnings
3  * to the console. It contains flags which can be used to turn messages on
4  * and off, and keeps track of the total number of errors/warnings.
5  * \author Steven Bell <steven.bell@student.oc.edu>
6  * \date 12 December 2010
7  */

```

```

8
9 class Console
10 {
11 public:
12     static Console& instance();
13     void setVerbose(bool verbose);
14     void printError(int lineNum, const char* format, ...);
15     void printWarning(int lineNum, const char* format, ...);
16     void printMessage(int lineNum, const char* format, ...);
17     void printVerbose(int lineNum, const char* format, ...);
18     void printSummary(void);
19     bool errorOccured(void) {return(mErrorCount > 0);}
20
21 private:
22     Console(); ///< Private singleton constructor
23     static Console mInstance;
24     int mErrorCount; ///< Number of times the error message function has been
        called
25     int mWarningCount; ///< Number of times the warning function has been called
26     bool mVerbose; ///< Whether or not to print verbose stuff.
27
28 };
29
30 // Convenience macros
31 #define ERROR(line, ...) Console::instance().printError(line, __VA_ARGS__)
32 #define WARNING(line, ...) Console::instance().printWarning(line, __VA_ARGS__)
33 #define MESSAGE(line, ...) Console::instance().printMessage(line, __VA_ARGS__)
34 #define VERBOSE(line, ...) Console::instance().printVerbose(line, __VA_ARGS__)

```

```

1 /** \file console.cpp
2  * \author Steven Bell <steven.bell@student.oc.edu>
3  * \date 12 December 2010
4  */
5
6 #include <cstdio>
7 #include <stdarg.h>
8 #include "console.h"
9
10 // Single instance instantiation
11 Console Console::mInstance;
12
13 Console::Console()
14 {
15     mErrorCount = 0;
16     mWarningCount = 0;
17 }
18
19 Console& Console::instance(void)
20 {
21     return(mInstance);
22 }
23
24 void Console::printError(int lineNum, const char*format, ...)
25 {

```

```

26     mErrorCount++;
27
28     va_list argptr;
29     va_start(argptr, format);
30     printf("%d_Error:_", lineNumber);
31     vprintf(format, argptr);
32     printf("\n");
33     va_end(argptr);
34 }
35
36 void Console::printWarning(int lineNumber, const char*format, ...)
37 {
38     mWarningCount++;
39
40     va_list argptr;
41     va_start(argptr, format);
42     printf("%d_Warning:_", lineNumber);
43     vprintf(format, argptr);
44     printf("\n");
45     va_end(argptr);
46 }
47
48 void Console::printMessage(int lineNumber, const char*format, ...)
49 {
50     va_list argptr;
51     va_start(argptr, format);
52     printf("%d:_", lineNumber);
53     vprintf(format, argptr);
54     printf("\n");
55     va_end(argptr);
56 }
57
58 void Console::printVerbose(int lineNumber, const char*format, ...)
59 {
60     if(mVerbose){
61         va_list argptr;
62         va_start(argptr, format);
63         printf("%d:_", lineNumber);
64         vprintf(format, argptr);
65         printf("\n");
66         va_end(argptr);
67     }
68 }
69
70 void Console::printSummary(void)
71 {
72     printf("\n");
73     printf("%d_errors,_%d_warnings\n", mErrorCount, mWarningCount);
74 }
75
76
77 void Console::setVerbose(bool verbose)
78 {
79     mVerbose = verbose;

```



## **F Final test program**

The code on the following pages is based on the test code provided on Blackboard and modified for my assembler. The code was syntax-highlighted with Vim.

```

* Final test program for IC design, Part A
* Steven Bell
* 13 December 2010
*
* Several changes were made from the code given on Blackboard to this code:
* - The assembler takes two opcodes at most, so adding an immediate to a register
*   puts the value back into that same register. A COPY is done first to achieve
*   the effect of adding an immediate and putting the result in another register.
* - The registers are labeled a through h.
* - RAM is located at 0x2000 through 0x3999.

```

```

* Test NOP

```

```

NOP

```

```

* Test LOAD and STORE with RAM

```

```

LOAD a, #0x55 * Load immediate
STORE a, 0x2000 * Store to RAM, 0x55 expected
LOAD b, 0x2000 * Load back from RAM into another register
STORE b, 0x2001 * Store back to RAM, 0x55 expected

```

```

* Test ADD

```

```

COPY c, b * Copy from b into c
ADD c, #0x22 * Add an immediate to c
STORE c, 0x2002 * Expect 0x77

```

```

* Test flags from ADD

```

```

LOAD e, #0x5A
ADD b, a * Should give 0xAA in b
ADD e, #0xA5 * Should give 0xFF in e
JCAR end * Should not jump
STORE b, 0x2003 * Should be 0xAA
STORE e, 0x2004 * Should be 0xFF
JZERO end * Should not jump

```

```

* Test SUBtract and flags

```

```

LOAD d, #0xA9
COPY f, b * Copy b to f (should be 0xAA)
SUB f, #0x33 * Subtract 0x33, f should be 0x77
SUB d, c * Subtract c from d, d should be 0x32
JCAR end * Should not jump
JZERO end * Should not jump
STORE d, 0x2005 * Should be 0x32
STORE f, 0x2006 * Should be 0x77
JMP pass1
HALT

```

```

pass1:

```

```

COPY a, e * Copy 0xFF into a
ADD a, #0xFF * Add 0xFF, should give 0xFE with carry
STORE a, 0x2007 * Should be 0xFE
JZERO end * Should not jump
JCAR pass2 * Should jump
HALT

```

```

pass2:

```

```

ADD a, #0x01 * Should give 0xFF, no carry
JZERO end * Should not jump
JCAR end * Should not jump
STORE a, 0x2008 * Should give 0xFF
ADD a, #0x01 * Should give 0x00 with a carry
JZERO pass3 * Should jump
HALT

```

