CSCE-312-200 Final Project Lab Manual

Gabriel Britain, Jack Graham, Jonathan Innis

Division of Labor

COMPONENT	**DEVELOPER**
Arithmetic/Logic Unit	Gabriel
Register File	Jonathan
Memory Bus	Jonathan, Gabriel
Fetch	Jack
Comparator	Jonathan

ISA

OPERATION	PC + 0	PC + 1	PC + 2	PC + 3
rmmov	0x00	RA : RB		
mrmov	0x01	RA : RB		
rrmov	0x02	RA : RB		
irmov	0x03	RA : 00	CONST	
add	0x04	RA : RB		
sub	0x05	RA : RB		
cmp	0x06	RA : 00		
mult	0x07	RA:RB		
jmp	0x08	FLAG : 00	NEW_PC	
halt	0x0f			

Each cell in this table represents a single byte in ROM. The colon (`X : Y`) denotes that X is represented by the leftmost 4 bits, and Y is represented by the rightmost 4 bits, allowing two small codes to be stored in a single byte.

Flags

Flag	**Code**
AO (always on)	0
`LZ (< 0)`	1
`LE (≤ 0)`	2
`EQ (= 0)`	3
`GE (≥ 0)`	4
`GZ (> 0)`	5

Instruction Breakdown

INSTRUCTION	**ARGUMENT(S)**	**BYTE LENGTH**	**DESCRIPTION**
`rmmov`	`RA, RB`	**2**	move data from `RA` to the memory location whose address is stored in `RB`
`mrmov`	`RA, RB`	**2**	move data from the memory location whose address is stored in `RA` to `RB`
`rrmov`	`RA, RB`	**2**	move data from `RA` to `RB`
`irmov`	`CONST, RA`	**4**	move `CONST` into `RA` (**NOTE:** opposite order of x86 `irmovq`)
`add`	`RA, RB`	**2**	add value in `RA` to `RB`
`sub`	`RA, RB`	**2**	subtract value in `RA` from `RB`
`cmp`	`RA`	**2**	set flags based on value in `RA`
`jmp`	`FLAG, NEW_PC`	**4**	if `FLAG` is set, set `PC` to `NEW_PC`

Note that RA and RB are registers, FLAG is a flag, and NEW_PC and CONST are 16-bit constants.

Design Rationale

The ISA was designed with a philosophy of minimalism balanced with ease of use. The instructions can be split into 3 categories: moves, operations, and control flow.

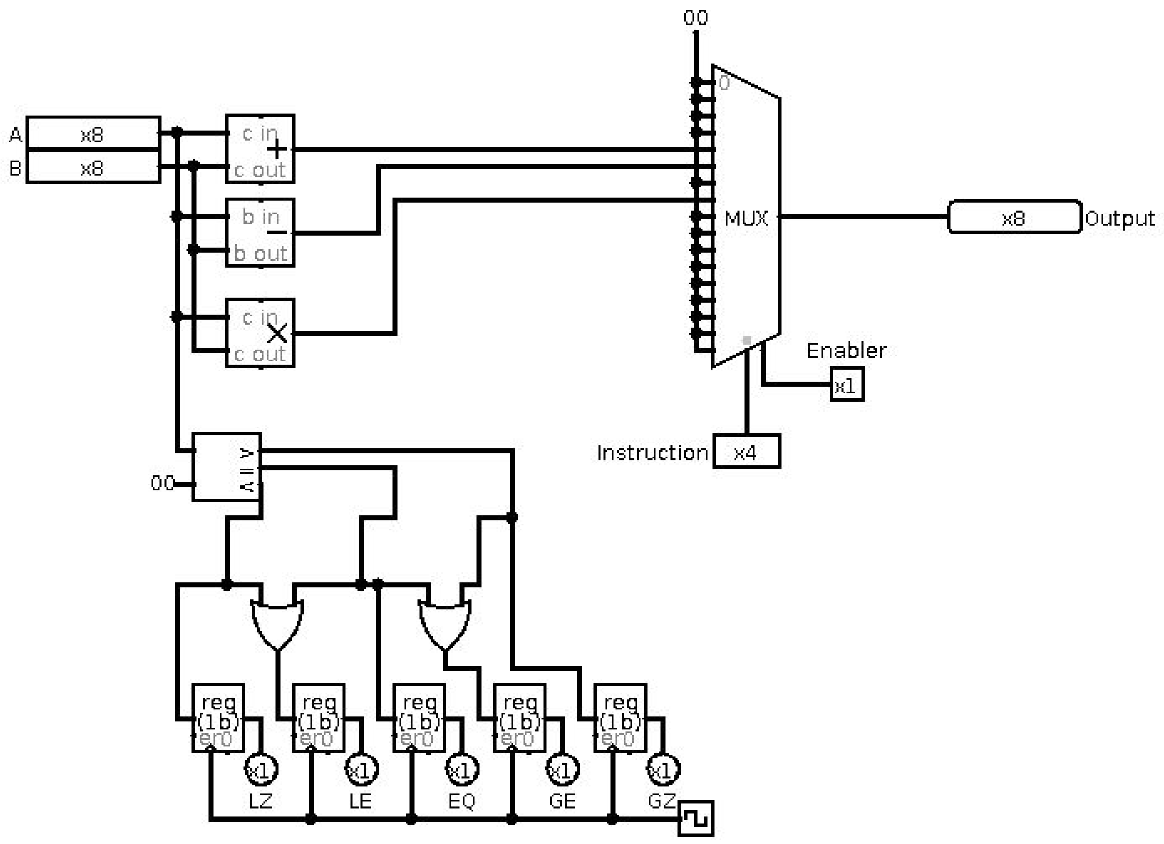
The mov instructions (`rmmov`, `mrmov`, `rrmov`) all are pretty straightforward. An interesting feature of them is that they allow constants to be passed in as either a memory address or a constant value, in the case of `rrmov`. This means that the `irmov` instruction is superfluous as "`irmov 1, ra`" is equivalent to "`rrmov 1, ra`"; as such, it does not exist in the assembler, and is only used internally to load a temporary register with a constant. In essence, the instruction "`rrmov 1, ra`" compiles into "`irmov 1, rtmp; rrmov rtmp, ra`".

