CPU Lab

Hashim Chaudhry - 3762

November 2021

mARMi 16

mARMi 16 is a 16 bit minimal ARM-like CPU that is able to execute load, store, conditional, arithmetic, and logical instructions. The following sections are intended to provide documentation on the micro-architecture of the CPU's hardware along with providing example assembly and machine files that can be used to test the CPU's various instructions. The full CPU schematic is shown below.

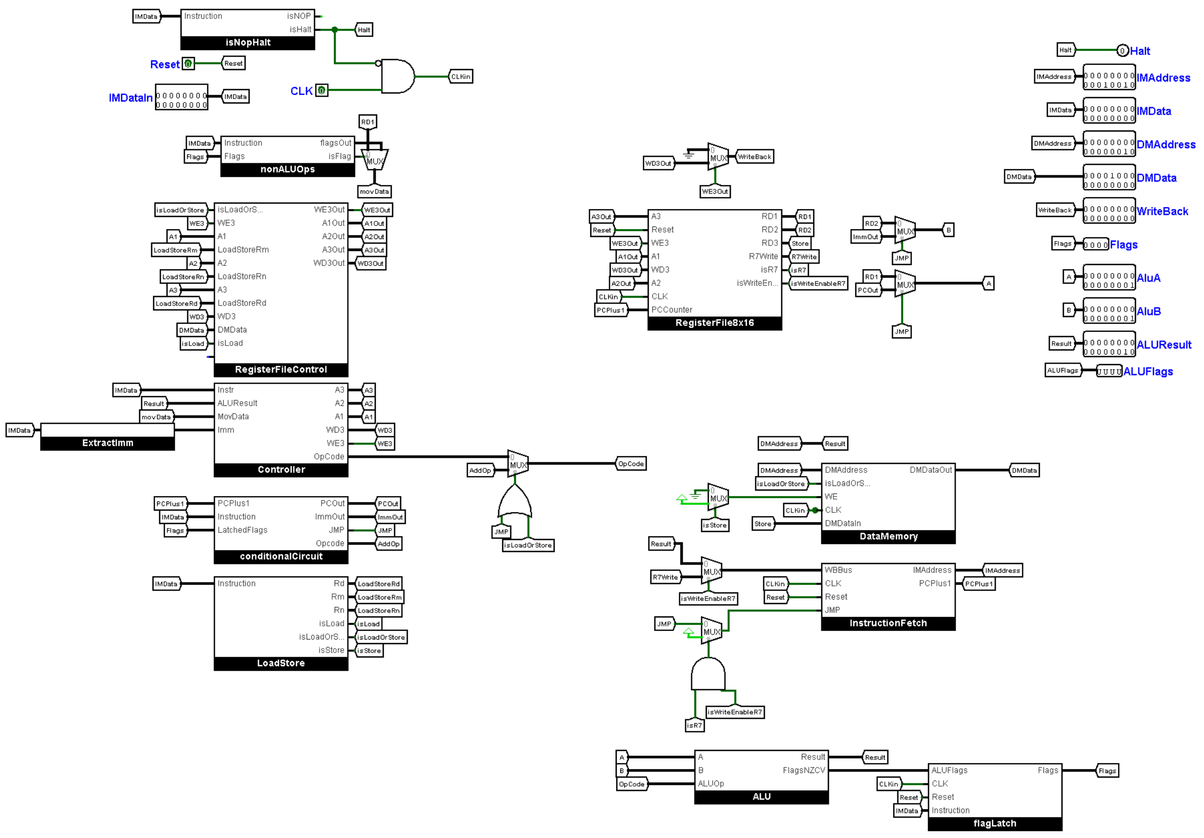


Figure 1: *mARMi 16 CPU*

mARMi consists of 4 main components: a register file, an instruction fetch unit, a Control Unit, and an ALU.

Instruction Fetch

The instruction fetch unit is designed to fetch the current instruction address being executed from ROM and then store it in a program counter, called $IMAddress$. We also provide a separate output called $PCPlus1$, which provides the address of the next instruction, or $IMAddress + 1$, and is routed to register $R7$ in our register file.

The instruction fetch takes in 3 inputs: a system wide clock input, a system wide reset input, a JMP flag, and a $WBBus$ input. $WBBus$ supplies an address based on if a branch instruction was executed. The JMP input acts as a flag. If JMP is HIGH, then we allow $WBBus$ to be accepted as the next instruction in our program counter. Otherwise, we use *IMAddress + 1* as the next instruction.

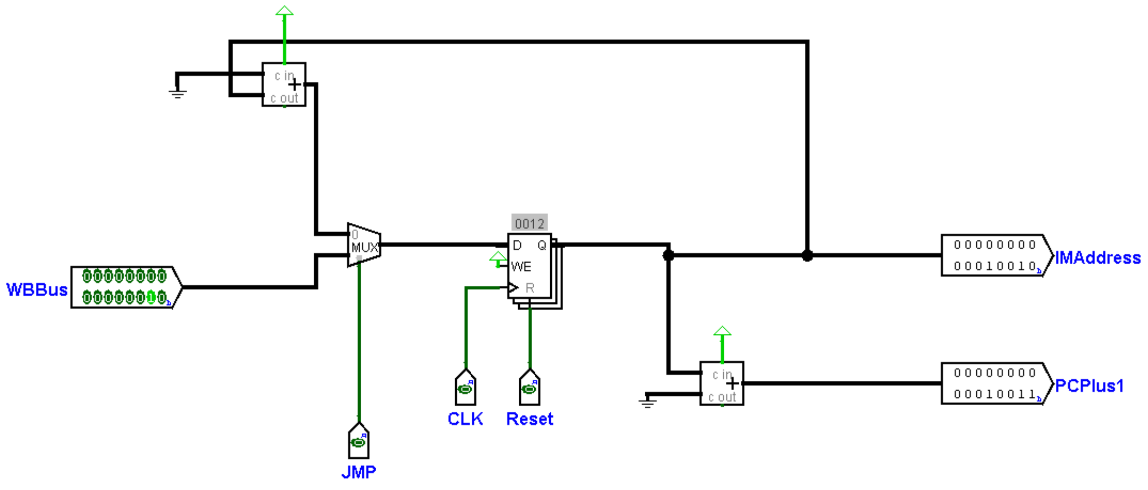


Figure 2: *Instruction Fetch Unit*

The write enable to this register is always high, as it is updated every clock cycle. Computing *IMAddress + 1* is done by adding the current address to 0 and providing a carry-in that is always high.

Arithmetic and Logic Unit

The arithmetic and logic unit (ALU) is a unit that performs all arithmetic and logical instructions. The ALU selects the correct value to output on *Result* based by using a multiplexer with *ALUOp* as the selector. The *ALUOp* bits are the 4 bits after the first 2 most significant bits on an ALU instruction. All inputs to the ALU are controlled by our control unit and register file. The ALU uses two sub circuits: a FlagController circuit and an ALU shifter circuit. The FlagController circuit computes the carry and overflow flags and the ALU shifter computes the shift output. For shift instructions, the lower 4 bits of input B are extracted to tell us the shift amount.