```

pass3:

```

```

STORE a, 0x2009 * Should give 0x00
SUB d, #0x21 * Should give 0x11 in d
JZERO end * Should not jump
JCAR end * Should not jump
STORE d, 0x200A * Should give 0x11
SUB d, #0x11 * Should give 0x00 in d
JCAR end * Should not jump
JZERO pass4 * Should jump
HALT

```

```

pass4:

```

```

STORE d, 0x200B * Should give 0x00
SUB d, #0x33 * Should give -0x33 = 0xCD in d
JZERO end * Should not jump
JNEG pass5 * Should jump
HALT

pass5:
STORE d, 0x200C * Should give -0x33 = 0xCD
ADD f, #0x00 * f should contain 0x77, and it shouldn't change!
JCAR end * Should not branch
SUB f, e * e should contain 0xFF, giving 0x78 with a borrow
JZERO end * Should not jump
JNEG end * Should not jump (the resulting value is positive)
JCAR pass6 * Should jump (a borrow occurred)

* Test multiply
pass6:
STORE f, 0x200D * Should give 0x78
COPY g, b * Should give 0xAA in g
MUL g, #0xA5 * Multiply giving 6D92
STORE g, 0x200E * Store the top half of the multiply (0x6D)
STORE h, 0x200F * Store the bottom half of the multiply (0x92)
LOAD h, #0xBB * Load 0xBB into h
MUL h, g * Multiply the top and bottom half of gh together, result should be 0x4F9F
STORE g, 0x2010 * Write the result out (0x4F)
STORE h, 0x2011 * (0x9F)

* Test unary ALU operations
COPY b, a * b should contain 0x00
COMP b * b should be 0xFF
STORE b, 0x2012
COMP b * Complement again, should give 0x00
STORE b, 0x2013
JCAR end * Shouldn't jump
JZERO pass7 * Should jump
HALT

pass7:
COPY e, d * e should now contain 0xCD = 0b 1100 1101
LSR e * Logical shift right
STORE e, 0x2014 * Should be 0b 0110 0110 = 0x66
LSR e * Logical shift right again
STORE e, 0x2015 * Should be 0x 0011 0011 = 0x33
COPY g, e * g should now be 0x33
LSL g * Logical shift left
STORE g, 0x2016 * Should be 0x 0110 0110 = 0x66
LSL g
STORE g, 0x2017 * Should be 0x 1100 1100 = 0xCC

end:
NOP
NOP
HALT

```

```

* Final test program for IC design, Part B
* Steven Bell
* 14 December 2010
*
* Several changes were made from the code given on Blackboard to this code:
* - The assembler takes two opcodes at most, so adding an immediate to a register
*   puts the value back into that same register. A COPY is done first to achieve
*   the effect of adding an immediate and putting the result in another register.
* - The registers are labeled a through h.
* - RAM is located at 0x2000 through 0x3999.

* Begin by loading registers with their nominal values from part A

LOAD a, #0x55
LOAD b, #0xFF
LOAD c, #0xFF
LOAD d, #0x33
* e will be assigned later
LOAD f, #0x03
* g will be assigned later
LOAD h, #0xC0

* Continue testing shifts
COPY e, d * Copy 0x33 = 0b 0011 0011 into e
LSR e * Shift e right (shift in a 0, giving 0b 0001 1001 = 0x19)
STORE e, 0x2000 * Should give 0x19
LSR e * 0b 0000 1100 = 0x0C
STORE e, 0x2001 * Should give 0x0C
COPY g, e
LSL g * 0b 0001 1000 = 0x18
STORE g, 0x2002 * Should give 0x18
LSL g * 0b 0011 0000 = 0x30
STORE g, 0x2003 * Should give 0x30

* Test shifts with jumps
LSR f * Shift 0b 0000 0011 to get 0b 0000 0001 with a carry
JCAR shift1 * Should jump
HALT

shift1:
LSR f * f should now be 0x00
JZERO shift2 * Should jump
HALT

shift2:
LSL h * h was 0xC0 = 0b 1100 0000, should now be 0b 1000 0000 with a carry
JCAR shift3 * Should jump
HALT

shift3:
LSL h * h will now be 0x00
JZERO pass8 * Should jump
HALT

* Test AND
pass8:
COPY f, a * Copy 0x55 into f
AND f, #0x0F * Mask the top four bits
STORE f, 0x2004 * Should be 0x05
AND e, a * e = 0b 0000 1100, a = 0b 0101 0101
STORE e, 0x2005 * Should be 0x04
COPY b, f * Copy 0x00 into b
AND b, #0xF0
STORE b, 0x2006 * Should be 0x00
JZERO pass9
HALT

* Test AND of 0x00 and 0xFF, and jump on zero
pass9:
AND c, b * c = 0xFF, b = 0x00
STORE c, 0x2007 * Should be 0x00
JZERO pass10 * Should jump
HALT

