

<BSCS/ COMPUTER SCIENCE /MAJOR>

ASSESSMENT REPORT ACADEMIC YEAR 2021 – 2022

I. LOGISTICS

1. Please indicate the name and email of the program contact person to whom feedback should be sent (usually Chair, Program Director, or Faculty Assessment Coordinator).

EJ Jung, ejung2@usfca.edu, Faculty Assessment Coordinator of CS dept.

2. Please indicate if you are submitting report for (a) a Major, (b) a Minor, (c) an aggregate report for a Major & Minor (in which case, each should be explained in a separate paragraph as in this template), (d) a Graduate or (e) a Certificate Program

(a) CS Major

3. Please note that a Curricular Map should accompany every assessment report. Has there been any revisions to the Curricular Map?

No changes were made.

II. MISSION STATEMENT & PROGRAM LEARNING OUTCOMES

1. Were any changes made to the program mission statement since the last assessment cycle in October 2021? Kindly state “Yes” or “No.” Please provide the current mission statement below. If you are submitting an aggregate report, please provide the current mission statements of both the major and the minor program

Mission Statement (Major/Graduate/Certificate):

No changes were made.

Students who graduate with a Bachelor of Science (B.S.) degree in Computer Science will be prepared for both graduate school and for software development careers. The curriculum provides a solid base in computer science fundamentals that includes software design and development, problem solving and debugging, theoretical and mathematical foundations, computer systems, and system software.

- 2. Were any changes made to the program learning outcomes (PLOs) since the last assessment cycle in October 2020? Kindly state “Yes” or “No.” Please provide the current PLOs below. If you are submitting an aggregate report, please provide the current PLOs for both the major and the minor programs.**

PLOs (Major/Graduate/Certificate):

No changes were made.

- THEORY: Explain and analyze standard computer science algorithms and describe and analyze theoretical aspects of various programming languages.
- APPLICATION: Apply problem-solving skills to implement medium- and large-scale programs in a variety of programming languages.
- SYSTEMS: Describe the interactions between low-level hardware, operating systems, and applications.
- PROJECT: Demonstrate effective communication and organization as part of a team of software developers or researchers collaborating on a large computer program.

- 3. State the particular Program Learning Outcome(s) you assessed for the academic year 2020-2021.**

PLO(s) being assessed (Major/Graduate/Certificate):

- SYSTEMS: Describe the interactions between low-level hardware, operating systems, and applications.

III. METHODOLOGY

Describe the methodology that you used to assess the PLO(s).

For this assessment we focus on how CS 315 is meeting part of the SYSTEMS area learning outcomes, specifically the ability of students “to describe the interactions between low-level hardware and applications.” Our assessment focuses on 4 sections of CS 315 from Fall 2021. These 4 sections consisted of 54 total students.

Course Objectives

Computer Architecture refers to the organization of the hardware that executes computer programs. The processor is the most important and complex part of a computer system. As such, it is very important for software developers to understand how processors execute code correctly and with high performance.

This course examines the machine representation of data, low-level programming in C and assembly language, machine language, and the design and implementation of processors at the digital logic level.

Learning Outcomes and Assessments

Upon completion of this course the students will have learned the following topics:

1. Understand and manipulate the machine-level representation of numbers and data in C (labs)
2. Write C and assembly language programs for the ARM architecture (Project02, Project03)
3. Design and implement an emulator which executes ARM machine code as a processor might (Project04)
4. Learn about digital design by building circuits which solve computational problems (Project05)
5. Design and implement a single-cycle processor using schematics in a simulator (Project06)
6. Design and implement a pipelined processor using schematics in a simulator (Project07)

Assessment

1. Projects are the main assessment tool for the course. The project corresponding to each learning outcome is shown in parentheses above.
2. Projects will be graded for both correctness (using an automated tool) and comprehension (using 1:1 interactive grading meeting with the instructor or TA)
3. Low-stakes labs will build into each project
4. The final exam will be comprehensive over the semester's work

We analyze the performance of the Fall 2021 CS 315 students on two projects: Project04 and Project06, since these are the most comprehensive projects and require the synthesis of many different concepts. These two projects account for 30% of the student's overall grade.

Both of these projects are interactively graded so students not only have to show that their solutions can pass automated test cases, they also have to explain their implementations orally and respond to questions that allow them to demonstrate their understanding of their solutions and underlying concepts. See attached project descriptions and rubrics.

IV. RESULTS & MAJOR FINDINGS

What are the major takeaways from your assessment exercise?

To assess mastery, we split the students into four groups:

1. Complete mastery of the outcome
2. Mastered the outcome in most parts
3. Mastered some parts of the outcome
4. Did not master the outcome at the level intended

Mastery Group / Project	1	2	3	4	Total
Project04	21	9	7	17	54
Project06	10	10	7	27	54
Class Score	17	18	10	9	54

Interpretation of results.

- For Project04, 69% of the students mastered at least some parts of the learning outcomes, whereas 39% of students achieved complete mastery.
- For Project06, 50% of the students mastered at least some parts of the learning outcomes, whereas 19% of the students achieved complete mastery.
- Note that 4 of the 54 students effectively did not show up to class and did not complete any assignments. They either did a late withdrawal or they earned an F in the class. It is reasonable to eliminate these students from category 4 because they really didn't attempt the coursework.
- The most interesting result of this analysis is the jump from 17 students in group 4 (did not master the outcome at the level intended) for Project04 to 27 for Project06. Here are factors that contribute to this result:

- Project06 is a significant implementation using digital design (digital circuits), which is much different than conventional programming. Students learn digital design in Lab05, and Project05, but their exposure is limited compared to having several classes in computer programming. For Project04 students use the C Programming Language, which is conventional and also covered in other courses.
- Project06 is due late in the semester just before Thanksgiving. There seems to be a “burnout” factor at this point in the semester.
- Note that students are given a chance to earn back points lost, in this case up to 80% of lost points after the due date, but given that this assignment is the second to last project and the last major project, some students don’t seem to have time to earn back points and ultimately master the project concepts.
- Fall 2021 was the first semester in which classes were offered in person after a year of remote learning, however in CS 315 remote attendance was allowed. Therefore, students taking prerequisite courses did so completely online, which may have contributed to an overall lack of preparation and maturity for understanding the material.
- These results also show how student performance on other assignments can mask their low performance on these two critical projects. Only 17% of students did not meet any overall outcomes for the course when considering all the assignments.

V. CLOSING THE LOOP

1. Based on your results, what changes/modifications are you planning in order to achieve the desired level of mastery in the assessed learning outcome? This section could also address more long-term planning that your department/program is considering and does not require that any changes need to be implemented in the next academic year itself.