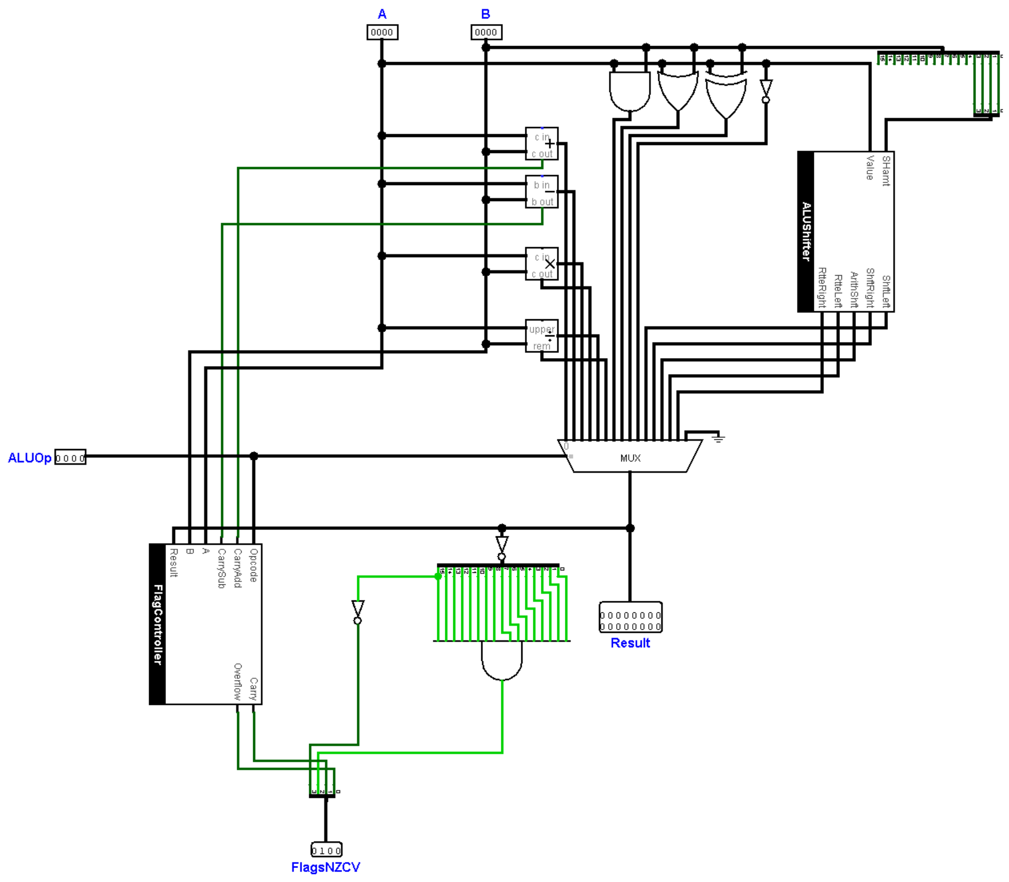


Figure 3: *Arithmetic and Logic Unit*

ALUShifter

The ALUShifter circuit below computes the a shift on A based on the shift amount *SHAmt*. We then compute each shift output them on their respective lines. These outputs are then routed to the final multiplexer on the ALU.

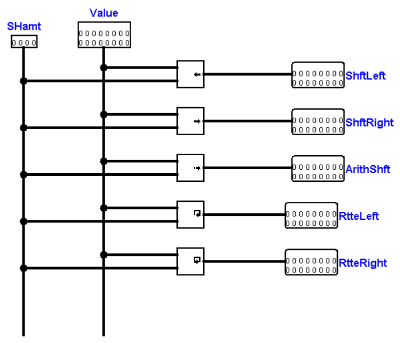


Figure 4: *Shift Unit for ALU*

FlagController

The FlagController unit uses a more complex circuit to compute carry and overflow flags for addition and subtraction. It takes in the opcode of the CPU to determine whether an ADD or SUB instruction was called, along with A, B, *Result*, and the carry outs from the adder and subtractor.

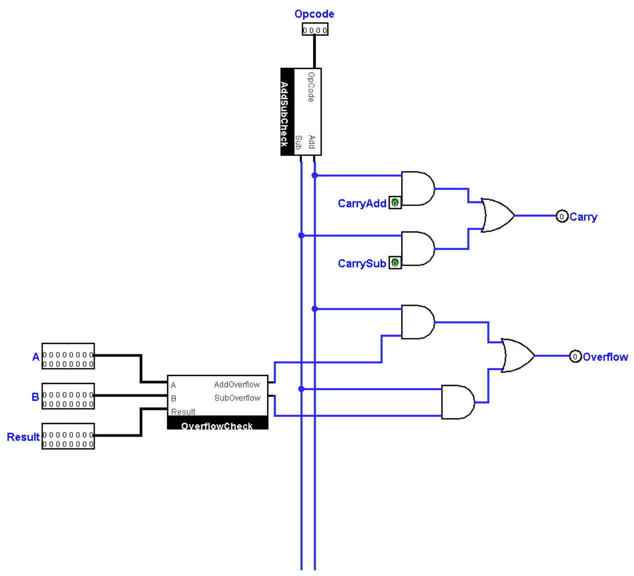


Figure 5: *Flag Controller for ALU*

The AddSubCheck unit allows us to mask the carry and overflow flags on all instructions other than ADD or SUB and also allows us to correctly output carry and overflow based on whether we executed an ADD or SUB instruction.

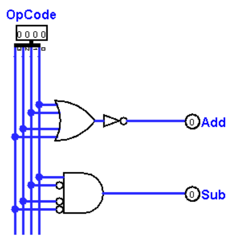


Figure 6: *AddSubCheck unit in FlagController*

The overflow check unit checks an overflow based on the values of A, B, and *Result*. It has two outputs, an overflow flag for ADD and another for SUB, which are tied to the final overflow flag output for FlagController.An overflow is defined for addition when the sign of the two operands is the same but the result sign is different, and for subtraction it is defined as when the operands have different signs and the result has the same sign as the subtrahend. The sign bit in two's complement is the high bit and, as such, is extracted in OverflowCheck.

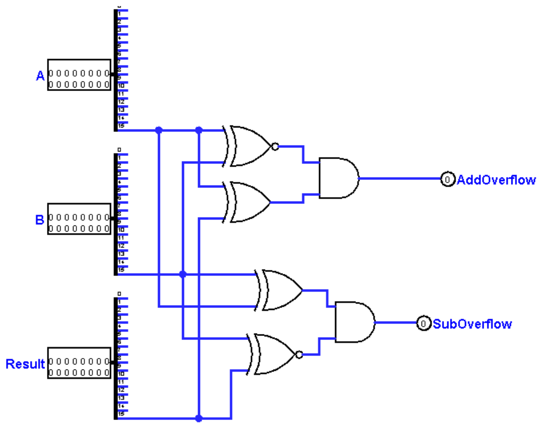


Figure 7: *OverflowCheck unit in FlagController*

The final flags, *Zero* and *Negative*, are computed using gates. The *Zero* flag checks whether each result bit is zero via an AND gate, and the *Negative* flag is set if the high bit of result is set. This is done directly on the ALU.

Latched Flags

An important part of our ALU is properly supplying arithmetic operation flags so that conditional branches work correctly. However, we only want our flags to be set only when an ALU operation is executed. (Branch instruction don't change the latched flags even though they go through the ALU). As such, a dedicated register keeps track of these flags.

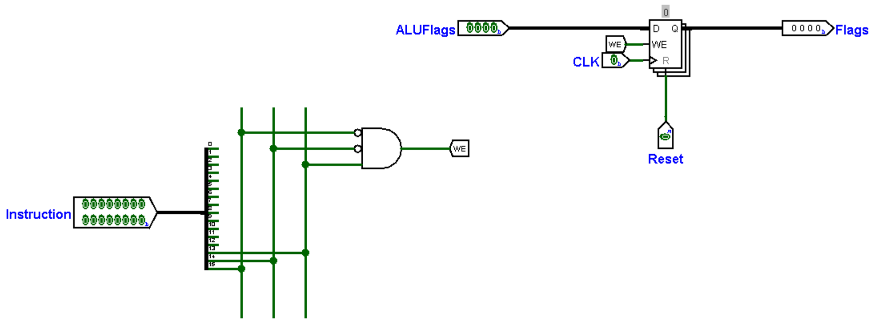


Figure 8: *FlagLatch unit*

The Flag Latch circuit allows uses the higher 3 bits of the instruction read from memory to determine whether an ALU instruction was encountered. This acts as a write enable for our flag latch register. If it is enabled, the flags from the ALU are saved and are otherwise thrown away. We also provide a system wide reset and clock for this unit.

Register File

The register file is an 8x16 unit that saves values from the ALU and stores loaded values from memory. It takes 3 three bit address inputs, *A1*, *A2*, and *A3*, where *A3*, is largely intended to be read on *RD3* only for STR instruction and written for writebacks on *WD3* for other operations. We can write to *A3* based on whether or not *WE3* is HIGH, in which case the value on *WD3* is added to the appropriate register. The coordination of these inputs is largely controlled by the Control Unit, which is discussed later. Demultiplexers are largely used to write to correct registers and multiplexers are used to read from correct registers based on the addresses supplied.