* Test OR

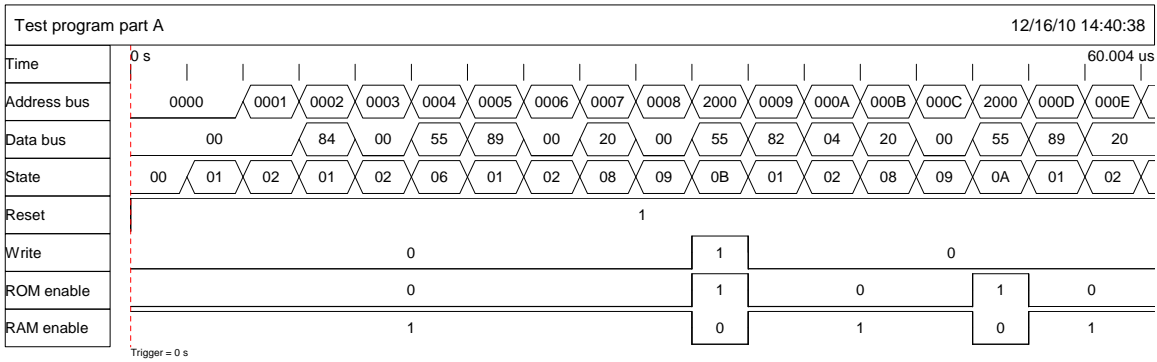
```

```
pass10:
COPY c, a * Copy 0x55 into c
OR c, #0xA0 * 0b 0101 0101 OR 0b 1010 0000 gives 0xF5
STORE c, 0x2008 * Should be 0xF5
OR g, e * g = 0x30 OR e = 0x04
STORE g, 0x2009 * Should be 0x34
COPY c, h * Copy 0x00 into c
OR c, #0x00
STORE c, 0x200A * Should be 0x00
JZERO pass11 * Should jump
HALT
```

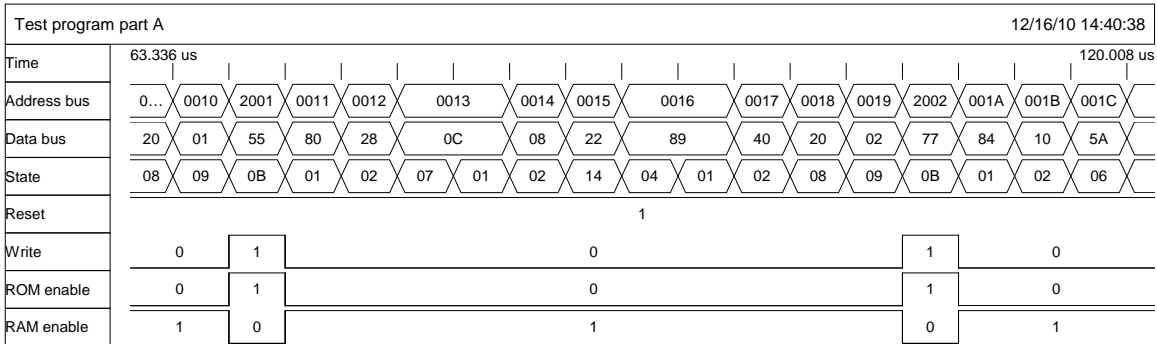
```
pass11:
OR h, b * 0x00 OR 0x00
STORE h, 0x200B * Should be 0x00
JZERO end * Should jump
HALT
```