The arithmetic instructions (`add`, `sub`, and `mult`) are pretty straightforward. Each takes in a register or constant, plus a register, then updates the second argument with the result of performing the specified operation on it with the first argument.

Finally, the control operations (`cmp`, `jmp`, and `halt`) are probably the most interesting. `cmp` sets the comparison flags (`lz`, `le`, `eq`, `ge`, `gz`), based on whatever is in the register passed to it. `jmp`, then, takes in a flag and a marker. A marker has to come before a jmp instruction that uses it as input, and looks like "`start:`". Then, calling an instruction like "`jmp ao, start`" will always (since `ao` is always set) jump to the instruction after the "`start:`" marker. `halt`, finally, is quite simple; it just ends the program.

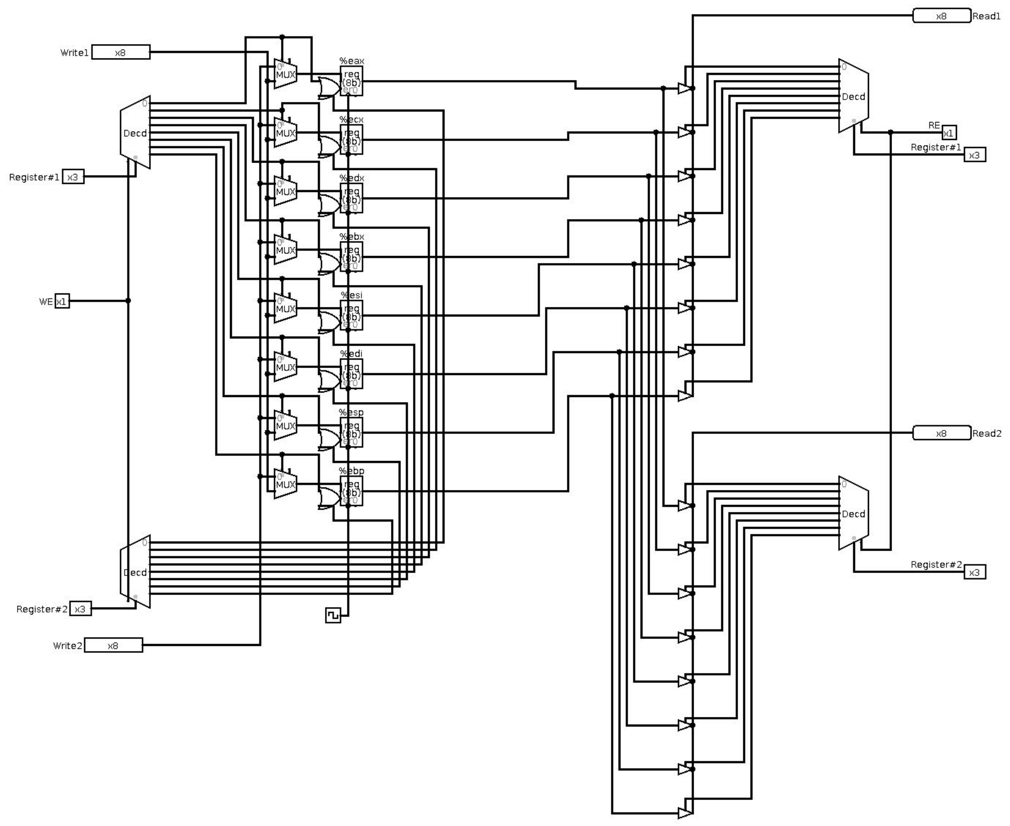
Processor Explanation

ALU



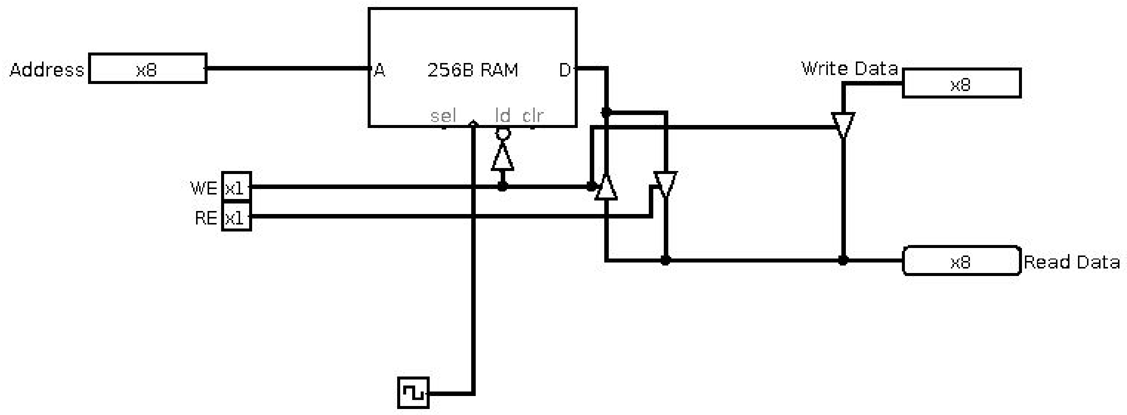