As we progress into a regular cadence of in-person instruction, we need to re-evaluate this assessment to see if we see a drop in mastery from Project04 to Project06 consistently. We also need to reconsider the overall percentage of the final grade coming from these projects given their importance. That is, higher overall weights will better indicate the importance of

these projects to the students and will make it less likely that students can compensate using the other projects.

In Fall 2022 we changed the target processor architecture type from ARM to RISC-V. It turns out that RISC-V is easier to implement in digital logic compared to ARM. We will perform this assessment analysis again once we complete Fall 2022 to see if this change has an impact on Project06 mastery.

2. What were the most important suggestions/feedback from the FDCD on your last assessment report (for academic year 2020-2021, submitted in October 2021)? How did you incorporate or address the suggestion(s) in this report?

Based on the feedback from the FDCD, we used multiple assignments that demonstrate the target PLO with multiple faculty for assessment this year.

ADDITIONAL MATERIALS

(Any rubrics used for assessment, relevant tables, charts and figures should be included here)



Project04 - ARM Emulator

Due

1. Wednesday 10/13 at 11:59 PM to the GitHub assignment for your section
2. Sign up for interactive grading on Thursday 10/14 using the schedule published by your instructor

Requirements

1. You will write an emulator in C for a subset of the ARMv7 Instruction Set Architecture (ISA).
2. You do not have to emulate the entire instruction set; just enough to emulate `fib_rec`, `get_bitseq`, `is_pal`, `max3`, `merge_sort`, `quadratic`, `sort`, and `to_upper`. You will use your implementation of `sort/find_max_index` and `merge/merge_sort`. The other assembly implementations are given.
3. Your emulator will need the logic (decoding and emulating instructions) and state (`arm_state`) from lab04
4. Your emulator will support dynamic analysis of instruction execution. Here are the metrics you will collect. Note that
 1. # of instructions executed (`i_count`)
 2. # of data processing and mul instructions executed (`dp_count`)
 3. # of SDT (memory) instructions executed (`sdt_count`)
 4. # of branch instructions executed including `b`, `bl`, `bx`, `bCC` (`b_count`)
 5. # of branches taken including `b`, `bl`, `bx`, `bCC` (`b_taken`)
 6. # of branches not taken from `bCC` (`b_not_taken`)
5. Your emulator will include an implementation of a 4-way set-associative cache, as is common in commercial processors. We are giving you an implementation of a direct mapped cache. In addition to the set associative cache implementation, you will collect the following metrics:

	×
--	---

 1. Total memory references (refs)
 2. Hits (hits)

common in commercial processors. We are giving you an implementation of a direct-mapped cache. In addition to the set-associative cache implementation, you will collect the following metrics:

1. Total memory references (refs)
2. Hits (hits)
3. Misses (misses)
4. Cold misses (misses_cold)
5. Hot misses (misses_hot)

Given

1. In lecture and lab, we will:
 1. illustrate how to decode machine code and execute the operations specified
 2. illustrate a direct-mapped cache and describe the data structures and algorithms required for a set-associative cache
2. In-class coding will be pushed to Github. You will provide the rest of the code yourself
3. We will provide autograder test cases for the emulation targets

Grading Rubric

Automated testing

80 pts: Automated tests

Interactive grading

10 pts: code walkthrough, including, but not limited to, your implementation of dynamic analysis and the instruction cache.

Coding Style

10 pts: Clean repo, consistent naming and complex code



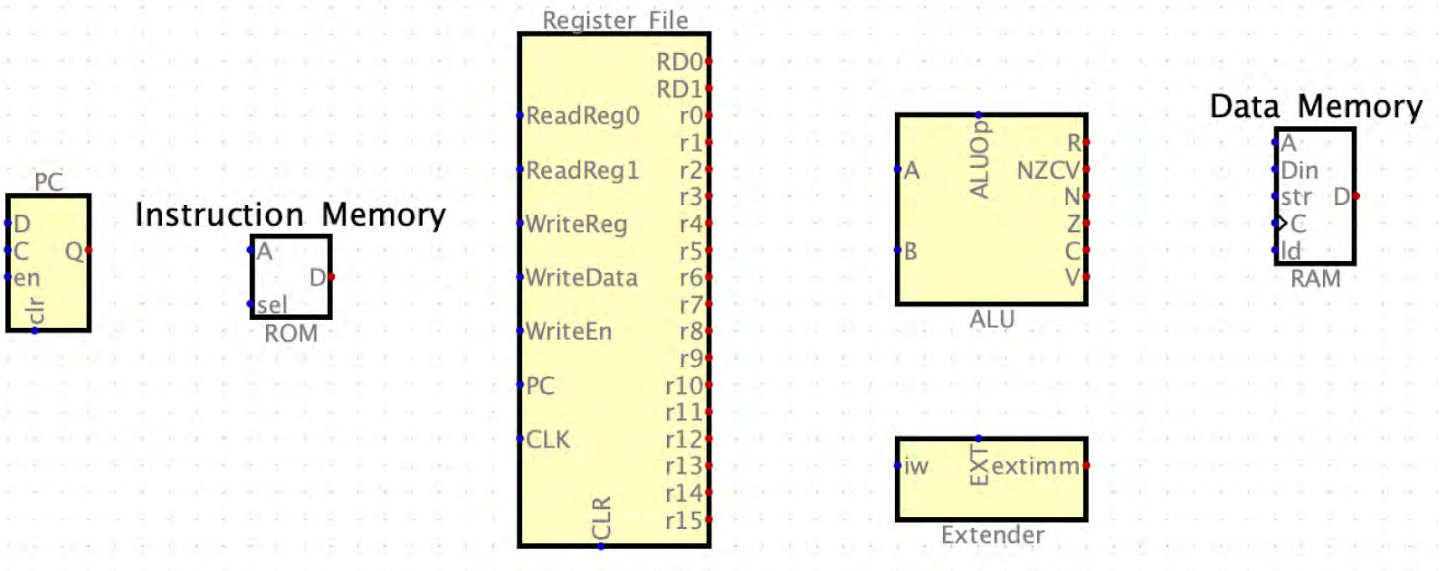
necessarily



Guide to Project06 - Part 1

The project spec for project06 provides high-level requirements. This guide, along with lecture material, will drill down into the details. Using all of this information, you will implement and simulate a single-cycle ARM processor in Digital

Major Components



1. The PC register will hold the address of the instruction we are currently executing
2. The Instruction Memory holds the machine code representation of our test programs, including matches_s and merge_sort_s from Project03.
3. You will use the hex output from `makerom3.py` as input into instruction memory
4. The Register File supports reading and writing from ARM registers. We will expose the register values as outputs, so they may be used as inputs to other circuits
5. The Arithmetic Logic Unit (ALU) supports addition, subtraction, multiplication, logical shift left, and logical shift right. The ALU also computes the condition code bits N, Z, C, V for supporting conditional execution. We will use the ALU for branch target calculation.
6. The Extender supports zero-extension of 8- and 12-bit unsigned immediate values, as well as sign extension of 24-bit bra
7. Data Memory is a Digital RAM component which supports our simulated stack, storing function parameters and preserved register values, as needed.