```
end:
STORE a, 0x200C * Should be 0x55
MOVE 0x200C, 0x200D * Put 0x55 into 0x200C
LOAD b, 0x200D
HALT
```

## G Logic analyzer captures



Time -23.957549 ns to 60.804259431 us Page 1



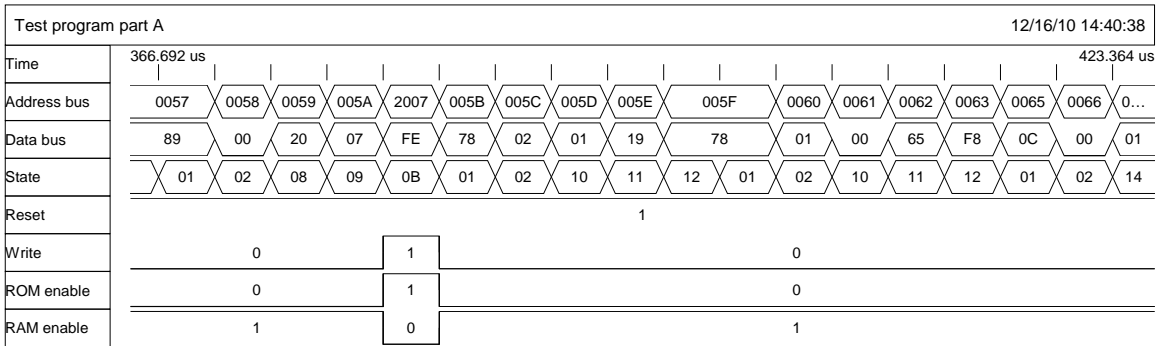
Time 60.816238206 us to 121.644455186 us Page 2

Test program part A													12/16/10 14:40:38					
Time	123.34 us												180.012 us					
Address bus	001E	001F	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	2003	0...				
Data bus	04	0C	10	A5	78	01	19	89	20	03	AA	89						
State	02	03	04	01	02	14	04	01	02	10	11	12	01	02	08	09	0B	01
Reset	1																	
Write	0															1	0	
ROM enable	0															1	0	
RAM enable	1															0	1	

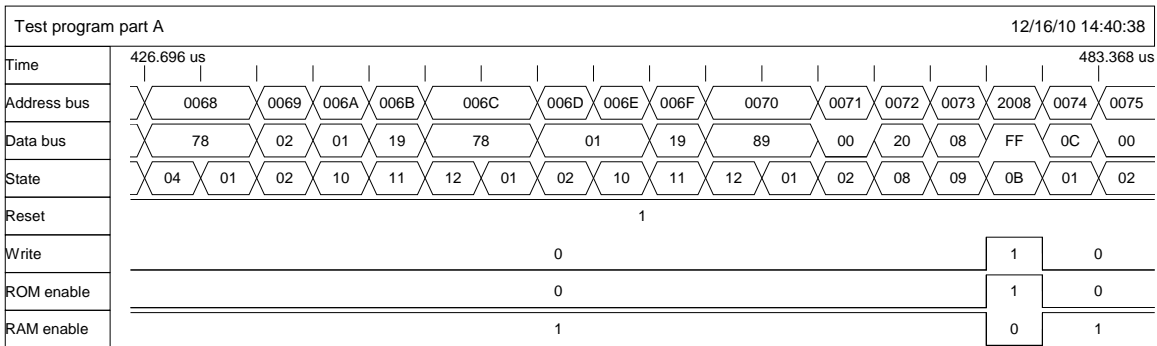
Test program part A													12/16/10 14:40:38					
Time	183.348 us												240.016 us					
Address bus	002B	002C	002D	2004	002E	002F	0030	0031	0032	0033	0034	0035	0036	0037	0038	0039		
Data bus	80	20	04	FF	78	02	01	19	84	0C	A9	80	34	14	33			
State	02	08	09	0B	01	02	10	11	12	01	02	06	01	02	07	01	02	14
Reset	1																	
Write	0	1															0	
ROM enable	0	1															0	
RAM enable	1	0															1	

Test program part A																	12/16/10 14:40:38	