The ALU consists of four basic arithmetic functions: addition, subtraction, multiplication, and comparison, which are indicated by the built-in Logisim components. Based on the instruction provided to the 16x1 mux, the output of the corresponding arithmetic component will be sent to the output. No output will be emitted if the enabler is low. Additionally, the ALU will set flags based on the results of comparing the first operand to 0, which are used for the `cmp` command.

Register File (Jonathan)



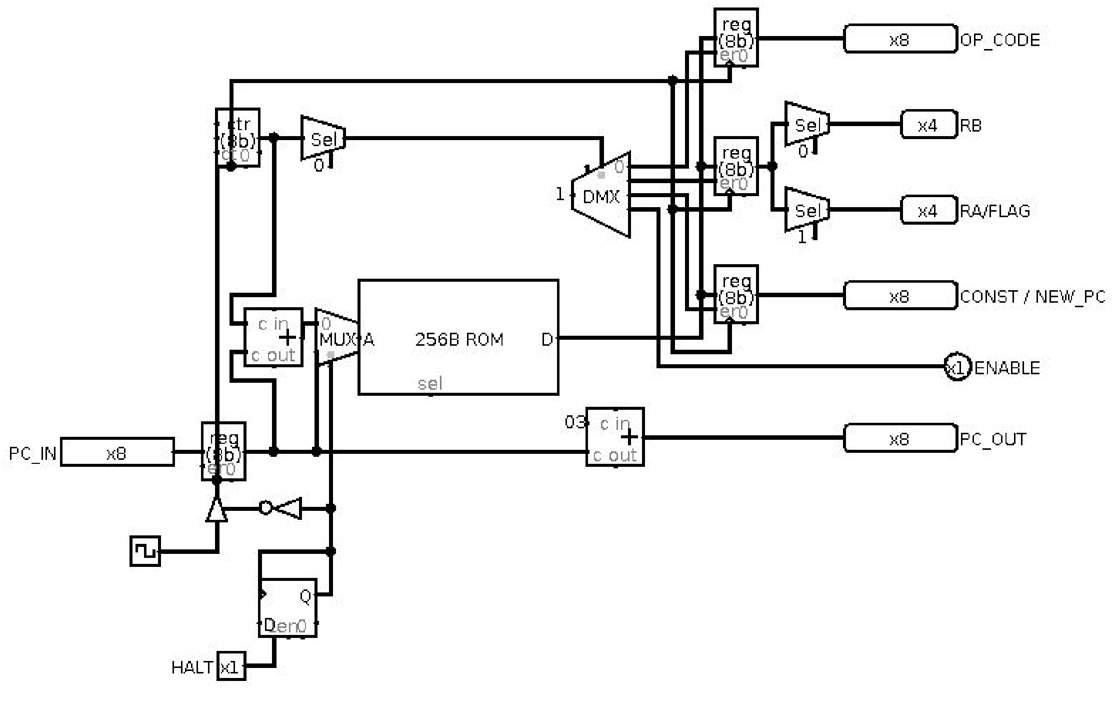
The register file stores the addresses of specific values. It can be written to by the fetch component, and can be read from by the rest of the processor. The register file works like a normal register file. It is designed with two ports for reading and two ports for writing. This works by using a multiplexer for each time that it would write to a register and then writing to that register from either port if either port enables that register through the decoder. Additionally, the two-port read works like the normal read by having a decoder that enables buffers and reads from that register to the output if that buffer is enabled through the decoder from the three-bit input.