7. Data Memory is a Digital RAM component which supports our simulated stack, storing function parameters and preserved register values, as needed.

PC

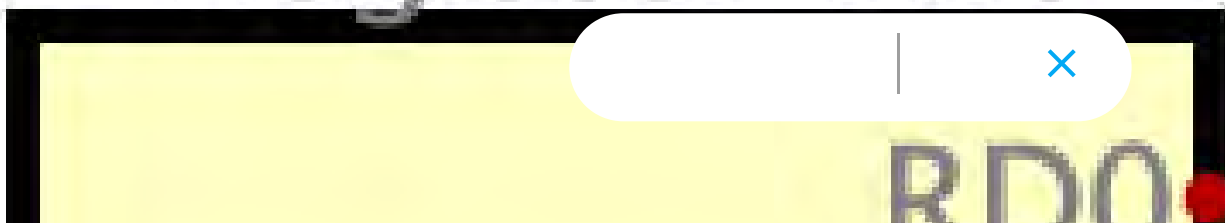
1. We will represent PC using a Digital Register component, in the top level of our processor
2. Digital does not provide a CLR input to its registers, but we can add one as our project grows
3. EN will always be 1 so we can move the PC forward 4 bytes on each CLK cycle
4. As in Project04, you will update the PC with a calculated address for BL and BX instructions
5. The D output will address the Instruction Memory to retrieve the instruction word at the PC address

Instruction Memory

1. Instruction memory will be built out of Digital ROM components, starting with one and expanding to more for each simulated function
2. Each ARM machine code instruction is 32 bits, so the ROM will have 32 data bits
3. The number of address bits will vary based on the size of your simulated programs. Recall that with 8 address bits, you can address $2^8 = 256$ instructions
4. Similar to Project05, note that the ROM is addressed by 32-bit words, but the PC represents a byte address. You can obtain the word address using a splitter
5. You will use `makerom3.py` to generate a `.hex` file which can be loaded into the ROM.

Register File

Register File



1. You will create a circuit which supports reading from two registers and writing to one register in the same clock cycle.

2. Inputs

1. ReadReg0 (4 bits) selects the register value to output on RD0
2. ReadReg1 (4 bits) selects the register value to output on RD1
3. WriteReg (4 bits) selects the destination register to update
4. WriteEn (1 bit) determines whether we will update a register in this clock cycle
5. WriteData (32 bits) contains the value we will write to WriteReg if WriteEn is high
6. PC (32 bits) contains the value of the PC so register 15 (PC) can be selected on RD0 or RD1. Hint: you will want to input $PC + 8$ when computing the branch target address for a branch instruction
7. CLK (1 bit) is the clock input from the top-level clock component
8. CLR (1 bit) allows the register values to be set to zero on a clock tick

3. Outputs

1. RD0 (32 bits) contains the value of the register specified by ReadReg0
2. RD1 (32 bits) contains the value of the register specified by ReadReg1
3. r0-r15 (32 bits) are outputs for the 16 registers (including the PC which is passed through).
 1. You should use these as inputs to tunnels which are used to build the dashboard on the top-level circuit.
 2. You should also reflect PC and PC+8 on the dashboard using tunnels, since that will be useful for debugging branch calculation

Arithmetic Logic Unit (ALU)





1. The ALU is a combinational logic component. It does not hold state like sequential logic components, so it does not need a CLK input.
2. Data processing instructions use addition, subtraction, multiplication, logical shift left, and logical shift right.
3. Load/store memory instructions use addition and subtraction to calculate target memory addresses
4. Branch instructions use addition to calculate the target branch address
5. Inputs
 1. A (32 bits) is the first ALU operand
 2. B (32 bits) is the second ALU operand
 3. ALUop (3 bits) selects an ALU operation:
 1. 0b000: add
 2. 0b001: sub
 3. 0b010: mul
 4. 0b011: mov
 5. 0b100: lsl
 6. 0b101 lsr
6. Outputs
 1. R (32 bits) is the ALU result
 2. NZCV (4 bits) are the CPSR values computed for a CMP instruction. The 4-bit output will be wired to the Control Unit
 1. You should also wire tunnels from NZCV to the dashboard for debugging purposes
7. If you need your ALU to support more operations, you may need to add more selector bits and connect those to the

Initial Top-Level Circuit



∴ This starter circuit gives you a general idea of how to start.

Extender

1. The extender will perform sign extension of 8- and 12-bit unsigned immediate values (that is, zero extension out to 32 bits) and 24 bit signed branch offsets (that is, test bit 23 and extend its value out to 32 bits)

2. Inputs

1. iw (32 bits) the current instruction word

2. EXT (2 bits) the extender selector, which selects between

1. 00: 8-bit zero extension. Concatenate the low 8 bits with 24 bits of 0 to output a 32-bit value

2. 01: 12-bit zero extension. Concatenate the low 12 bits with 20 bits of 0 to output a 32-bit value

3. 10: 24 bit sign extension. Recall from Project04 that the 24-bit branch offset is expressed in words, and the PC is expressed in bytes. Therefore you must multiply the sign-extended value by 4 as you did in Project04. This can be accomplished using Digital's Barrel Shifter or Multiply components.

3. Outputs

1. extimm (32 bits) the extended value

Data Memory





1. We will simulate our stack using the Digital RAM component
2. We will use the component with separate load(ld) and store (str) inputs.
3. We will configure the RAM for 32 data bits, which means we can load and store only 32-bit values (which is why won't be able to execute reverse_s)
4. You will configure the number of address bits based on the amount of stack space your test programs need. Recursive functions may use quite a bit of stack space
5. Inputs
 1. A (you figure out how many bits you need): the input address for the read or write to memory
 2. Din (32 bits): the value to be written to memory when str is 1
 3. str (1 bit): set to 1 if we are storing (writing) to memory
 4. C (1 bit): clock input
 5. ld (1 bit) set to 1 if we are loading (reading) from memory
6. Outputs
 1. D (32 bits): the data value we read from memory at address A
7. As with the ROM, we are only supporting word addresses. Therefore, when calculating a target memory address for LDR or STR instructions, we need to convert the byte address into a word address before sending it to the RAM component
8. In addition, we need to ensure that the target address is the same as the number of bits in the A input

In this post and in upcoming lectures, we

instructions and splitting off the bits of the instruction word to interpret what the

instruction means. Once we have a circuit which represents the meaning of the

instruction, we can use that to make decisions about how to select and direct data

8. In addition, we need to ensure that the target address is the same as the number of bits in the A input

In this post and in upcoming lectures, we will discuss a methodology for identifying instructions and splitting off the bits of the instruction word to interpret what the instruction means. Once we have a circuit which represents the meaning of the instruction, we can use that to make decisions about how to **select and direct** data on the data path and determine the control inputs to the processor components such as the register file and ALU.