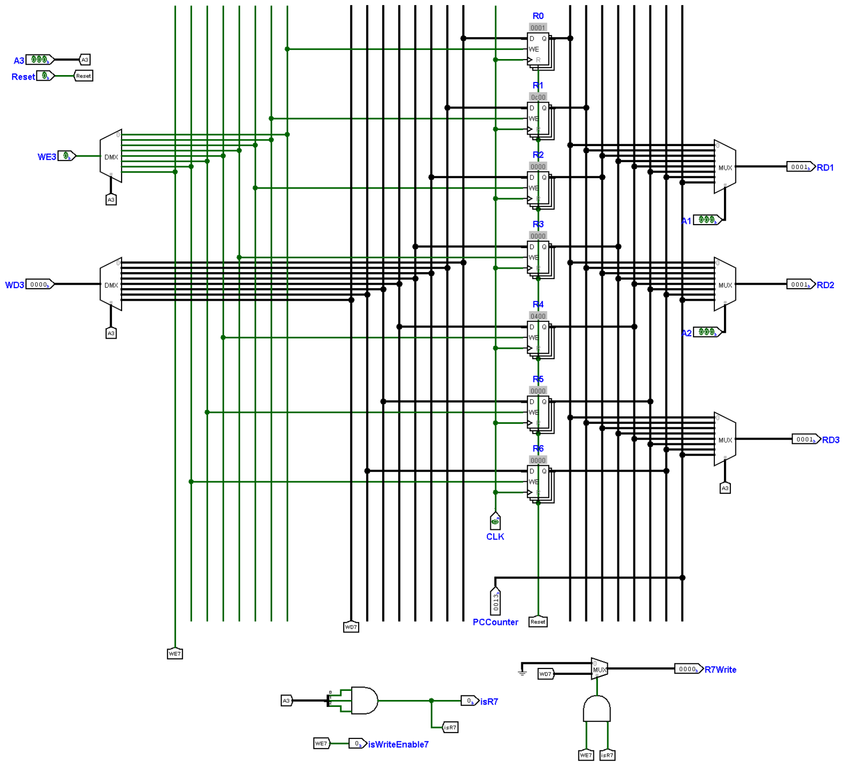


Figure 9: *Register File for CPU*

Register *R7* is a special register that is reserved for the program counter. However, to allow additional functionality to the mARMi CPU, we allow reads and write to the program counter. The *R7Write* output checks to see if the write enable is on and if we are accessing register *R7*, and if so, outputs the data on *R7* write. The output *isR7* is an output flag that lets other outside components, notably the instruction register and writeback paths, in the CPU know that we are writing to the program counter. Reading from *R7* is done by the multiplexers on the output, which will select the correct register based on the read address.

Data Memory (RAM)

The data memory module for our CPU acts as 64K RAM for storing values out of registers asynchronously. It accepts a write enable input that allows writes, a *DataIn* value to write, and a *isLoadOrStore* flag that controls a multiplexer and determines whether the address is the result of an LDR or STR instruction. *DMDataOut* simply outputs the data value being asked for based on DMAddress. Note that the output enable is always true in this module, as the Control Unit has a flag that determines whether the output data will be writtin back to the register file from RAM.

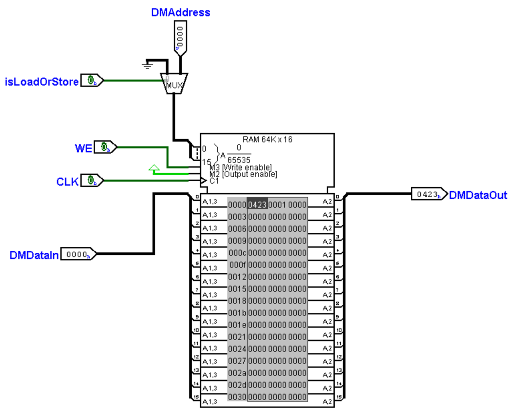


Figure 10: *Data Memory module for CPU*

NOP and HALT

The isNopHalt module tests for NOP instructions and HALT instructions and outputs a flag on the respective output. The NOP instruction flag is not used in this CPU as the NOP instruction does not activate any units in the CPU. It is, however, provided for convenience should later added modules in the CPU require such a flag. The Halt flag is used to control the CPU's access to the clock.

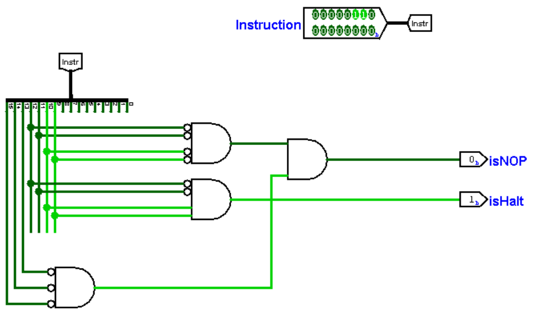


Figure 11: *Module testing for NOPs and Halts*

Control Unit

The Control Unit acts as a central brain for decoding instructions and properly setting up other components in the CPU for the instruction. It is a key junction for all instructions and data routing in the mARMi CPU. We consider the *ID* of an instruction to be the high 4 bits, and the *MSB* to the highest order bit. Our control unit simply extracts needed input values from the instructions and routes them into various subcircuits. Each subcircuit then determines which address that needs to be outputted, along with any signal flags for the CPU.

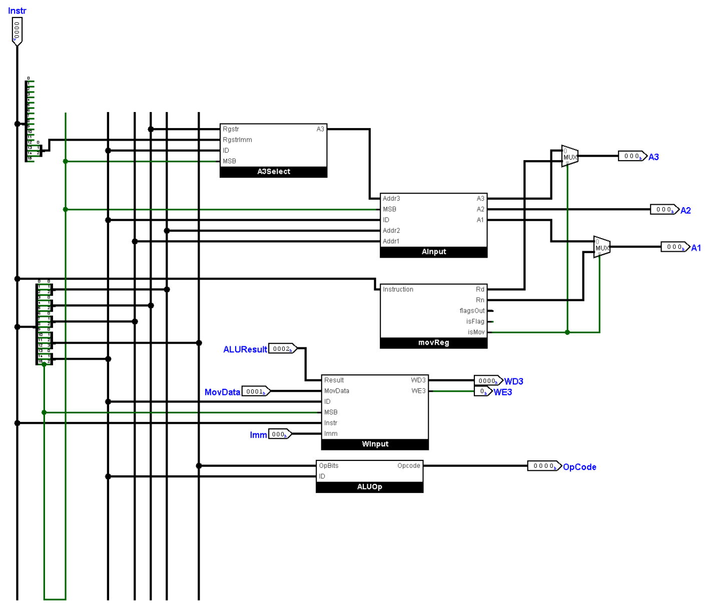


Figure 12: *Controller*

Each output on the controller corresponds to a value on the register file, except for *OpCode*, which instead outputs an OpCode for the ALU based on whether or not the value an ALU instruction was read. The control units complex circuitry is compartmentalized into various subcircuits.

A3Select

The output value for the address on *A3* varies based on the instruction. If the value was a MOV immediate instruction, tested by looking at the MSB, then *A3* outputs the destination register input, called *RgstrImm*. This is extracted in the control unit directly using splitters. Otherwise, if the operation was an ALU operation, we simply output the ALU destination register in the instruction that corresponds to *A3*. This again is also extracted in the Control Unit via splitters.

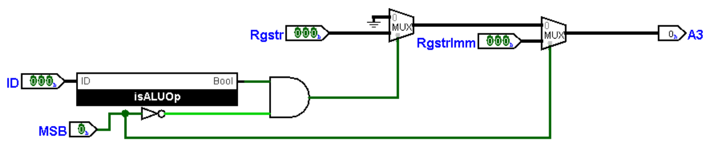


Figure 13: *Circuit to correctly extract A3*

isALUOp

This is a subcircuit that tests whether the bits in the instructions constitute an ALU operation, and if so, sets a flag. It is critical for circuits such as the is Latched Flags unit.

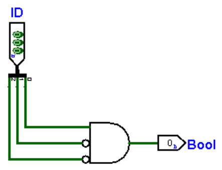


Figure 14: *Circuit that tests whether instruction is an ALU operation*

AInput

The AInput subcircuit takes in the 3 high order bits and the most significant bit and determines the outputs for *A1, A2,* and *A3* based on the instruction. The outputs of this subcircuit are routed to the controller output for register addresses. The input *A3* for this unit is directly tied to the output of *A3Select*, and it allows for *A3* to pass if the instruction operations signified a MOV immediate or ALU operation. *A1* and *A2* are only outputted for ALU operations. Otherwise, they are masked.