Time	243.352 us																303.356 us	
Address bus	003A	003B	003C		003D	003E	003F	0040	0041	0042	0043	0044	0045	0046				
Data bus	10	4C	78		01		19	78	02	01	19	89	60	20				
State	04	01	02	03	04	01	02	10	11	12	01	02	10	11	12	01	02	08
Reset																	1	
Write																	0	
ROM enable																	0	
RAM enable																	1	

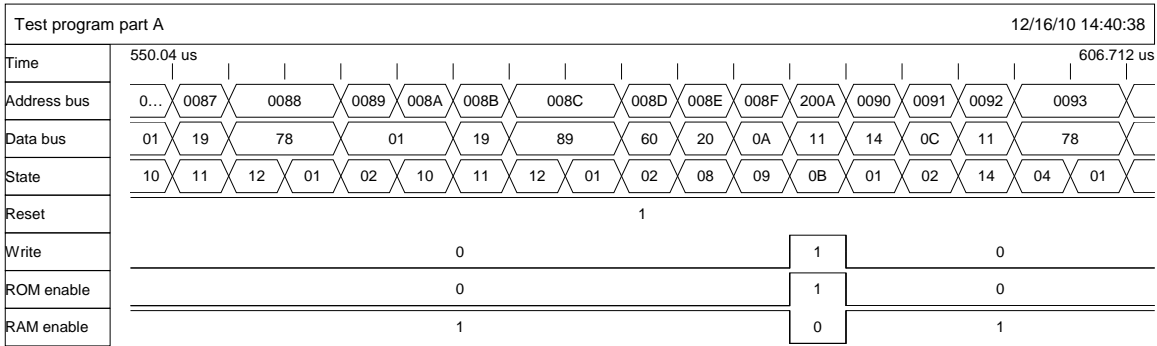
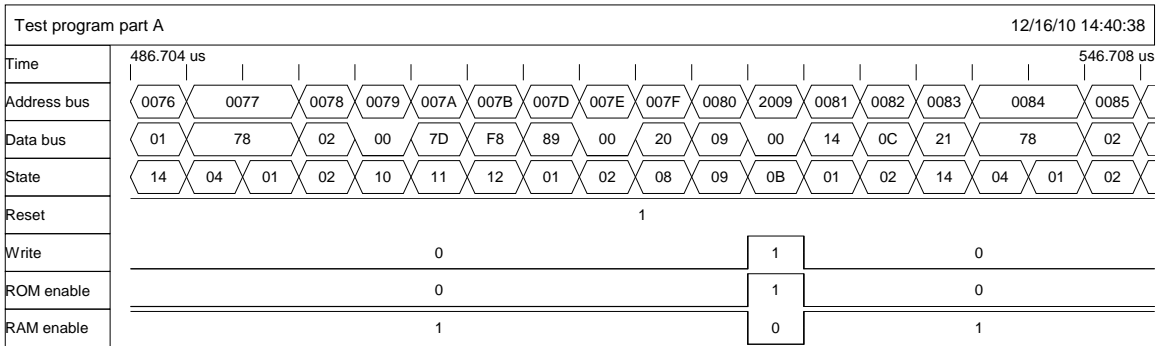
Test program part A																	12/16/10 14:40:38	
Time	306.688 us																363.36 us	
Address bus	0...	2005	0048	0049	004A	004B	2006	004C	004D	004E	004F	0050	0052	0053	0054	0055	0056	
Data bus	05	32	89	A0	20	06	77	78	00	52	F8	80	0C	00	FF			
State	09	0B	01	02	08	09	0B	01	02	10	11	12	01	02	07	01	02	14
Reset																	1	
Write	0	1	0				1	0										
ROM enable	0	1	0				1	0										
RAM enable	1	0	1				0	1										



Time 365.01721698 us to 425.845433961 us Page 7



Time 425.857412735 us to 486.685629716 us Page 8



Test program part A																12/16/10 14:40:38			
Time	610.044 us															666.716 us			
Address bus	0095	0096	0097	0098	0099	009A	009B	009D	009E	009F	00A0	200B	00A1	00A2	00A3	00A4			