Adding Control and Data Paths

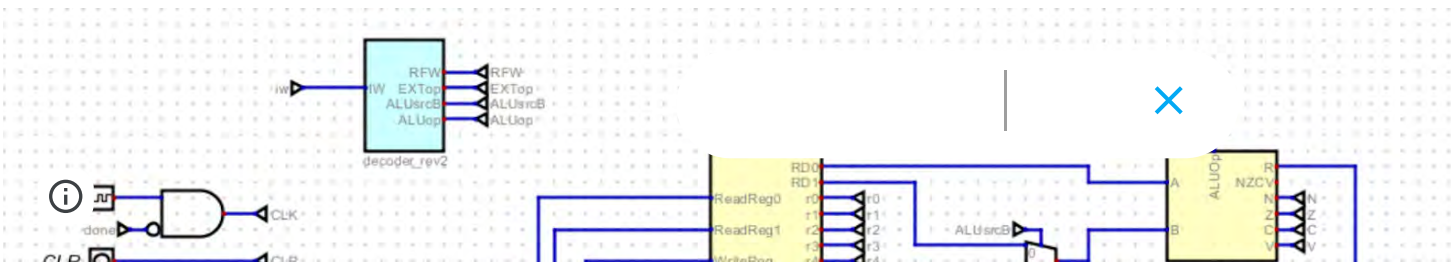
To start, let's use this short program:

```
first_s:
```

```
    mov r0, #1
    mov r1, #2
    add r2, r0, r1
    add r0, r0, #0
```

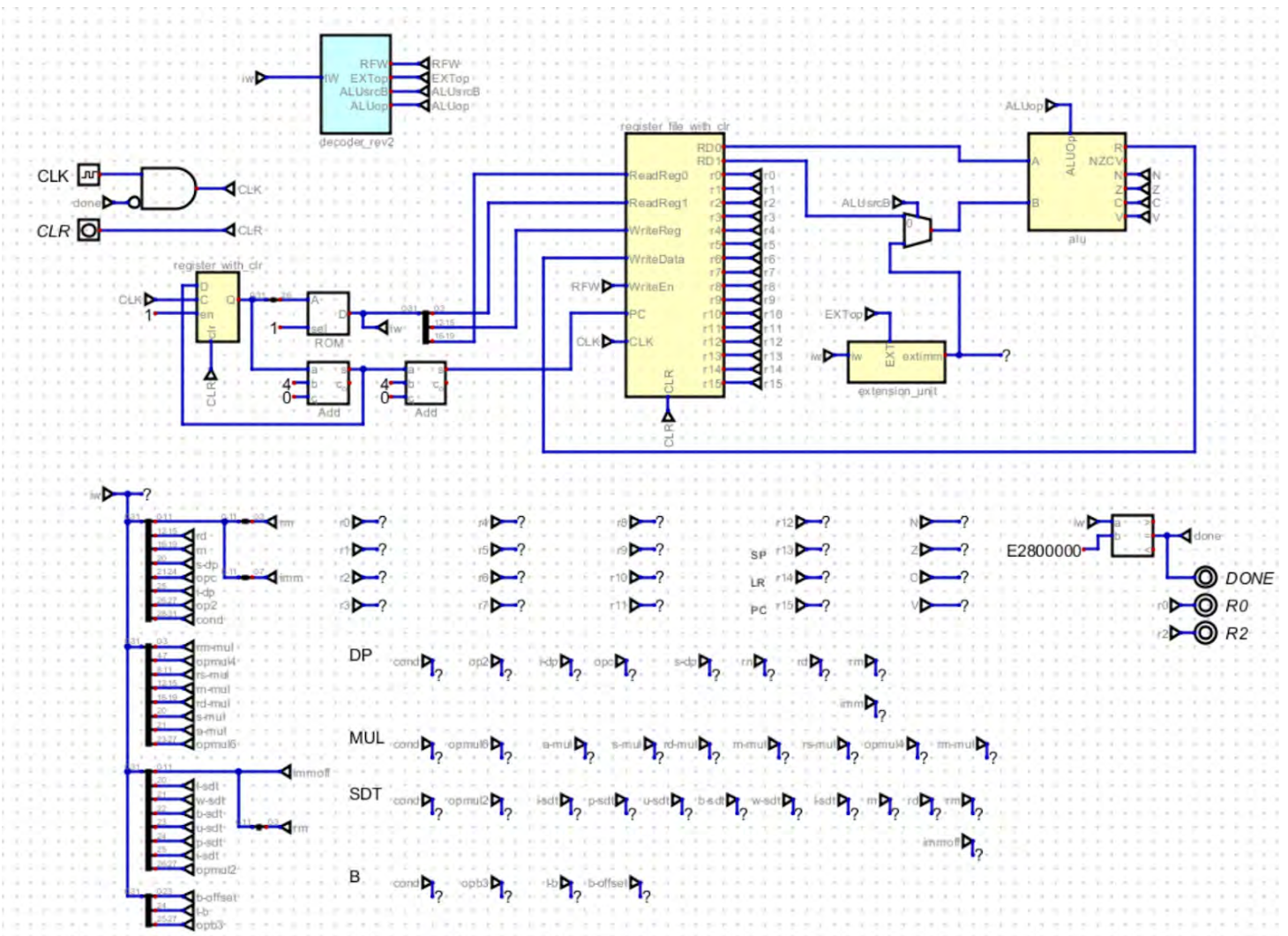
To execute these instructions we need to:

1. Add a data path for the add instruction word coming out of the ROM. In the schematic below, that's shown with bits 0-3 (rm) going into ReadReg0, bits 12-15 (rn) going into ReadReg1, and bits 16-19 (rd) going into WriteReg
2. Add a data path for the mov instruction. In the schematic, that's shown using the iw tunnel coming out of the ROM, and into the Extension Unit.
3. Add an initial Decoder (Control Unit) to choose between adding registers and moving immediate data. The schematic shows the output of that decision in ALUop, which is the selector into the ALU.
4. Now you can execute the mov and add instructions to put $1 + 2 = 3$ in r2



ALUop, which is the selector into the ALU.