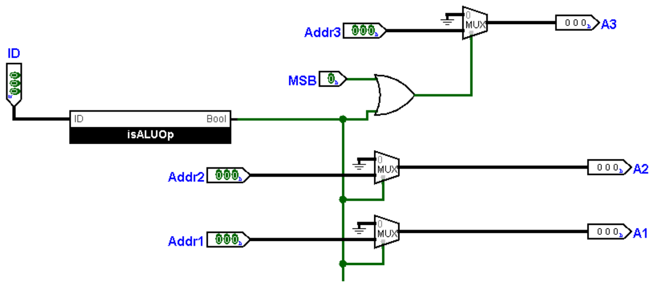


Figure 15: *Circuit that routes inputs for register addresses*

movReg

The movReg unit tests to see if we are moving data from one register to another, and, if so, outputs the register addresses it has extracted from the instructions as the address lines. It also outputs flags such as *flagsOut*, *isFlag*, and *isMov* to control the multiplexers in the main control circuit and CPU. The *flagsOut* and *isFlag* flags are intneded to notify the controller that we are moving the ALU latched flags into a register. There are two *MOV* instructions, both of which have the same location for *Rd* and so no logic is needed to pick the correct address. However, the value for *Rn* depends on whether we are moving values between registers or moving the ALU latched flags. The multiplexers on the *Rn* output test this condition based on looking directly at the instruction opcode and then use multiplexers to correctly route the data.

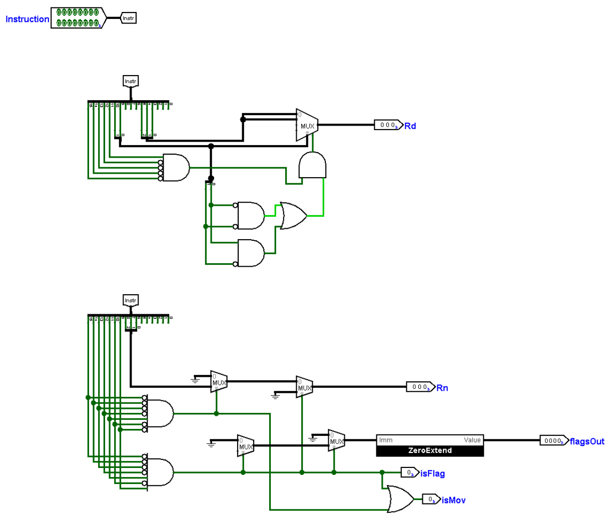


Figure 16: *Unit that test to see if instruction is simply moving register values*

ZeroExtend

The zero extend function takes a 4 bit value and zero extends it. This is intended to be used to sign extend the ALU latched flags when moving ALU flags to a register.

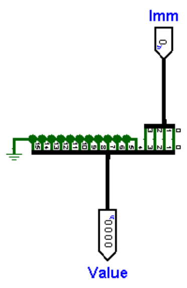


Figure 17: *Zero extend circuit*

WInput

This subcircuit determines whether a write enable will turn on and what data should be written if needed. We first test to see if the instruction is an ALU operation, and if so output the correct write data and enable. We also test for MOV immediate instructions, in which case *WD3* is instead the immediate value of the MOV instruction. Otherwise, if the instruction is a MOV register instruction, we select the

MOV data. This is done via multiplexers, where we test the MSB of the instruction for MOV immediate instructions and the *ID* for ALU operations and then select the correct output. In either of these cases, the write enable should be HIGH. We also sign extend our 12 bit immediate value before it is routed.

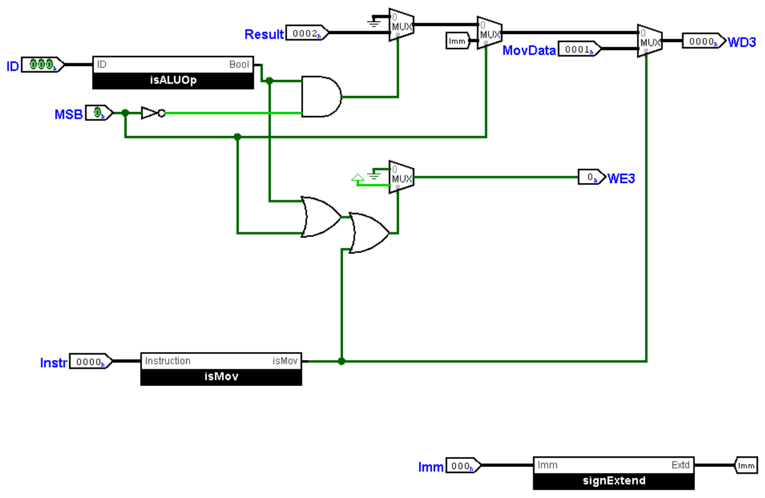


Figure 18: *Circuit determining write status*

SignExtend

The sign extend subciruit is used to sign extend immediate values before put into a register or used for other operations. It uses a multiplexer based on the high bit to determine whether to extend with 1's or 0's for the high 4 bits.

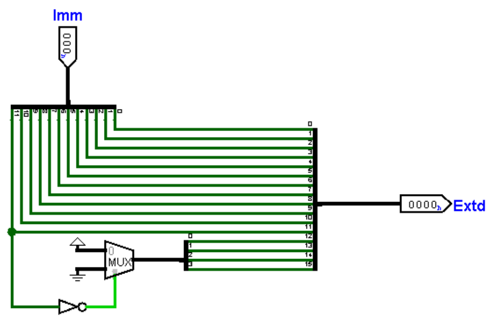


Figure 19: *Sign extend circuit*

isMov

Tests to see whether an instruction intends to move data between two registers. This circuit directly tests the bits necessary to determine whether an operation is a MOV register operation. This unit does not test for MOV immediate instructions.

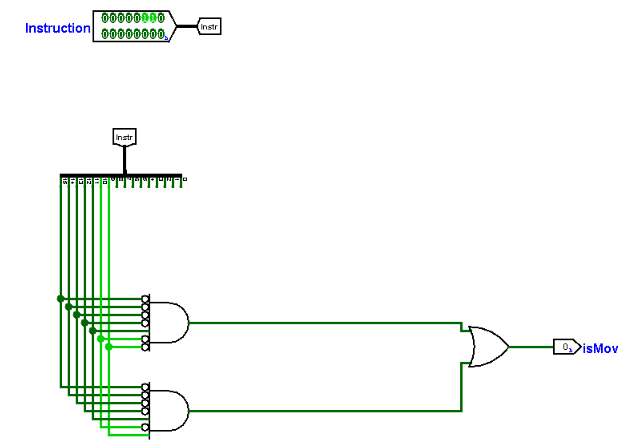


Figure 20: *Circuit that test for MOV register instruction*

ALUOp

This circuit outputs the ALU operation if our instruction is an ALU operation. In such a case, the multiplexer outputs the ALU operation; otherwise, the output of the ALU is 0000. We mask the ALUOp outside the controller if the operation does not require the ALU.

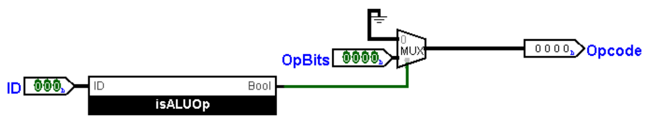


Figure 21: *Circuit that extracts ALU operation if instruction requires ALU*

LoadStore

This circuit is intended to handle memory accesses to and from RAM. It test for the LDR and STR bits, and outputs boolean values on the *isLoad*, *isStore*, and *isLoadOrStore* flags. It also extracts the necessary register addresses if the instruction is LDR or STR and outputs them using multiplexers. The flag outputs are used on multiplexers in the CPU.