Data bus	01	19	78	02	00	9D	F8	89	60	20	0B	00	14	0C	33	78			
State	10	11	12	01	02	10	11	12	01	02	08	09	0B	01	02	14	04	01	
Reset	1																		
Write																1		0	
ROM enable																1		0	
RAM enable																0		1	

Time 608.378 us to 669.20621698 us Page 11

Test program part A																12/16/10 14:40:38			
Time	670.048 us															726.72 us			
Address bus	00A5	00A6	00A7	00A8	00A9	00AA	00AB	00AC	00AE	00AF	00B0	00B1	200C	00B2	00B3	00B4	00B5		
Data bus	02	01	19	78	03	00	AE	F8	89	60	20	0C	CD	0C	14	00	78		
State	02	10	11	12	01	02	10	11	12	01	02	08	09	0B	01	02	14	04	
Reset	1																		
Write																1		0	
ROM enable																1		0	
RAM enable																0		1	

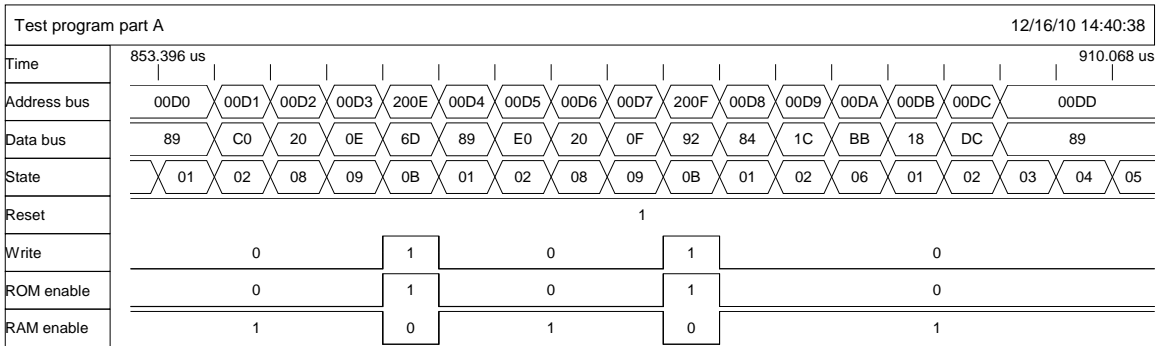
Time 669.218195755 us to 730.046412735 us Page 12

Test program part A																	12/16/10 14:40:38	
Time	733.388 us																790.056 us	
Address bus	00B5	00B6	00B7	00B8	00B9	00BA	00BB	00BC	00BD	00BE	00BF	00C0	00C1	00C2				
Data bus	78	01	19	10	94	78	02	01	19	78	03	01	19					
State	01	02	10	11	12	01	02	03	04	01	02	10	11	12	01	02	10	11
Reset																	1	
Write																	0	
ROM enable																	0	