Memory Bus (Jonathan, Gabriel)



The memory bus interfaces with RAM, which is where all of the data that is manipulated during the program is stored. When provided an address (or data) by the processor, it either retrieves or stores the value stored at that address (assuming that the respective enabler is activated).

Fetch (Jack)

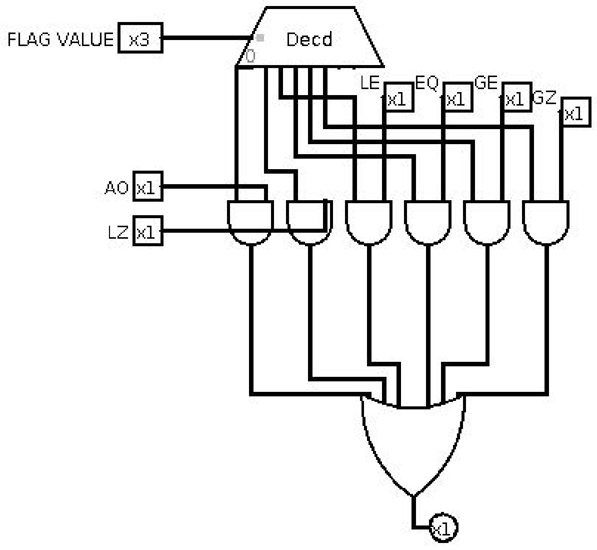


The fetch component has two inputs: HALT (true if we need to halt, otherwise false) and PC_IN. PC_IN contains the new value of PC, used to address the ROM. Each instruction is composed of 3 bytes, which are located at PC+0, PC+1, and PC+2.

The fetch component then uses the value of PC_IN to address the ROM, which contains the program instructions. Since an instruction is 3 bytes, it takes 3 clock cycles to do this; each byte is placed into its associated output. Byte 0 goes to the OP_CODE output, byte 1 is decomposed into RA/FLAG and RB outputs, and byte 2 goes into the CONST output. On the fourth clock cycle, the ENABLE output is set to 1; this signals to the other components that all of the outputs have been loaded with the instructions, and are thus ready for use. Finally, the clock rolls over back to 0, to begin another fetch cycle.

Additionally, the PC_OUT is set to PC_IN + 3; this can be overridden in the case of a jmp instruction whose flag is set to TRUE, but usually is directly routed to PC_IN the next cycle.