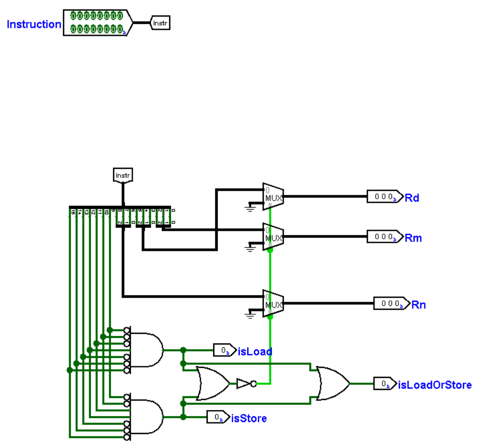


Figure 22: *Circuit that sets up LDR and STR instruction registers and Flags*

This unit is largely intended to extend some functionality for our Control Unit for more instructions.

Conditional Circuit

The conditional circuit is intended to be used with instructions that require branching. As such, it takes an input from the latched flags and uses the Z flag to determine whether or not a branch will occur and the immediate value to be passed on based on the result of the conditional implemented with multiplexers. The output of each is then routed to a final 4 input multiplexer which determines what the immediate value will be based on whether we read in a B, BEQ, or BNEQ instruction, and the condition is met. The select is done directly with the two bits after the two high order bits in the instruction, tell us which branch is needed. We also use an enable which disables this final multiplexer if the instruction is not a branch instruction to mask the output. This immediate value is sign extended before it is finally used. We also provide an ADD opcode output for convenience when this circuit is used elsewhere, as branch instructions will ADD a number to the program counter.

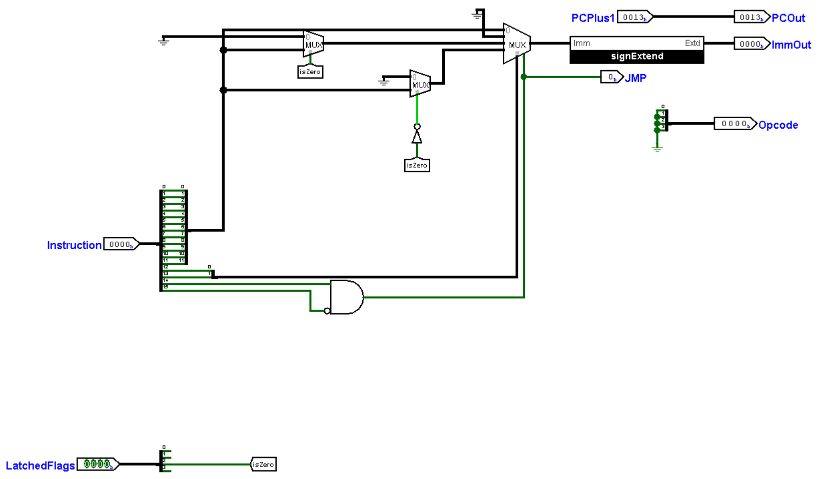


Figure 23: *Testing and evaluating branch instructions circuit*

Handling Flag Movement

This circuit handles any non-ALU based operations, notably handling the movement of the latched flags to another register and setting the appropriate boolean flag if necessary. Its primary purpose is to supply the latched flags to a register. Based on the instruction, the output will either output the latched flags, zero extended, or not output anything.

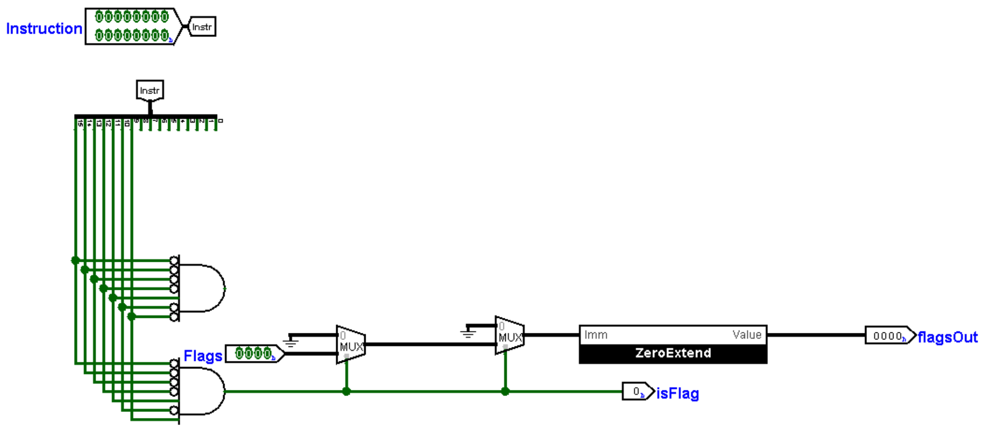


Figure 24: *Subcircuit testing non-ALU instructions, notable flag movement instructions*

Register File Control

The Register File Control is a final check before the necessary data enters the register file, ALU, and other circuits. It is an extension of the Control circuit that largely tests for LDR and STR operations and preps data accordingly. It uses one final stage of multiplexers, with the select on each multiplexer based on the *isLoad* and *isStore* flags, before the data is routed to where it needs to go. Keeping this unit seperate from the control unit allows for easier data routing as multiplexers are all combined into one subcircuit.

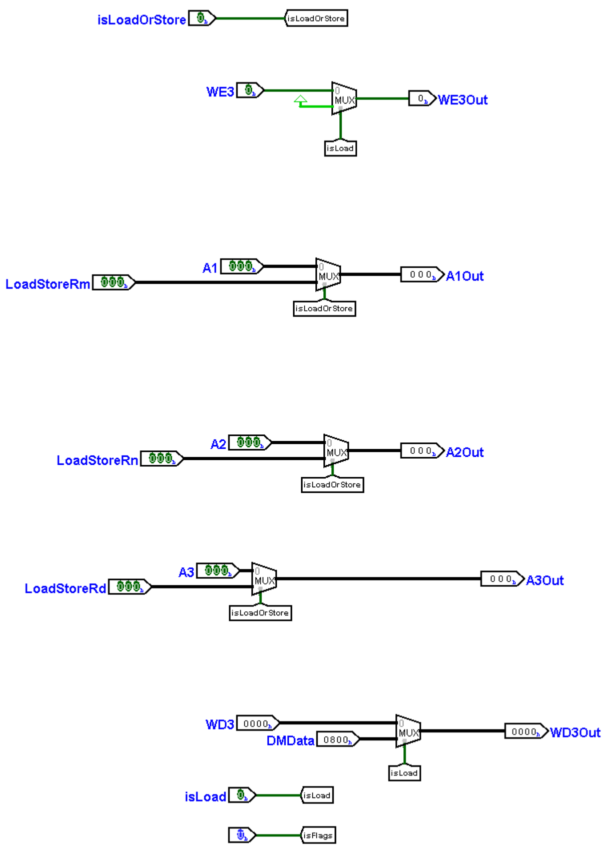


Figure 25: *The Register File Control*

Data Routing

With all necessary components, we can now focus on routing data. All data routing was done with 2 input multiplexers with selects based on the various boolean flags the above modules provided to allow for simple and easy routing.

The first routing occurs directly out of the control unit for the ALU opcode. If an LDR, STR, or JMP instruction occurred our opcode should be 0000 (ADD). Otherwise, we simply use the opcode the Controller extracted from the instruction. No matter what, our ALU will always compute a result (needed or not),and whether this result is used is determined by various signal flags.

We also need to route between inputs A and B of the ALU. If the instruction read was LDR or STR, then we know that the A should be the program counter and B should be the immediate value extracted from the branch instruction that is outputted from the Conditional circuit. Otherwise, if there was not LDR or STR instruction, we simply use *RD1* and *RD2*. As outputted from the main Control Unit.

We also use multiplexers to route where the result of the ALU goes. This is done with the *isLoadOrStore* flags for the Data memory, *isWriteEnableR7* and *JMP* flags for the instruction fetch unit, and the *WE3* flag for the register file. If none of these flags are ON, then the value is thrown away. For all instructions, data is waiting to be written, but these flags acts as enables that determine whether a write occurs or not.

We also use a multiplexer for our writeback. If WE3 is on, that means a writeback occurred in the register file, and as such is outputted to the write back display on the front panel controller. Otherwise, zero is displayed.

The instruction fetch unit also takes in another multiplexer to decide whether a write to R7 occurred or a branch occurred based on the value of the R7 boolean flags that were outputted from the register file. If either of these occurred, the *JMP* flag in the instruction fetch unit is turned ON.