RAM enable																	1	

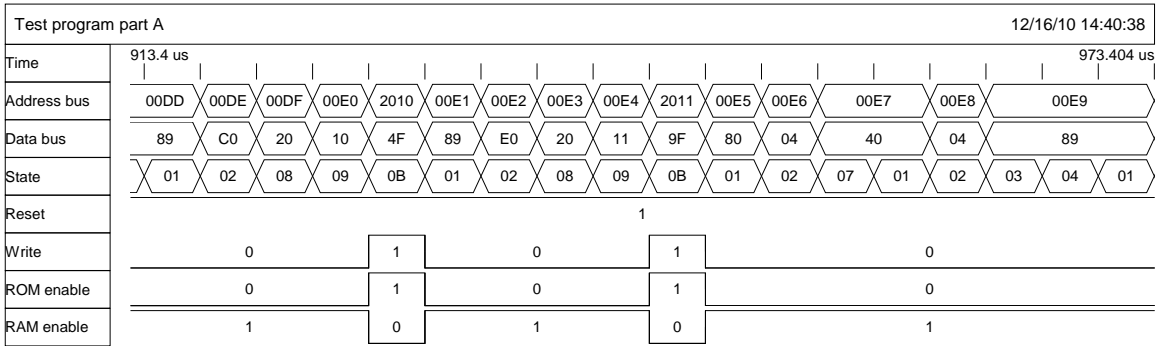
Time 730.05839151 us to 790.88660849 us Page 13

Test program part A																	12/16/10 14:40:38	
Time	793.392 us																850.064 us	
Address bus	00C3	00C4	00C5	00C6	00C7	00C8	00C9	00CA	200D	00CB	00CC	00CD	00CE	00CF	00D0			
Data bus	78	01	00	C7	89	A0	20	0D	78	80	38	1C	18	A5	89			
State	12	01	02	10	11	12	01	02	08	09	0B	01	02	07	01	02	14	04
Reset																	1	
Write																	0	
ROM enable																	0	
RAM enable																	1	

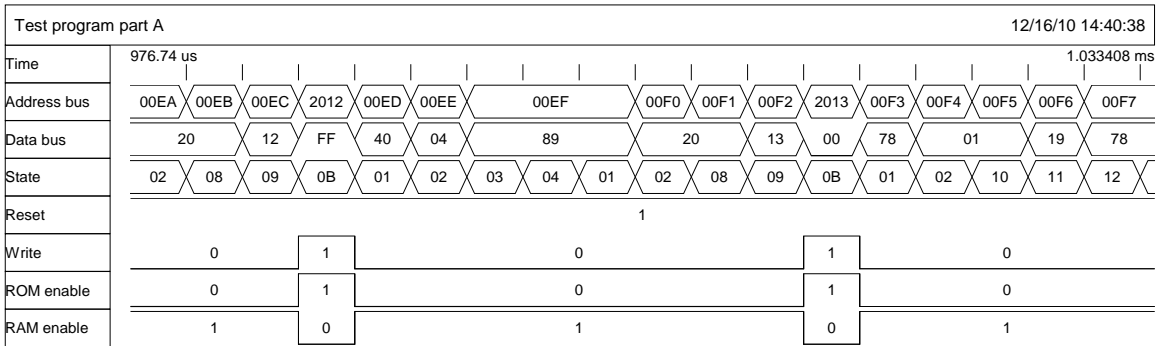
Time 790.898587265 us to 851.726804245 us Page 14



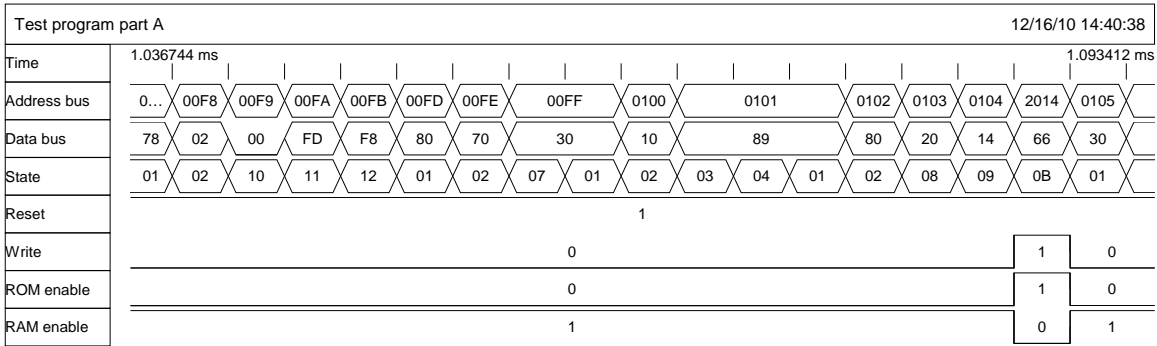
Time 851.73878302 us to 912.567 us Page 15



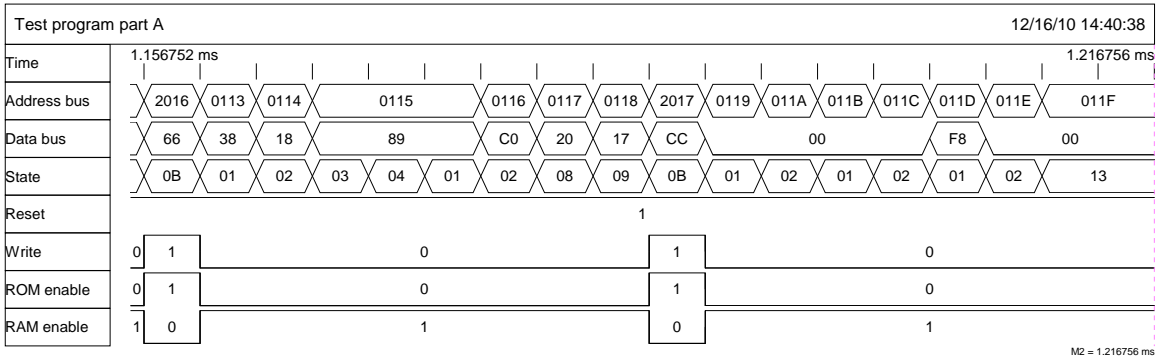
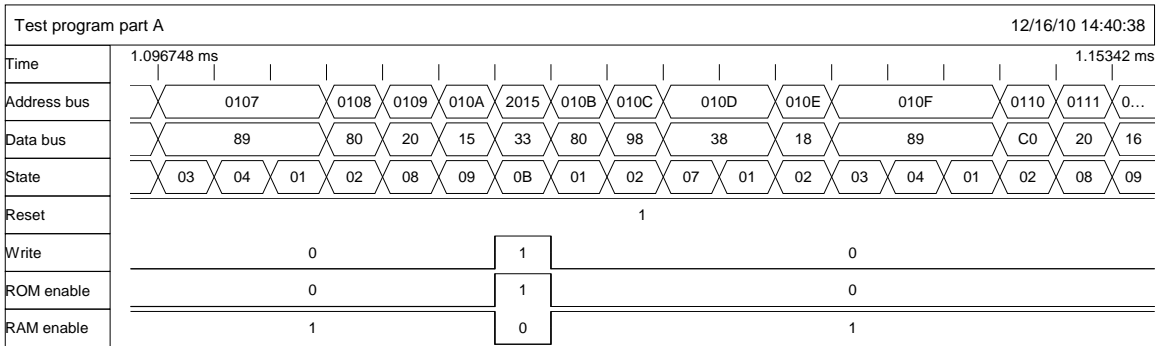
Time 912.578978775 us to 973.407195755 us Page 16

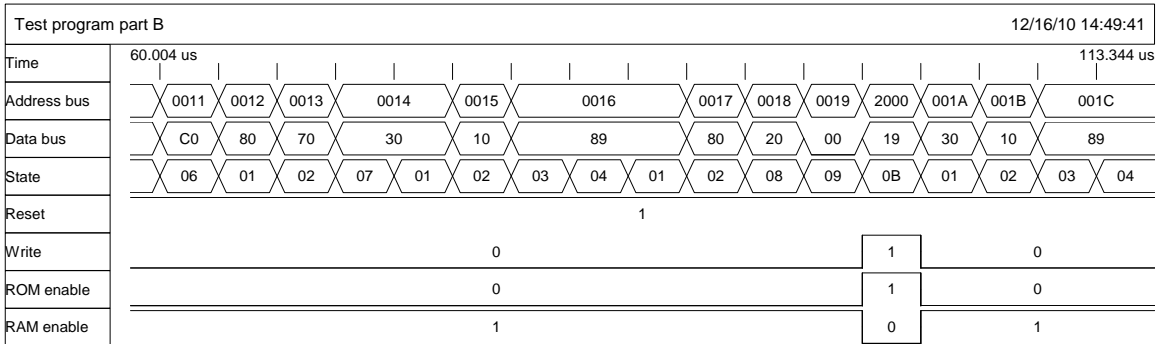
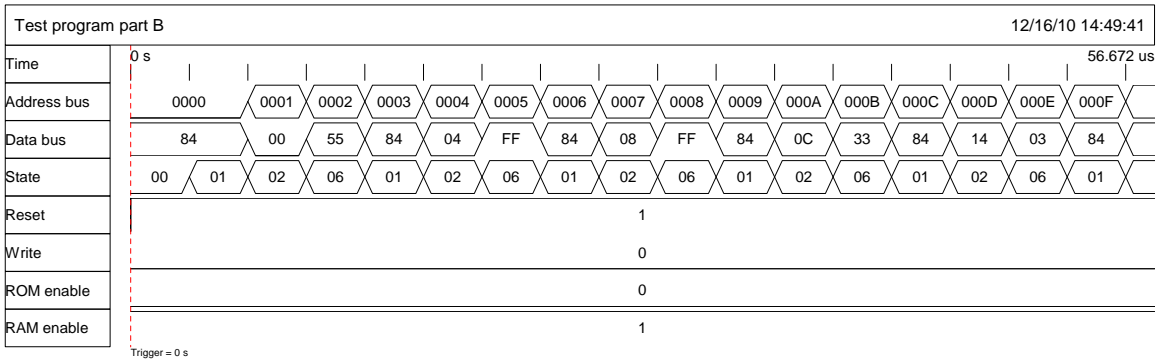


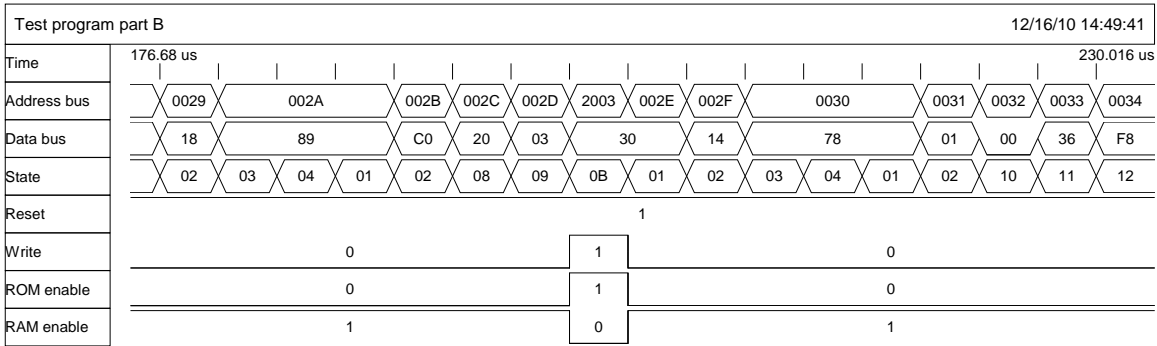
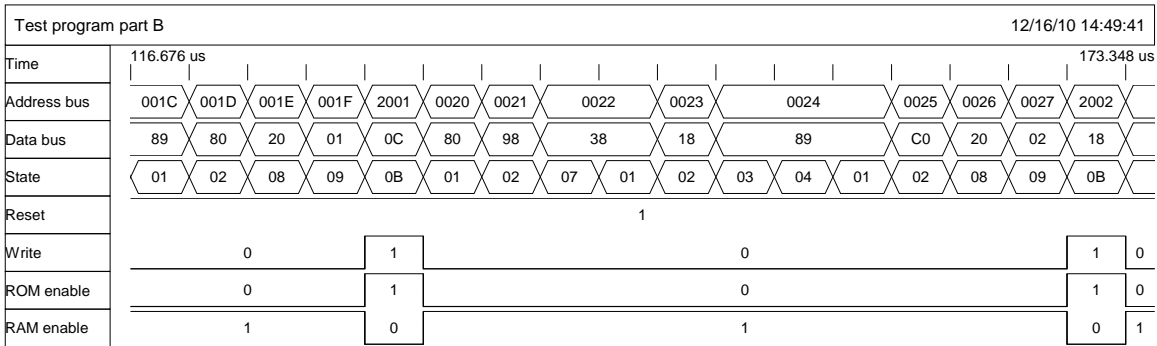
Time 973.419174529 us to 1.03424739151 ms Page 17



Time 1.034259370284 ms to 1.095087587265 ms Page 18





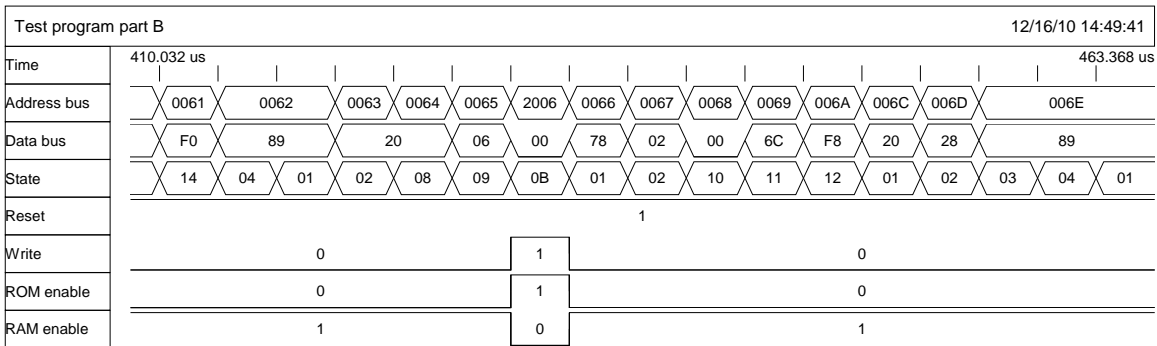
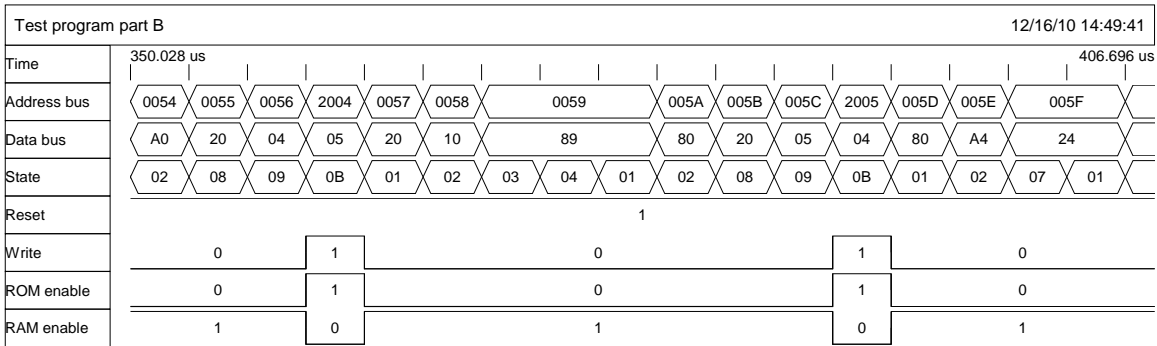


