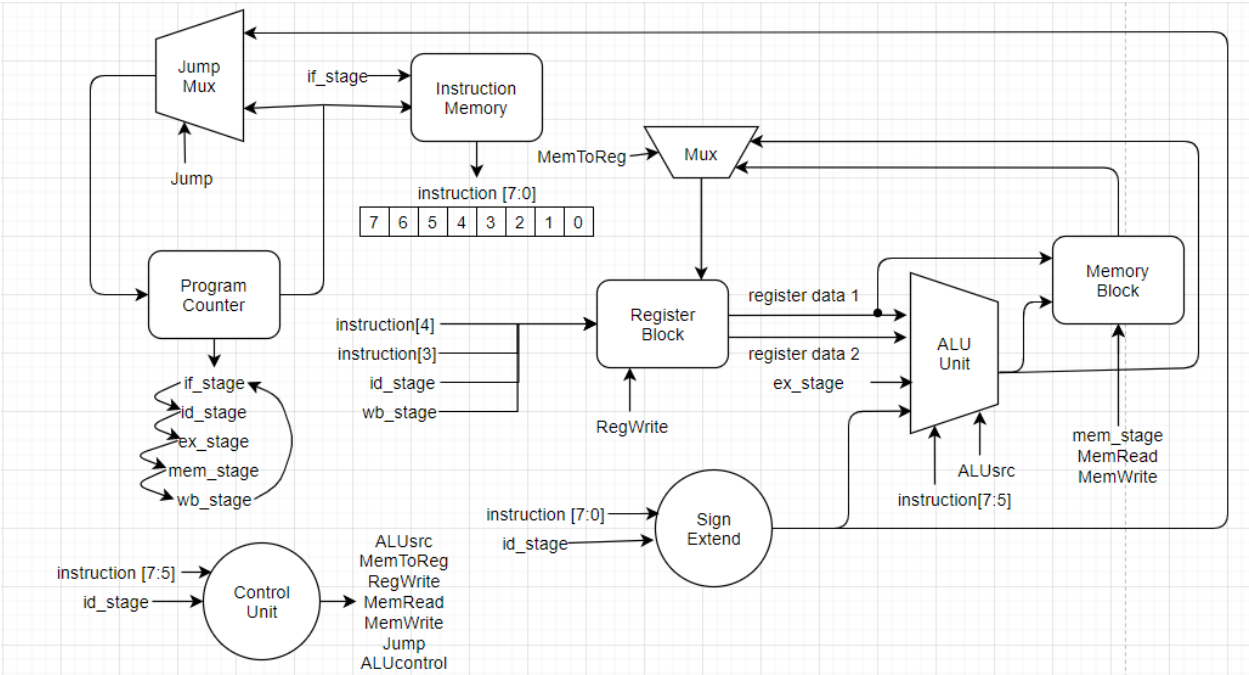
Daniel Jin
Project 1: 8 bit Non-pipelined Processor via Verilog

Final Report

Schematic



Short Description of Each Component

Program counter

This component keeps track of which stage the processor is on. Each clock hit advances the stage, but when the next stage is to be the instruction fetch stage, this module updates the program counter to be the result from the jump mux. This program counter will then be passed into instruction memory to fetch the new instruction.

Instruction memory

This component takes in the program counter and outputs the 8 bit instruction, making it available for all other components to use.

Register block

In the instruction decode stage, this module takes in bits 4 and 3 from the instruction, regardless of what the instruction actually is. The module interprets bits 4 and 3 as registers, and outputs the values of the corresponding registers. Whether or not these register values will or should be used is up to the other components and the instruction itself.

In the write back stage, this module receives an 8 bit value. If the RegWrite from the control unit is set, then the value passed into this module is written to the correct register. If RegWrite is not set, then nothing is done with this 8 bit value.

Control unit

In the instruction decode stage, this module takes in the first 3 bits of the instruction, the opcode. Based on the opcode, this module correctly sets and outputs the control signals, ALUsrc,

MemToReg, MemRead, MemWrite, Jump, ALUcontrol, RegDest and Branch. However, RegDest and Branch are unnecessary, as they will never be used in this project.

Sign extend

In the instruction decode stage, this module takes in the entire 8 bit instruction as an input. Based on the opcode, the output will differ. If the opcode indicates that the instruction is an I-type instruction (lw, sw, or addi), then it will zero extend the last 3 bits of the instruction, and output that. If the opcode indicates that it is a jump instruction, then it will zero extend the last 5 bits of the instruction, and output that. If the instruction is an R-type instruction, this module will still output something, although it will never be used.

ALU

In the execute stage, this module takes as inputs the opcode from the instruction, ALUsrc and ALUcontrol from the control unit, 2 values from registers, and 1 value from the sign extend module. Based on ALUsrc and the opcode, it will correctly compute with the proper input values. Based on ALUcontrol, it will conduct the correct operation (addition or subtraction) with the proper input values. This module then outputs the result. This result will be used by non-jump instructions.

Memory

In the memory stage, this module takes in the value from the ALU unit. Whether or not this value will be used depends on MemRead and MemWrite from the control unit. If neither are set, then it must not be an instruction that involves reading or writing memory and therefore the value passed in from the ALU unit will not be used in this module. If either MemRead or MemWrite is set, then it must mean the value from the ALU unit is a memory address.
If MemRead is set, this module outputs the value at the memory location. If MemWrite is set, then this module takes in an 8 bit value and writes it to the memory location.

Jump mux

At the write back stage, this module takes in 2 inputs. The first input is the program counter, when the module immediately adds 1 to it. The second input is the output from the sign extend module. This module is a mux, which takes in Jump from the control unit as a selector. If Jump is set, then the input from the sign extend must be the jump location, so this module outputs that. If Jump is not set, then this module outputs the incremented program counter. The address that is outputted from this module will be the address that the next instruction will be fetched from.

Simulation

For the simulation, I chose the the following instruction set:

```
addi $t0, $t0, 5
addi $t1, $t1, 1
sub $t0, $t0, $t1
sw $t1, 0($t0)
lw $t0, 5($t1)
```

```
j L1
sub $t1, $t1, $t0
L1: add $t1, $t1, $t0
```

I initialized the memory space such that the value in a memory address is double the address itself. For example, the value at memory address 8 is 16, at memory address 20 is 40, etc. The simulation produced the following results:

```
------

Program Counter:   0

Instruction: 10000101

    ADDI t0 t0 5

Before t0:   0

Before t1:   0

New value t0:   5

New value t1:   0

------

Program Counter:   1

Instruction: 10011001

    ADDI t1 t1 1

Before t0:   5

Before t1:   0

New value t0:   5

New value t1:   1

------
```

```
------

Program Counter:   2

Instruction: 10101111

    SUB t0 t0 t1

Before t0:   5

Before t1:   1

New value t0:   4

New value t1:   1

------

Program Counter:   3

Instruction: 00110000

    SW t1 0(t0)

Before t0:   4

Before t1:   1

Before storing... mem[  4] =   8

After storing... mem[  4] =   1

------

Program Counter:   4

Instruction: 00001101

    LW t0 5(t1)

Before t0:   4

Before t1:   1

Reading mem[  6] =  12

New value t0:  12

New value t1:   1
```

```
------

Program Counter:   5

Instruction: 01000111

    JUMP  7

Before t0:  12

Before t1:   1

------

Program Counter:   7

Instruction: 01110111

    ADD t1 t1 t0

Before t0:  12

Before t1:   1

New value t0:  12

New value t1:  13

------
```