Another data routing occurs based on the MOV register instruction. The Control circuit requires that, for a MOV immediate instruction be supplied as one of its inputs. We use a multiplexer to determine whether *R1*, the source register, is used in the instruction, or the latched ALU flags are the source register, with the flag *isFlag* from the NonALUOp unit determining which one is selected. Again, the Control circuit may throw this value away if the instruction is not a MOV reg instruction (this is done by the isMov unit within the control unit).

Our final data routing process occurs in Data Memory. A multiplexer is used based on whether or not a STR operation occurred. If it does, the write enable for the Data Memory is enabled, and writing is allowed. Otherwise, writing is disabled to the module.

This completes all data routing for the CPU. Our final CPU circuit is shown below, rotate 90 degrees for larger details

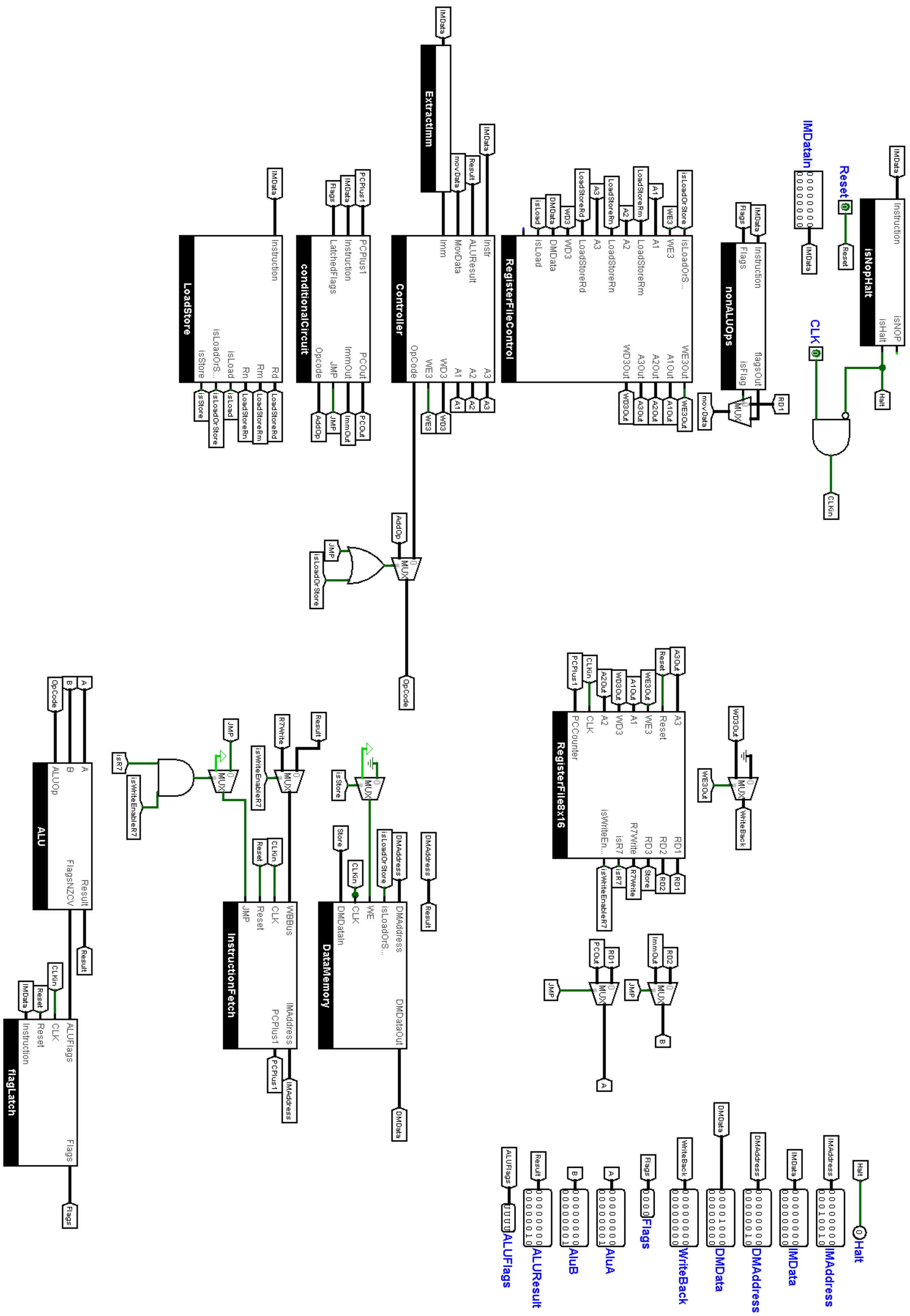


Figure 26: *CPU (rotated)*

Test Programs

The next few sections provide test circuits that were used on the CPU. There are 4 circuits, each of which test various sets of instructions and expects particular outputs. While each one has a primary aim, the objective is to try to ensure that we get a good mix of instructions such that we test various combinations of instructions on the CPU

Testing the ”Odd” Movement between Registers

The following tests movement between registers, mostly with latched flags and the program counter which are ”odd” moves. It also triggers a flag and then supplies a NOP to ensure nothing happens. This program is intended to be seen step by step.

```
0x800a  ; MOV R0, 10       ; Move 10 into R0
0x900a  ; MOV R1, 10       ; Move 10 into R1
0x2211  ; SUB R2, R0, R1   ; Compute 10 - 10, store result in R2
0x0000  ; NOP              ; Test the NOP instruction (Zero flag shouldn't change)
0x0A18  ; MOV R3, FLAGS    ; Move FLAGS into R3
0x08D0  ; MOV R2, R3       ; Move data from R3 into R2
0xF008  ; MOV R7, 8        ; Move 8 into R7 (testing program counter moves)
0x0000  ; NOP              ; Test NOP again
0x09E0  ; MOV R4, R7       ; Store PC value into R4
0x0600  ; HALT             ; HALT
```

Figure 27: *Testing the MOV instructions*

The assembly for using this in our CPU is below.

```
v2.0 raw
v2.0 raw
800a 900a 2211 0000 0A18 08D0 F008 0000
09E0 0600
```

Testing Various ALU operations

The following program tests basic ALU operations

```
800a    ; MOV R0, 10        ; Load 10 into R0
9001    ; MOV R1, 1         ; Load 1 into R1
3401    ; LSL R0, R0, R1    ; Shift R0 left once (R0 is now 20)
3611    ; LSR R2, R0, R1    ; Shift R0 right once, store in R2 (R2 is 10)
b064    ; MOV R3, 100       ; Move 100 into R3
c014    ; MOV R4, 20        ; Move 20 into R4
28ec    ; DIV R5, R3, R4    ; Compute 20/4, store in R5
3000    ; EOR R0, R0, R0    ; Clear R0

3200    ; NOT R0, R0        ; Not R0 (FFFF)
9000    ; MOV R1, 0         ; Move 0 into R1
3248    ; NOT R1, R1        ; Not R1
2040    ; ADD R0, R1, R0    ; Generate a Carry by adding R0 + R1

9001    ; MOV R1, 1         ; Move 1 into R1
3c49    ; ROR R1, R1, R1    ; Rotate R1 once right (8000)
9002    ; MOV R1, 2         ; Move 2 into R1
3a01    ; ROL R0, R0, R1    ; Rotate R0 left twice
0600    ; HALT              ; End
```

Figure 28: *Testing the ALU*

The machine code is below

```
v2.0 raw
800a 9001 3401 3611 b064 c014 28ec 3000
3200 9000 3248 2040 9001 3c49 9002 3a01
0600
```

Testing Branches

The following section intends to test branches. If the code loops infinitely, something is wrong within the CPU.

```
8400    ; MOV R0, 1024      ; Move 1024 into R0
9400    ; MOV R1, 1024      ; Move 1024 into R1
2211    ; SUB R2, R0, R1    ; Activate 0 Flag
7ffc    ; BNE -4            ; If Z flag or branch doesn't work, this program will loop
6002    ; BEQ 2             ; If equal skip the next two instructions; otherwise, will
3000    ; EOR R0, R0, R0    ;  Clear R0 (Shouldn't run)
4ff9    ; B -7              ; Loop infinitely (Shouldn't run)
2050    ; ADD R2, R1, R0    ; Add R1 + R0, store in R2
0888    ; MOV R1, R2        ; Move R2 to R1
0600    ; HALT              ; End program
```