Comparator (Jonathan)



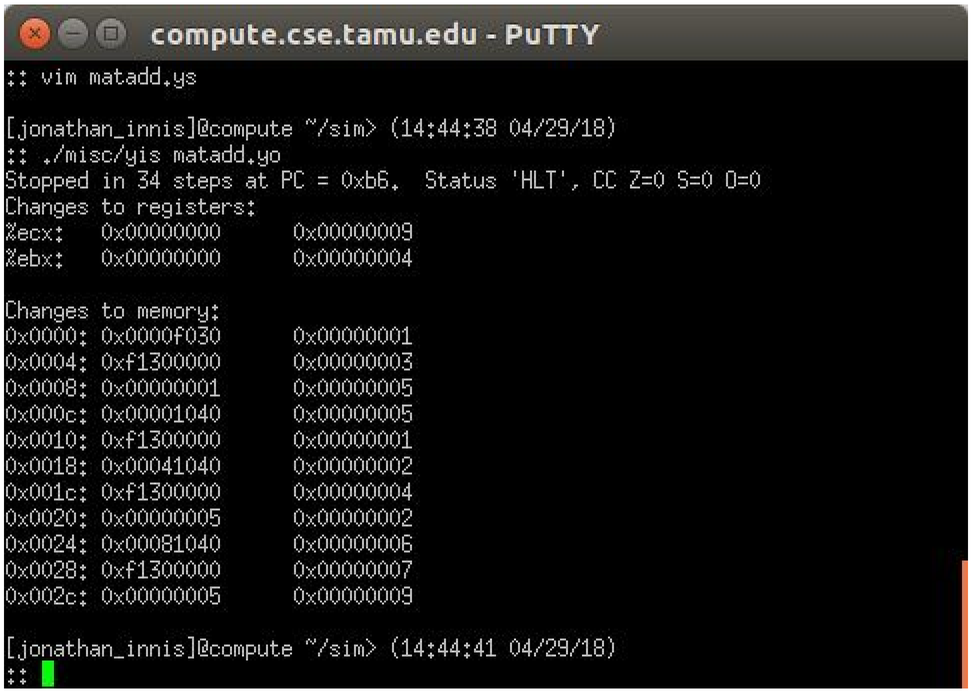
The comparator part of the processor receives the flag values, and decodes that value, and compares it to the appropriate code, to indicate his It reads in three-bit input that is then decoded. Each part of the decoder corresponds a flag value that will be checked. If the flag that has been selected is on, the decoder for that selected flag will be on as well as the flag itself meaning that the AND will evaluate to TRUE. Thus, if any of the AND gates evaluate to true, the OR gate will also evaluate to TRUE enabling the jump that was called to occur. The comparator is only used when the instruction that is passed from FETCH is 0x08.

Testing

For both the matrix multiplication and the matrix addition, we created an assembler in python that would take assembly code and translate that code into the necessary binary for our processor. We would take a file that is demaracted as `*.ildm` and translate that into `*.bongle`. This bongle can then be loaded into the ROM in the processor so that the processor can execute the task that it has been given. For the `matadd.bongle`, the matrix A contains the values [1,3; 5,5] and the matrix B contains the values [1,3; 2,4]. Thus, when we add these values together in the processor, we expect to see [2,6; 7,9]. For the `matmult.bongle`, the matrix A contains the values [1,3; 5,5] and the matrix B contains the values [1,3; 2,4]. Thus, when we multiply the values together in the process, we expect to see [7,15; 15, 35] which we do see in the processor when we translate from hexadecimal to decimal.

Y86 Assembly Code

For the assembly code that is included in this lab, we decided to use four bytes for each constant in memory because of the way that Y86 takes values out of memory. Because Y86 pull values out of memory four bytes at a time, putting values every four bytes ensures that the values don't overflow into each other. For matrix A, we assumed that the matrix contained the values [1,3; 5,5] as in the matrix A for the `matadd.bongle` file. For matrix B, we assume that the matrix contained the values [1,3; 2,4] as in the matrix B of the `matadd.bongle` file. Thus, the matrix that will be produced by matrix C will be [2,6; 7,9].



```
:: vim matadd.ys

[jonathan_innis]@compute ~/sim> (14:44:38 04/29/18)
:: ./misc/yis matadd.yo
Stopped in 34 steps at PC = 0xb6.  Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%ecx:   0x00000000      0x00000009
%ebx:   0x00000000      0x00000004

Changes to memory:
0x0000: 0x0000f030      0x00000001
0x0004: 0xf1300000      0x00000003
0x0008: 0x00000001      0x00000005
0x000c: 0x00001040      0x00000005
0x0010: 0xf1300000      0x00000001
0x0018: 0x00041040      0x00000002
0x001c: 0xf1300000      0x00000004
0x0020: 0x00000005      0x00000002
0x0024: 0x00081040      0x00000006
0x0028: 0xf1300000      0x00000007
0x002c: 0x00000005      0x00000009

[jonathan_innis]@compute ~/sim> (14:44:41 04/29/18)
::
```