Nicholas Warner (Sec 501)

Jennings Fairchild (Sec 503)

6 Dec 14

CSCE 312

**Final Project**

1. Below are the four operations supported by our ISA, with three more operations that could be enabled if needed for future use. This processor was designed as a general use processor, and so can be expanded in future iterations with up to 15 operations (instructions have a 4 bit OpCode).
   1. The CPU operates on 8 bit values, and so uses 8 bit ROM and RAM
   2. The instructions are 16 bits – 4 bits for the OpCode, 12 bits for the three registers to use or for the one 8 bit memory address and one register to use in the operation. Fetching the instruction takes two clock cycles, but allows for an easy to use, versatile ISA.
   3. OpCodes:
      1. 0000: nop
      2. 0001: halt
      3. 0010: mrmovl
      4. 0011: rmmovl
      5. 0100: addl
      6. 0101: mull
      7. 0110: irmovl
   4. Register Codes:
      1. 0000: EAX
      2. 0001: ECX
      3. 0010: EDX
      4. 0011: EBX
      5. 0100: ESI
      6. 0101: EDI
      7. 0110: ESP
      8. 0111: EBP
   5. Logisim Note: The values for the dot product to be computed from are stored in the “Ram Data” file (to be loaded into the RAM module), and the program to execute is stored in the “program” file (to be loaded into the ROM file). The RAM and ROM module were moved inside the microprocessor due to the fact that logisim would glitch and not be able to read the correct data into the processor. If this glitch did not exist, those modules would work outside the processor just as they do inside.

mrmovl

|  |  |  |  |
| --- | --- | --- | --- |
| opcode | memory address | | reg |
| x x x x | m m m m | m m m m | r r r r |

* Fetch:
  + Get next instruction from ROM
  + Send it to the instruction decoder
* Decoder:
  + Recognized as mrmovl from b15 … b12
  + Knows to grab RAM and location from b11 … b4
  + Knows it needs to go in register at b3 … b0
    - Enables that register for writing
* Fetch: get RAM at b11 … b4
* Execute: goes through ALU
* Write: write to register box, which has been opened

rmmovl

|  |  |  |  |
| --- | --- | --- | --- |
| opcode | reg | address | |
| x x x x | r r r r | m m m m | m m m m |

* Fetch:
  + Get next instruction from ROM
  + Send it to the instruction decoder
* Decoder:
  + Recognized as rmmovl from b15 … b12
  + Knows it needs to go in register at b11 … b8
    - Enables that register for writing
  + Knows to grab RAM and location from b7 … b0
* Fetch: read b11 … b8 to the data bus
* Execute: goes through ALU
* Write: write RAM at the address placed on the address bus

irmovl

|  |  |  |  |
| --- | --- | --- | --- |
| opcode | a1 | | reg |
| x x x x | ; ; ; ; | ; ; ; ; | r r r r |

* Fetch:
  + Get next instruction from ROM over 2 clock cycles
  + Send it to the instruction decoder
* Decoder:
  + Recognized as irmovl from b15 … b12
  + Knows to take explicit binary value from b11 … b4
  + Knows it needs to go in register at b3 … b0
    - Enables that register for writing
* Fetch: place immediate value on data bus
* Execute: goes through ALU
* Write: take immediate value off data bus and write it to register box

mull

|  |  |  |  |
| --- | --- | --- | --- |
| opcode | reg1 | reg2 | reg3 |
| x x x x | r r r r | r r r r | r r r r |

* Fetch:
  + Get next instruction from ROM over 2 clock cycles
  + Send it to the instruction decoder
* Decoder:
  + Recognized as mull from b15 … b12
  + Knows it needs to go in register at b11 … b8
  + Knows it needs to go in register at b7 … b4
  + Knows destination register from b3 … b0
    - Enables that register for writing
* Fetch:
  + Grab b11 … b8 and store in ALU A register
  + Grab b7 … b4 and store in ALU B register
* Execute: read A and B register into multiplier
* Write: take 16 bit answer over 2 clock cycles and store in 16 bit register with address retrieved from b3 … b0

addl

|  |  |  |  |
| --- | --- | --- | --- |
| opcode | reg1 | reg2 | reg3 |
| x x x x | r r r r | r r r r | r r r r |

* Fetch:
  + Get next instruction from ROM over 2 clock cycles
  + Send it to the instruction decoder
* Decoder:
  + Recognized as addl from b15 … b12
  + Knows it needs to go in register at b11 … b8
  + Knows it needs to go in register at b7 … b4
  + Knows destination register from b3 … b0
    - Enables that register for writing
* Fetch:
  + Grab b11 … b8 and store in ALU A register
  + Grab b7 … b4 and store in ALU B register
* Execute: read A and B register into adder
* Write: take 16 bit answer over 2 clock cycles and store in 16 bit register with address retrieved from b3 … b0

nop

|  |  |  |  |
| --- | --- | --- | --- |
| opcode |  |  |  |
| x x x x | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 |

* Fetch:
  + Get next instruction from ROM over 2 clock cycles
  + Send it to the instruction decoder
* Decoder:
  + Recognized as nop from b15 … b12
  + Other bits are irrelevant
    - Must still follow normal cycle
* Fetch: nothing
* Execute: nothing
* Write: nothing

halt

|  |  |  |  |
| --- | --- | --- | --- |
| opcode |  |  |  |
| x x x x | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 |

* Fetch:
  + Get next instruction from ROM over 2 clock cycles
  + Send it to the instruction decoder
* Decoder:
  + Recognized as halt from b15 … b12
  + Stop execution
* Fetch: nothing
* Execute: nothing
* Write: nothing

mrmovl 0, eax

mrmovl 1, ecx

mrmovl 2, edx

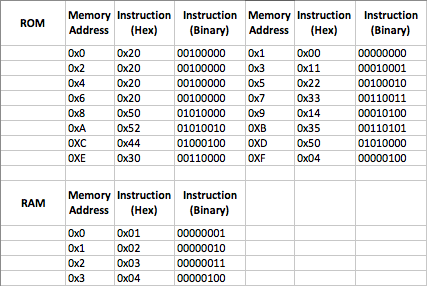
mrmovl 3, ebx

mull eax, ecx, esi

mull edx, ebx, edi

addl esi, edi. eax

rmmovl eax, return



1. See number 3.
2. *See project.circ*
3. *See project.circ*

1. *See project.circ*
2. Creating a traditional ROM chip with information you want programmed onto it is very expensive and time-consuming if done on a small scale, so using a PROM (Programmable ROM), you could initially program your ROM for individualized purposes. A PROM comes with all cells set to 1 with fuses in each cell, that if jolted with static electricity, will burn out and switch the cell’s state to 0. So, blowing the fuse of each cell to program the PROM to contain the information you would like will be programming a ROM chip.