Figure 29: *Testing branch and conditional instruction*

The machine code is listed below

```
v2.0 raw
8400 9400 2211 7ffc 6002 3000 4ff9 2050
0888 0600
```

Load and Store

This is the final test program, and it tests load and store operations. It also does branching, as the previous one, along with some ALU operations. Ideally, 0800 should be at address 0000 and 0C00 should be at address 0001.

```
8400    ; MOV R0, 1024          ; Move 1024 into R0
9400    ; MOV R1, 1024          ; Move 1024 into R1
2211    ; SUB R2, R0, R1        ; Activate Z flag
7000    ; BNE 0                 ; Branch Testing
6002    ; BEQ 2
3000    ; EOR R0, R0, R0
4000    ; B 0
2050    ; ADD R2, R1, R0        ; Compute R2 + R1, store 0800 into R2
0888    ; MOV R1, R2            ; Move R2 into R1
8000    ; MOV R0, 0             ; Clear R0
1808    ; STR R1, R0, R0        ; Store 0800 at address 0
9001    ; MOV R1, 1             ; Next address
2040    ; ADD R0, R1, R0        ; Increment memory counter by 1 (1)
a000    ; MOV R2, 0             ; Clear R2
108a    ; LDR R1, R2, R2        ; Load data from address 0 back into R1
c400    ; MOV R4, 1024          ; Move 1024 into R4
2109    ; ADD R1, R4, R1        ; Should compute 0x0400 + 0x0800
180a    ; STR R1, R0, R2        ; Store Result in address 1
0600    ; HALT                  ; Halt
```

Figure 30: *Testing Load instructions*

The machine code is listed below

```
v2.0 raw
8400 9400 2211 7000 6002 3000 4000 2050
0888 8000 1808 9001 2040 a000 108a c400
2109 180a 0600
```

Fibonacci

We also provide a Fibonacci assembly program. We list the code as an LST file. To run the program, simply put the Fibonacci number you want to compute in memory at address 0x80.

```
; Initialize
8080    ; MOV R0, 128         ; Move 0x80 to R0
9000    ; MOV R1, 0           ; Move 0 to R1
1040    ; LDR R0, R1, R0      ; Get the program counter
a000    ; MOV R2, 0           ; Value in Memory to write to
b001    ; MOV R3, 1           ; Value to add/sub every iteration

; Edge Cases (n = 0 or 1)
2209    ; SUB R1, R0, R1      ; Check to see if value is F(0)
7003    ; BNE 3               ; If Not Equal, skip next three instructions
7003    ; MOV R1, 0           ; Move 0 to R1
1889    ; STR R1, R2, R1      ; Store Value
0600    ; HALT                ; End program

9001    ; MOV R1, 1           ; Move 1 to R1
2209    ; SUB R1, R0, R1      ; Check to see if value is F(0)
7005    ; BNE 5               ; If Not Equal skip next 5 instructions
9000    ; MOV R1, 0           ; Move value F(0) to R1
1889    ; STR R1, R2, R1      ; Store value
9001    ; MOV R1, 1           ; Move F(1) to R1
184a    ; STR R1, R1, R2      ; Store F(1) at address (1)
0600    ; HALT                ; End Program

; Reinitialize values
8080    ; MOV R0, 128         ; Move 0x80 to R0
9000    ; MOV R1, 0           ; Move F(0) to R1
1040    ; LDR R0, R1, R0      ; Get the program counter
a000    ; MOV R2, 0           ; Value in Memory to write to
b001    ; MOV R3, 1           ; Value to add/sub every iteration
188a    ; STR R1, R2, R2      ; Initalize F(0) and F(1) in memory
204b    ; ADD R1, R1, R3      ; Compute F(1)
188b    ; STR R1, R2, R3      ; Store F(0) at address 1
a001    ; MOV R2, 1           ; Most Recent Fib value
b001    ; MOV R3, 1           ; Value to add/sub every iteration
c000    ; MOV R4, 0           ; Zero register
2203    ; SUB R0, R0, R3      ; Decrement R0 by 1 (we've computed F(1))

; Iteration
10ac    ; LDR R5, R2, R4      ; Load F(n) at Address R2
22b3    ; SUB R6, R2, R3      ; Compute n-1
11b4    ; LDR R6, R6, R4      ; Get F(n-1)
214e    ; ADD R1, R5, R6      ; Compute Fib Value
2093    ; ADD R2, R2, R3      ; Update Memory Address
188c    ; STR R1, R2, R4      ; Store Fib Value
2203    ; SUB R0, R0, R3      ; Decrease R0
6001    ; BEQ 1               ; If R0 is 0, skip next instruction
4ff7    ; B -9                ; Otherwise, iterate
0600    ; HALT
```

Figure 31: *Fibonacci Program*

The assembly file alone is shown below

```
v2.0 raw
8080 9000 1040 a000 b001 2209 7003 9000
1889 0600 9001 2209 7005 9000 1889 9001
184a 0600 8080 9000 1040 a000 b001 188a
204b 188b a001 b001 c000 2203 10ac 22b3
11b4 214e 2093 188c 2203 6001 4ff7 0600
```

Testing Register 7 (Program Counter) Operations

The following is intended to give a more detailed test of operations on register 7, as it is crucial to the execution of a program and operating errors on it can cause serious problems.

```
0000    ; NOP                   ; Increment PC
0000    ; NOP
0000    ; NOP
8004    ; MOV R0, 4             ; Setup Immediate values
9004    ; MOV R1, 4
2039    ; ADD R7, R0, R1        ; Move 8 into R7 (Update PC)
0000    ; NOP                   ; Shouldn't Execute
0000    ; NOP                   ; Shouldn't Execute
09D0    ; MOV R2, R7            ; Move program counter value into R2
9000    ; MOV R1, 0             ; Address to store PC
1879    ; STR R7, R1, R1        ; Store R7 at address 0
0600    ; HALT                  ; End program
```

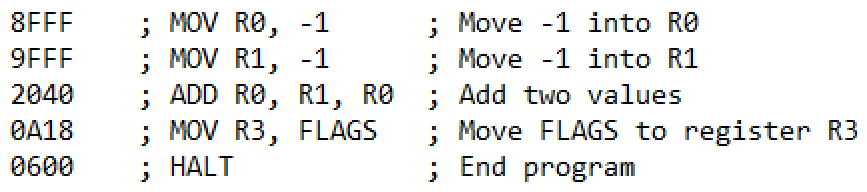
Figure 32: *Testing Latched Flags*

The assembly is shown below for this program

```
v2.0 raw
0000 0000 0000 8004 9004 2039 0000 0000
09D0 9000 1879 0600
```

Testing Flag Movement

This small program just tests to make sure Flags moves correctly



```
8FFF    ; MOV R0, -1      ; Move -1 into R0
9FFF    ; MOV R1, -1      ; Move -1 into R1
2040    ; ADD R0, R1, R0  ; Add two values
0A18    ; MOV R3, FLAGS   ; Move FLAGS to register R3
0600    ; HALT            ; End program
```

Figure 33: *Testing Latched Flags*

The assembly code is below

```
v2.0 raw
8FFF 9FFF 2040 0A18 0600
```

Memory

Question 8.8

A cache has the following parameters: b, block size given in numbers of words; S, number of sets; N, number of ways; and A, number of address bits

(a) In terms of the parameters described, what is the cache capacity, C?
(b) In terms of the parameters described, what is the total number of bits required to store the tags?
(c) What are S and N for a fully associative cache of capacity C words with block size b?
(d) What is S for a direct mapped cache of size C words and block size b?