4. Now you can execute the mov and add instructions to put 1 + 2 = 3 in r2



Developing the Control Unit

We will build the circuit for the control unit using the digital logic gates we learned about in Project05. To build the requirements, we will use a truth-table-like spreadsheet:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	
1																														
2																														
3																														
4																														
5																														
6																														

Using the Instruction Set Manual,

1. We can identify bits 27 and 26 from the instruction word for data processing



	IW input bits								Control output bits															
	27	26	25	24	23	22	21	20	RFW	EXTop1	Extop0	ALUsrcB	ALUop2	ALUop1	ALUop0	P8	P7	P6	P5	P4	P3	P2	P1	P0
5	add(r)	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	mov(i)	0	0	1	1	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0

Using the Instruction Set Manual,

1. We can identify bits 27 and 26 from the instruction word for data processing instructions, recall if these are set to 00, then the instruction is possibly a data processing instruction.
2. We can identify the Data Processing opcode in bits 24-21. Add is 0b0100 and mov is 0b1101.

We will use a simple model for outputs for the control unit:

1. Since the Register File has a WriteEn control input, let's make up a RFW (Register File Write) output to match it up with. RFW will be 1 when we identify an instruction which needs to write to the register file
2. Since the Extender needs a control input to select its operations, let's make up an EXTop output to match it up with. The values coming out of EXTop will be
 1. 0b00: 8-bit zero-extended immediate
 2. 0b01: 12-bit zero-extended memory address
 3. 0b10: 24-bit sign-extended branch offset
 4. 0b11: 5-bit zero-extended shift immediate
3. Since the ALU needs a control input to select its operations, let's make up an ALUop to select arithmetic instructions in the ALU. The values coming out of ALUop will be:
 1. 0b000: addition
 2. 0b001: subtraction
 3. 0b010: multiplication
 4. 0b011: move
 5. 0b100: logical shift left
 6. 0b101: logical shift right
 7. 0b110: arithmetic shift right

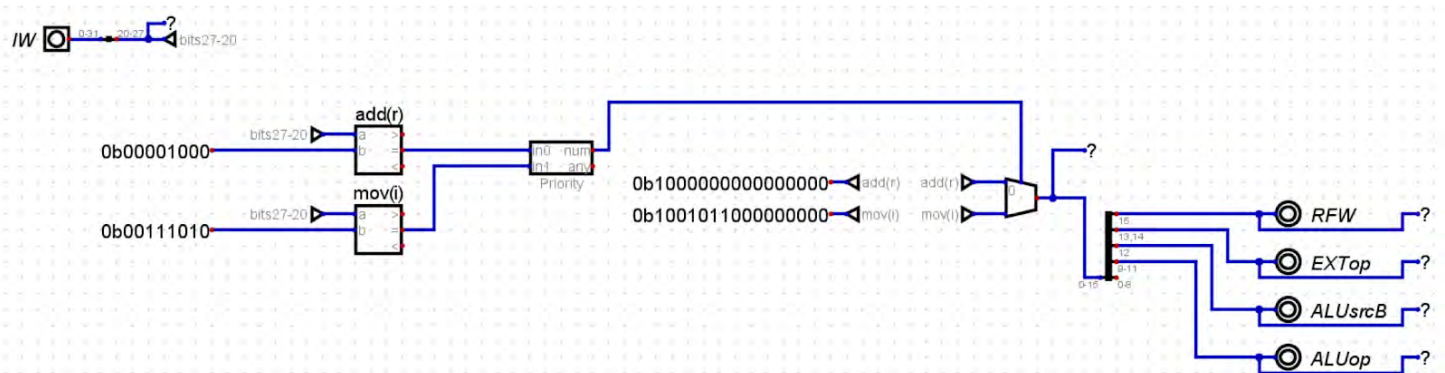


6. 0b101: logical shift right

7. 0b110: arithmetic shift right

4. Since we need a way to differentiate between ALU operations which use registers vs. those which use immediate values, we will make up a data output to send to the Extender Unit, and call it ALUsrcB


This schematic shows how to construct gates which match the spreadsheet above, and can execute the "first_s.s" program above. You will add instructions to the spreadsheet, which may involve new inputs and outputs, and add build a circuit like this within your Control Unit



Adding Support for BL and BX

Now that you can execute ADD and MOV instructions, let's move on to a version which supports BL and BX, as implemented in first_main.s

```
main:
    mov r0, #1
    mov r1, #2
    bl first_s
    add r0, r0, #0
first_s:
    add r0, r0, r1
    bx lr
```

To support branch, we need to modify the , as well as the control unit. In the schematic, that's shown with the branch multiplexer choosing PC + 4 or a branch target calculated by the ALU

```
add r0, r0, #0
```

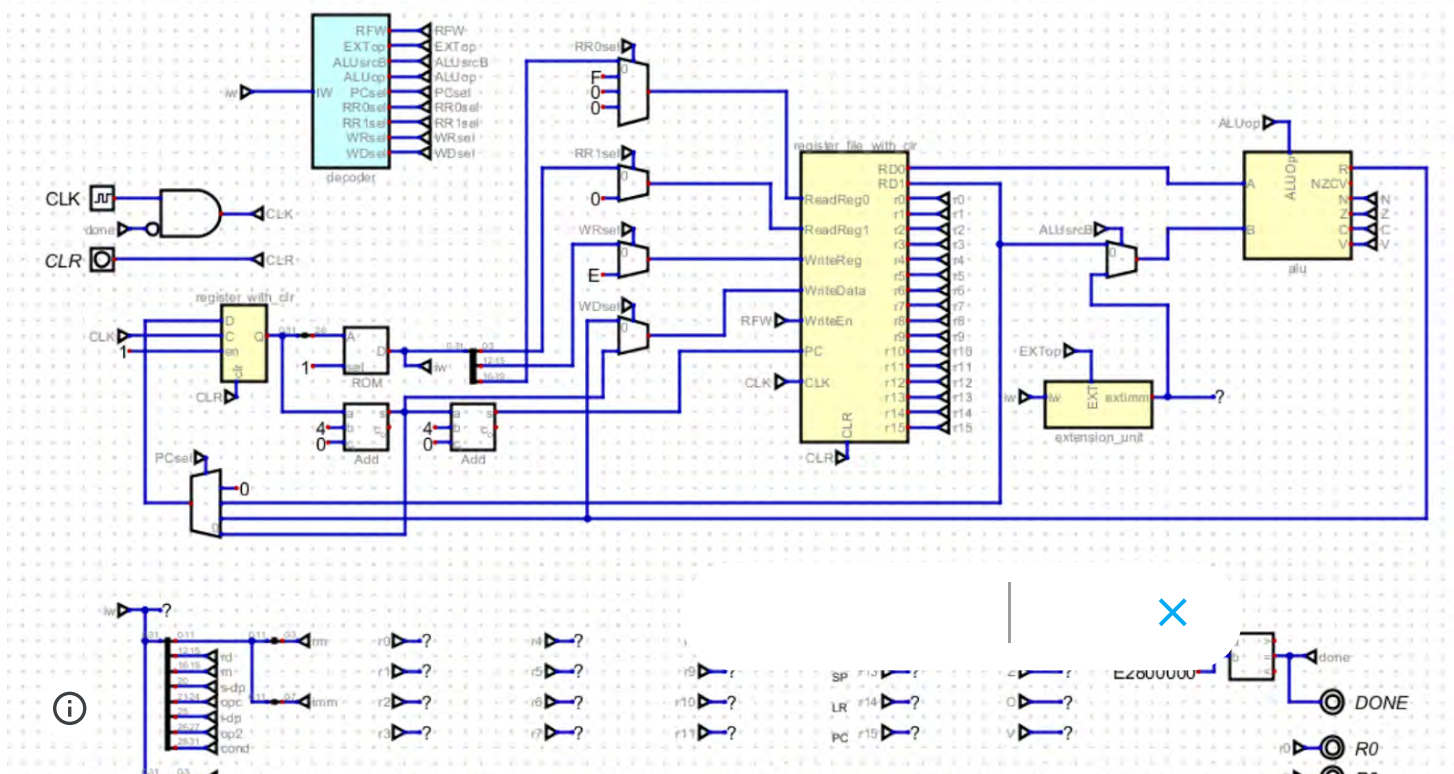
```
first_s:
```

```
add r0, r0, r1
```

```
bx lr
```