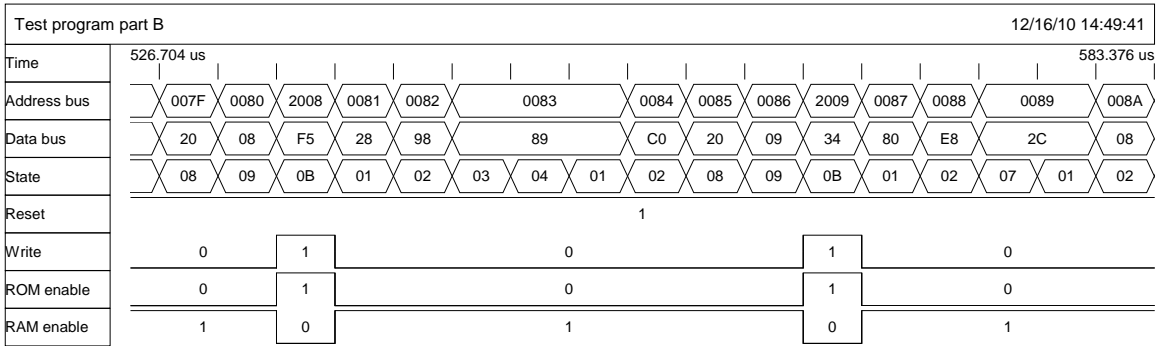
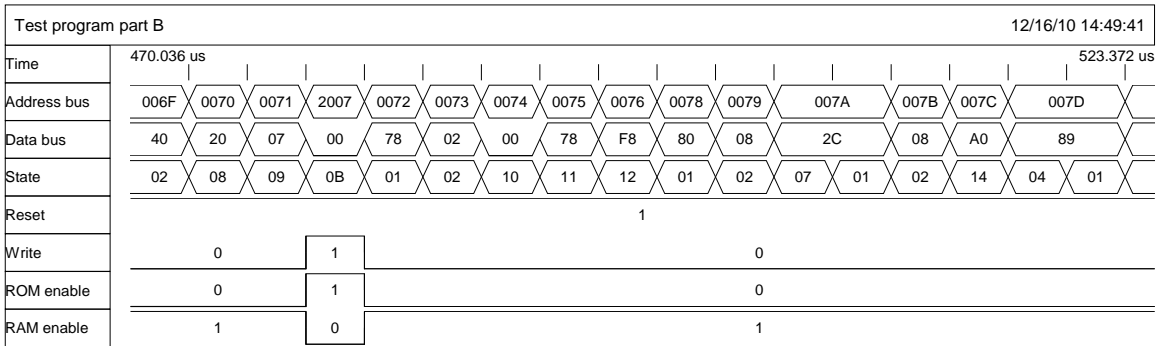
Test program part B														12/16/10 14:49:41				
Time	233.352 us														290.024 us			
Address bus	0036	0037	0038		0039	003A	003B	003C	003E	003F	0040		0041	0042	0043			
Data bus	30	14	78		02	00	3E	F8	38	1C	78		01	00	46			
State	01	02	03	04	01	02	10	11	12	01	02	03	04	01	02	10	11	
Reset	1																	
Write	0																	
ROM enable	0																	
RAM enable	1																	

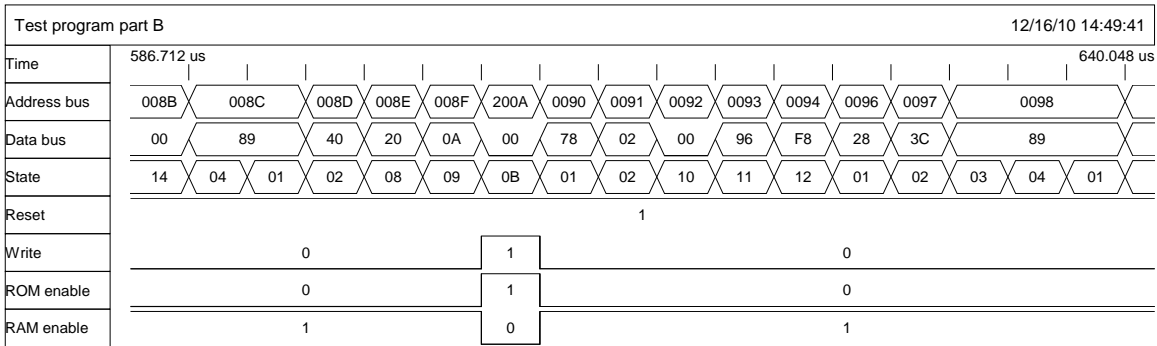
Time 233.343008795 us to 291.67301759 us Page 5

Test program part B														12/16/10 14:49:41				
Time	293.356 us														346.692 us			
Address bus	0046	0047	0048		0049	004A	004B	004C	004E	004F	0050	0051	0052	0053				
Data bus	38	1C	78		02	00	4E	F8	80	14	24	14	0F	89				
State	01	02	03	04	01	02	10	11	12	01	02	07	01	02	14	04	01	
Reset	1																	
Write	0																	
ROM enable	0																	
RAM enable	1																	

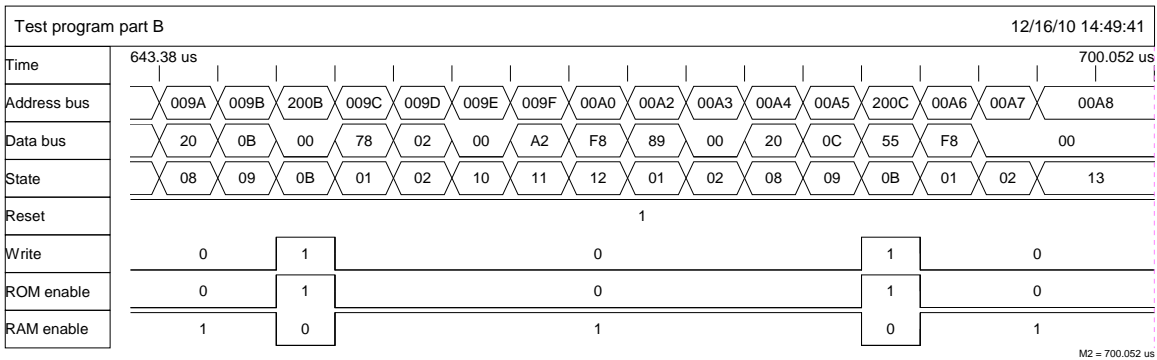
Time 291.684504397 us to 350.014513192 us Page 6







Time 583.39198241 us to 641.721991205 us Page 11



Time 641.733478013 us to 700.063486808 us Page 12