(a) The cache capacity is defined as the number of bits that the cache is able to hold. As S tells us how many sets per way, we can define it as

$$S = \frac{B}{N} \tag{1}$$

Where B is the number of blocks in the cache. We need to solve for B, which gives us

$$B = SN \tag{2}$$

As b gives us the block size given in number of words, we get that our cache has a capacity of

$$C = SNb \tag{3}$$

Our book defines a word as 4 bytes (for a 32 bit ARM CPU) so in terms of bytes, our answer is

$$C = 4SNb \tag{4}$$

(b) The tag for a cache allows us to check for a hit. By default, if you S sets, then you need

$$SetBits = log_2(S) \tag{5}$$

We also need to use some bits to determine the number of bits needed for the byte offset, which in our case is simply

$$OffsetBits = log_2(b) \tag{6}$$

This determines the number of offset bits needed. However, Assuming we have A address bits, the number of bits needed for the tag is given by

$$TagBits = A - (log_2(S) - log_2(b)) \tag{7}$$

(c) For a full associative cache, all cache blocs are in one set. This means that

$$S = 1 \tag{8}$$

The number of ways, or associativity, of a cache, is the number of blocks in a set. To find the number of blocks, we recognize that the capacity of the cache over the block size in words b gives us the bits per block. In our case, this leads to the equation

$$Blocks = \frac{C}{b} \tag{9}$$

As we have only 1 set in a fully associative cache, the final answer for the number of ways is simply

$$N = 1 * (Blocks) = Blocks = \frac{C}{b} \tag{10}$$

(d) A direct mapped cache is defined as having one block per set. Again, the number of blocks in a cache can be defined as

$$Blocks = \frac{C}{b} \tag{11}$$

And as each block is its own set, we get

$$Sets = Blocks = \frac{C}{b} \tag{12}$$

Question 8.10

Repeat Exercise 8.9 for the following repeating sequence of lw addresses (given in hexadecimal) and cache configurations. The cache capacity is still 16 words.

74 A0 78 38C AC 84 88 8C 7C 34 38 13C 388 18C

(a) direct mapped cache, b = 1 word
(b) fully associative cache, b = 2 words
(c) two-way set associative cache, b = 2 words
(d) direct mapped cache, b = 4 words

(a) For a direct mapped cache with a block size of 1 word, we will have the two lower bits for the offset and the next 4 for the set, as each block is in its own set. This will give us the following cache memory model

Set	Value
0	EMPTY
1	84
2	88 388
3	38C 8C 18C
4-7	EMPTY
8	A0
9	EMPTY
10	EMPTY
11	AC
12	EMPTY
13	74 34
14	78 38
15	7C 13C

As our block size is 1 word, and only 3 hits occur (the values don't change). As such, our miss rate is $1 - \frac{3}{14} = \frac{11}{14}$

(b) For a fully associative cache with block size 2, each block is in the same set. As such, we expect to have 8 ways. This gives us the cache memory model listed below. We notice that there were only two hits, and as such, that means that our miss rate was $\frac{12}{14}$.

Set	Value
0	[74, 78]
1	[A0, A4] [13C, 140]
2	[38C, 390] [388, 38C]
3	[AC B0] [18C, 190]
4	[84, 88]
5	[8C, 90]
6	[7C, 80]
7	[34, 38]

(c) Using a two way set associative cache with a block size of two words will give us that the total number of sets will be 4. This can be derived by realizing that if we 2 words in each block, and each set maps to one of two ways, we have 4 words per set (evenly split between two ways). As a result, for a 16 bit cache, we will expect that we have $\frac{16}{4} = 4$ sets. So, 2 bits will be used for set after the two bits for the offset. Using this cache model on the addresses given gives us the model below.

Set	Value
0	[A0, A4]
1	[74, 78] [84, 88] [34 38]
2	[389, 390]
3	[38C, 390] [AC, B0] [8C, 90] [7C, 80] [13C, 140] [18C, 190]

From our cache structure, we can see that our miss rate, with the addresses is $\frac{6}{14}$.

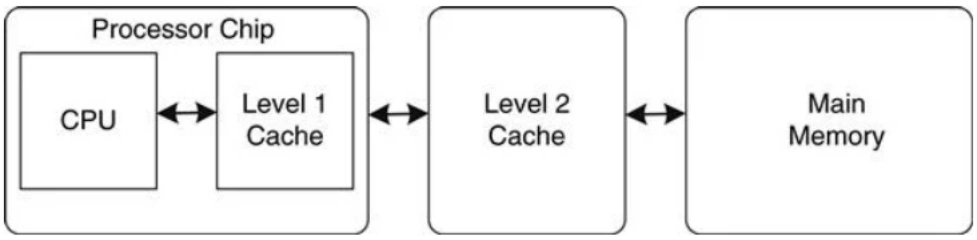
(d) For a direct mapped cache with 4 words, we get the a cache with 4 sets, each one holding a block. As such, we have 2 bits after the 2 offset bits for the sets. This gives us the cache model below

Set	Value
0	[A0, AC]
1	[84, 90] [34, 40] [388 394]
2	[74, 80]
3	[38C, 398] [13C, 148] [18C, 198]

Our miss rate is thus $\frac{7}{14}$ as our hit rate is 50%.

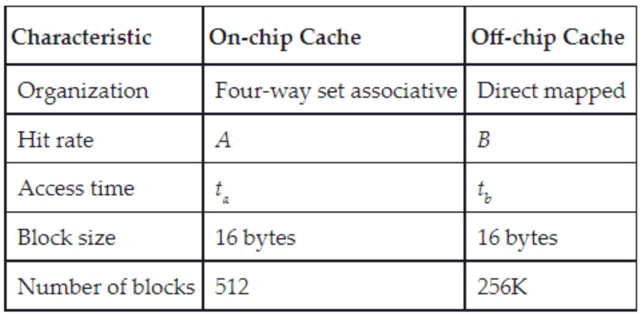
Question 8.14

You've joined a hot new Internet startup to build wrist watches with a built-in pager and Web browser. It uses an embedded processor with a multilevel cache scheme depicted in the figure below. The processor includes a small on-chip cache in addition to a large off-chip second-level cache.



Assume that the processor uses 32-bit physical addresses but accesses data only on word boundaries. The caches have the characteristics given in Table 8.5. The DRAM has an access time of t_m and a size of 512 MB.

(a) For a given word in memory, what is the total number of locations in which it might be found in the on-chip cache and in the second-level cache?
(b) What is the size, in bits, of each tag for the on-chip cache and the second-level cache?
(c) Give an expression for the average memory read access time. The caches are accessed in sequence.
(d) Measurements show that, for a particular problem of interest, the on-chip cache hit rate is 85% and the second-level cache hit rate is 90%. However, when the on-chip cache is disabled, the second-level cache hit rate shoots up to 98.5%. Give a brief explanation of this behavior.



Characteristic	On-chip Cache	Off-chip Cache
Organization	Four-way set associative	Direct mapped
Hit rate	A	B
Access time	t_a	t_b
Block size	16 bytes	16 bytes
Number of blocks	512	256K

(a) The total number of locations data can be stored in the cache is simply the block size in the cache multiplied by the number of blocks. As such, this gives us the equation

$$OnChip = 512 * 16 = 8192 \tag{13}$$

for the on-chip cache, and

$$OffChip = 16 * 256K = 4096K \tag{14}$$

for the off-chip cache.

(b) The on-chip cache is a 4-way set associative cache, meaning that each set has 4 (or blocks). Given that we have 512 blocks on this cache chip, we have a total of

$$\frac{512}{4} = 128 \tag{15}$$

sets on the cache, which require a total of 7 bits to represent. We also need 4 bits for the byte offset, as $2^4 = 16$ and a word is 16 bytes. As such, the tag has a total of

$$Tag = 32 - (7 + 4) = 21 \tag{16}$$

bits for the on-chip cache. As for the off-chip cache, we have a direct mapped cache with 256K blocks, and so have just as many sets. To represent this number, we need 18 bits, as $log_2(262144) = 18$. We also need another 4 bits for the byte offset. This tells us that the tag bits can be described as

$$TagOffChip = 32 - (18 + 4) = 10 \tag{17}$$

bits for the off-chip cache.

(c) The expression for the AMAT of our two three level memory system is shown below

$$AMAT = t_a + (1 - A)(t_b + (1 - B)t_m) \tag{18}$$

(d) The reason for why the off-chip cache system would see an increase in hit rate due to disabling the on-chip cache has to do with the fact that the on-chip cache will mostly hold data that is accessed regularly or being currently operated on - that is, data with a lot of temporal and spatial locality. On the other hand, the off-chip cache will pull data that is accessed occasionally, but not as regularly as the on-chip cache (memory contains data that is rarely accessed). This means that, in general, data that is regularly accessed will count as a hit for the first level cache only, not the second level cache.

However, when you have disabled the on-chip cache, your off-chip cache now contains data that is both regularly accessed and occasionally accessed. This means that you can now expect a hit rate increase for your off-chip cache, which makes the hit rate higher on the off-chip cache. This explains the reasoning for the observation made in this problem.