To support branch, we need to modify the data path and the control path, as well as the control unit. In the schematic, that's shown with the branch multiplexer choosing PC + 4 or a branch target calculated by the ALU

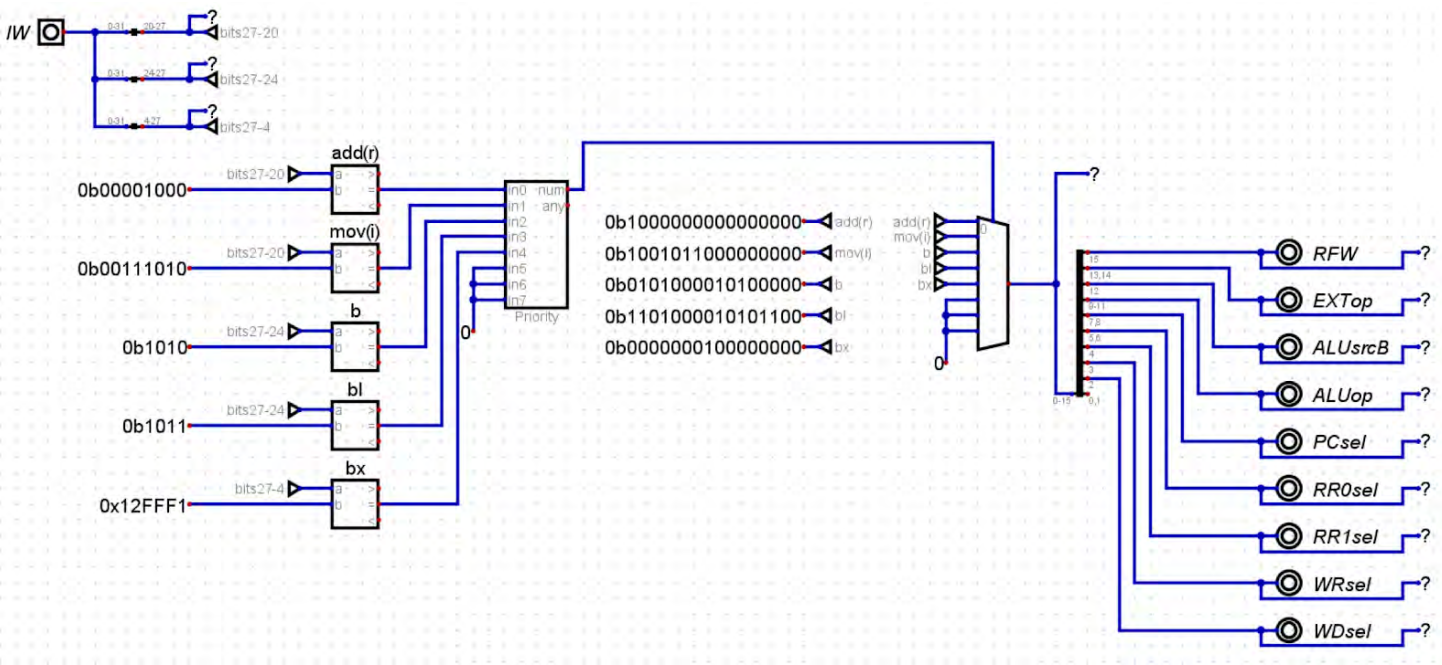
See the following new top-level processor circuit with an updated datapath and control path. Note that we have added several MUXes that allow us to direct traffic. For example, the MUX control by PCsel will choose which value to send to the PC. For normal instructions this will be PC+4, for B and BL this will be the Branch Target Address that is computed using the ALU, and for BX. The MUX controlled by RR0sel now chooses between rn for data processing instructions and 0xF (the PC) for the branch target calculation. The MUX controlled by RR1sel has been added for supporting future instructions not currently supported in this version. The MUX controlled by WRsel chooses between RD and 0xE (LR) for which register to write to and finally the MUX controlled by WDsel chooses between the ALU Result and PC+4 as for input into WriteData.



6	add(i)	0	0	1	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
7	mov(r)	0	0	0	1	1	0	1	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0
8	mov(i)	0	0	1	1	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0
9	b	1	0	1	0	X	X	X	X	0	1	0	1	0	0	0	0	1	0	1	0	0	0	0
10	bl	1	0	1	1	X	X	X	X	1	1	0	1	0	0	0	0	1	0	1	0	1	1	0
11	bx	(a)								0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
12																								
13	(a) Check for BXCDEF 0x12FFF1																							

When we identify branch instructions those will output the PC value on RD0, and choose the next PC value coming from the ALU result on the data path. That result will be the branch target address. BX chooses the value in the register number found in bits 3-0 (rn).

This table adds rows for adding an immediate value and moving a register value. It also adds new control lines for BL and BX. The schematic below shows how to add those inputs and outputs to our Control Unit circuit:



This version of the Control Unit uses tunnels between the circuits which identify the instructions and the circuits which build the outputs. This allows for legible and flexible outputs. We can also build the receiving end of the tunnel in other places in our processor. Notice how we have incrementally evolved the first decoder.

Note the use of a comparator to check for the long constant bit pattern of the BX instruction.



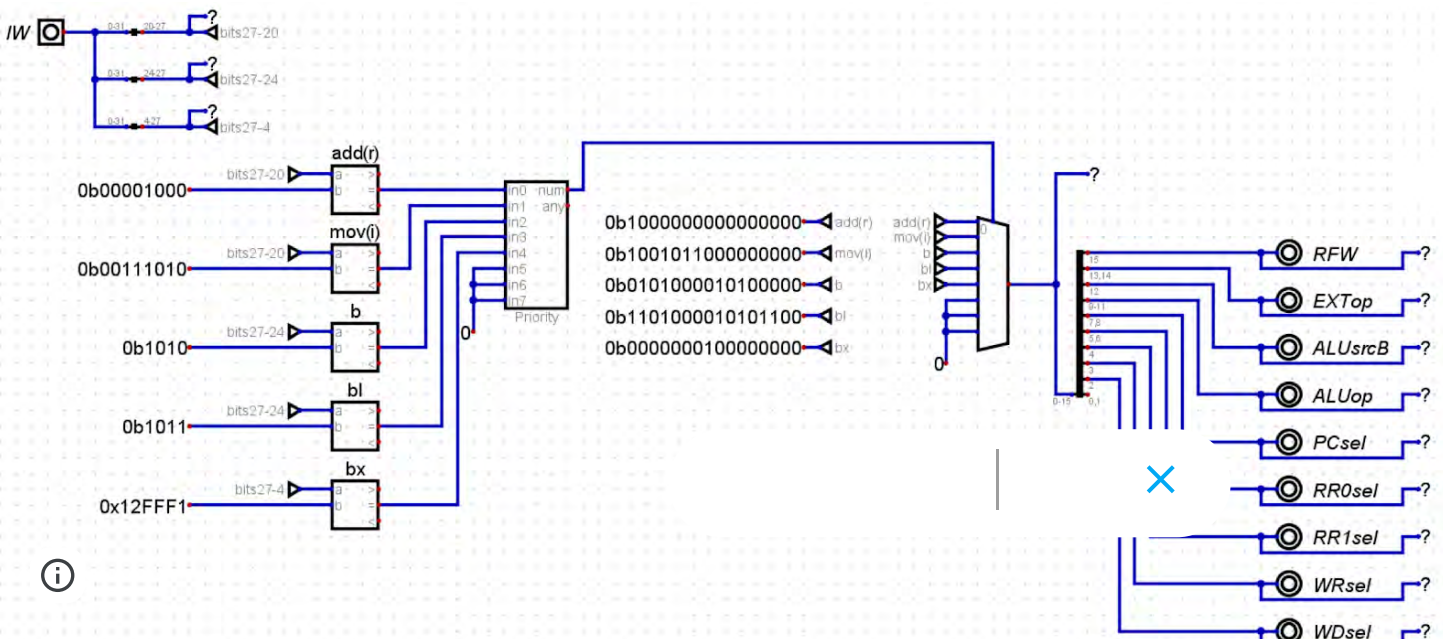
Finally, the constant zero input is temporary scaffolding. Once all the Control Unit outputs have at least one positive input, you won't need to run the zero line.

To support these instructions, we need to also evolve the Decoder (Control Unit) to control the new MUXes and new components. The spreadsheet below shows new inputs and outputs for these instructions. Note that an X input means (don't care), that is it could be 0 or 1. So we can ignore don't care values when detecting an instruction type.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB		
1																														
2																														
3		IW input bits									Control output bits																			
4		27	26	25	24	23	22	21	20		RFW	EXTop1	Extop0	ALUsrcB	ALUOp2	ALUOp1	ALUOp0	PCsel0		PCsel0	RR0sel1	RR0sel0	RR1sel	WRsel	WDsel	P1				
5	add(r)	0	0	0	0	1	0	0	0		1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
6	add(i)	0	0	1	0	1	0	0	0		1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
7	mov(r)	0	0	0	1	1	0	1	0		1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0		
8	mov(i)	0	0	1	1	1	0	1	0		1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0		
9	b	1	0	1	0	X	X	X	X		0	1	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0		
10	bl	1	0	1	1	X	X	X	X		1	1	0	1	0	0	0	0	0	0	1	0	1	0	1	1	0	0		
11	bx	(a)									0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0			
12																														
13	(a) Check for BXCODE 0x12FFF1																													

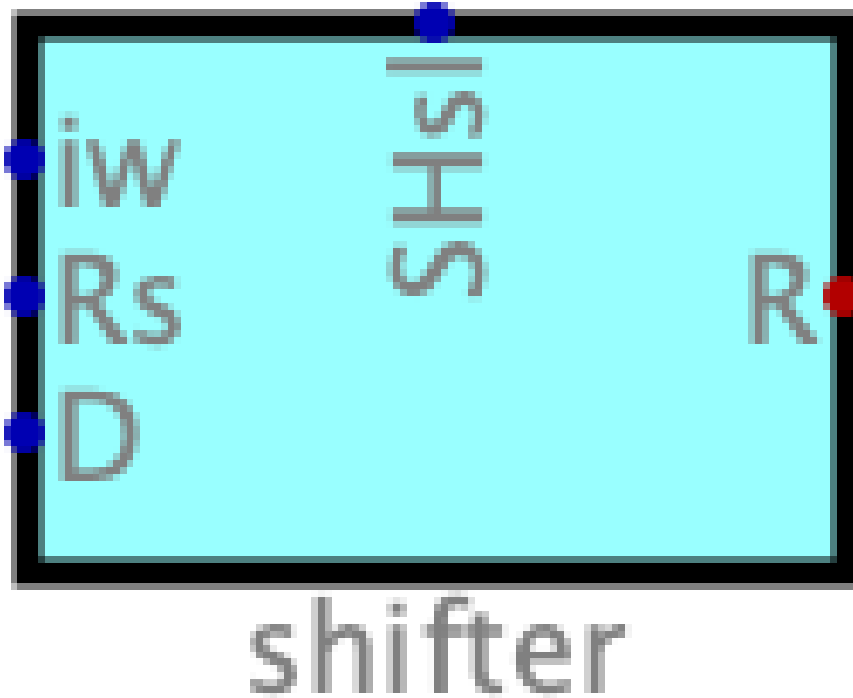
When we identify branch instructions those will output the PC value on RD0, and choose the next PC value coming from the ALU result on the data path. That result will be the branch target address. BX chooses the value in the register number found in bits 3-0 (rn).

This table adds rows for adding an immediate value and moving a register value. It also adds new control lines for BL and BX. The schematic below shows how to add those inputs and outputs to our Control Unit circuit:





Guide to Project06 - Part 3



Shift Unit

To support the implementation of `lsl`, `lsr`, and `asr` as well as `SDT` with a shift, e.g., `ldr r0, [r1, r2, LSL #2]` we will need a shift unit. The shift unit will have three inputs (`iw`, `Rm`, `Rs`, `SHsel`) and one output `R`:

- `iw` - The 32 bit instruction word. We need bits 4-11 which will be extracted in the Shift Unit.
- `D` - The value to be shifted.
- `Rs` - The shift amount if a register shift amount is selected
- `SHsel` - Choose to `D` unchanged or `D` `sr`
- `R` - The output value



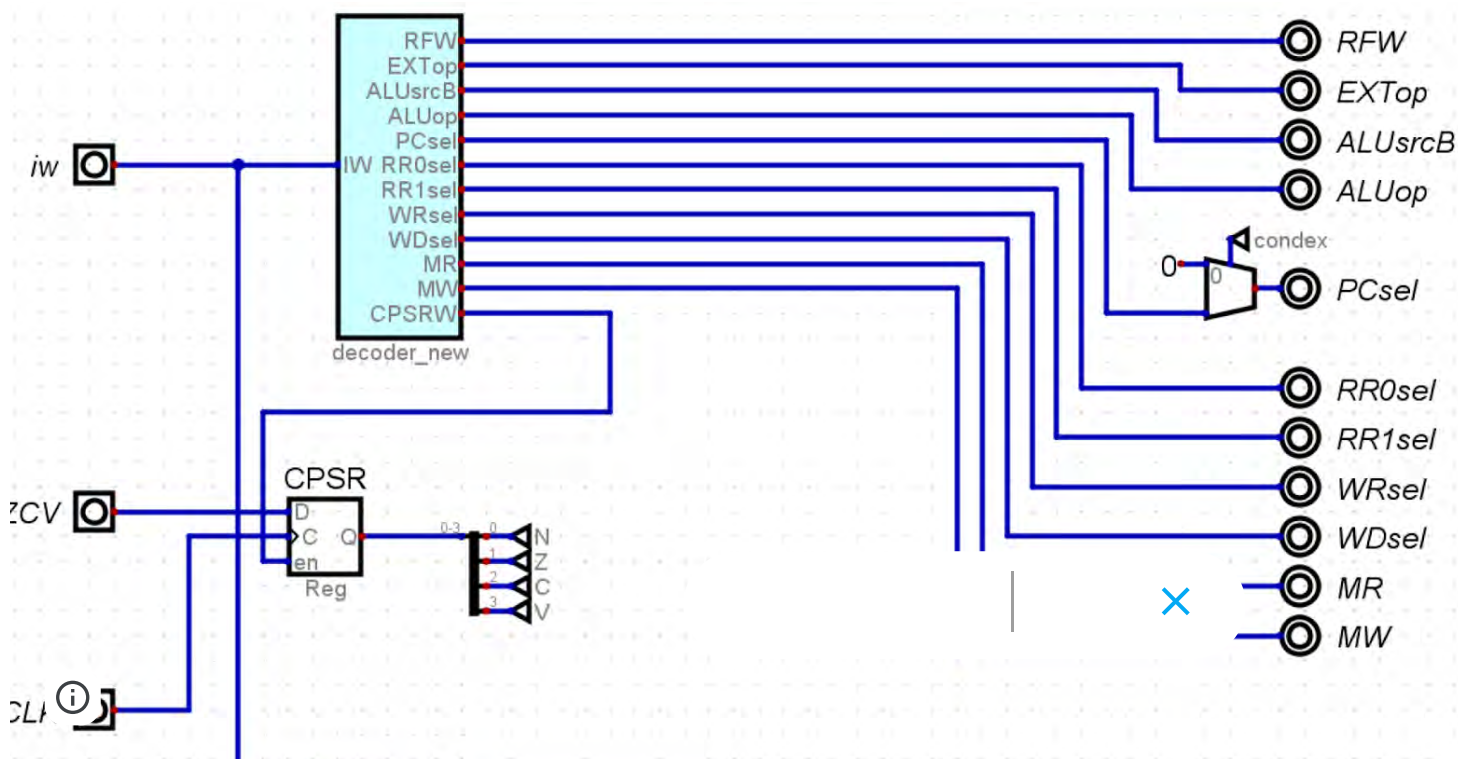
You will put the shift unit in front of the `ALUsrcB` MUX.

- D - The value to be shifted.
- Rs - The shift amount if a register shift amount is selected
- SHsel - Choose to D unchanged or D shifted as determine by bits 4-11.
- R - The output value

You will put the shift unit in front of the ALUsrcB MUX.

Conditional Execution

1. In order to support conditional branch instructions (e.g. BEQ, BNE, BGE, etc.) we will create a Control Unit that embeds our decoder.
2. The Decoder identifies instructions and sets the output control lines.
3. The conditional branch circuitry added below is incomplete as well as the decoder. You will add support for other cases as needed by your Project03 functions.
4. We will use a 4-bit register to store the CPSR bits which are the result of the subtraction operation in the ALU.
5. Recall that although the ARM architecture permits conditional execution of all instructions, you are only required to implement conditional branch instructions.





To support conditional execution:

1. We need a new output from the Main Decoder which determines whether we are executing a CMP instruction
2. When executing a CMP instruction, the EN input to the CPSR register will be set to 1
3. The splitter extracts the COND bits from bits 28-31 of the instruction word
4. The comparators for AL (always) and EQ (if equal) look for the condition codes shown in the Instruction Set Manual
5. If the OR gate outputs 1, then the PCsel output is 1. If PCsel is 1, we will calculate PC+branch offset. If PCsel is 0, we will simply move to the next instruction, PC+4
6. The key intuition here is that to ignore an instruction we simply don't update the state which would have been changed by the instruction.
7. This will be useful if you wish to support conditional execution of other instructions besides branch. The instruction executes, but its results are ignored, as though it never happened.

Using a Decode ROM

In Part 2, we showed how to generate the control bits using a multiplexer fed by constant values containing the control bits as a 15-bit value.


It's more concise to put the control bits in a ROM, and use the output of the Priority Encoder to index the ROM. Therefore each instruction decoded is an address, and the value at that ROM address is the 15-bit value containing the control bits. If your decoder has 32 or less outputs, your ROM will need 5 address bits ($2^5 = 32$). Your ROM will need enough data bits to hold all of the control bits.

To create the data in the ROM, you can copy/paste the concatenated control bits from your spreadsheet into a .hex file, and load that into your ROM. While you could just

type directly into the ROM's data editor, v
work when you add control bits or find bu

fr  at:

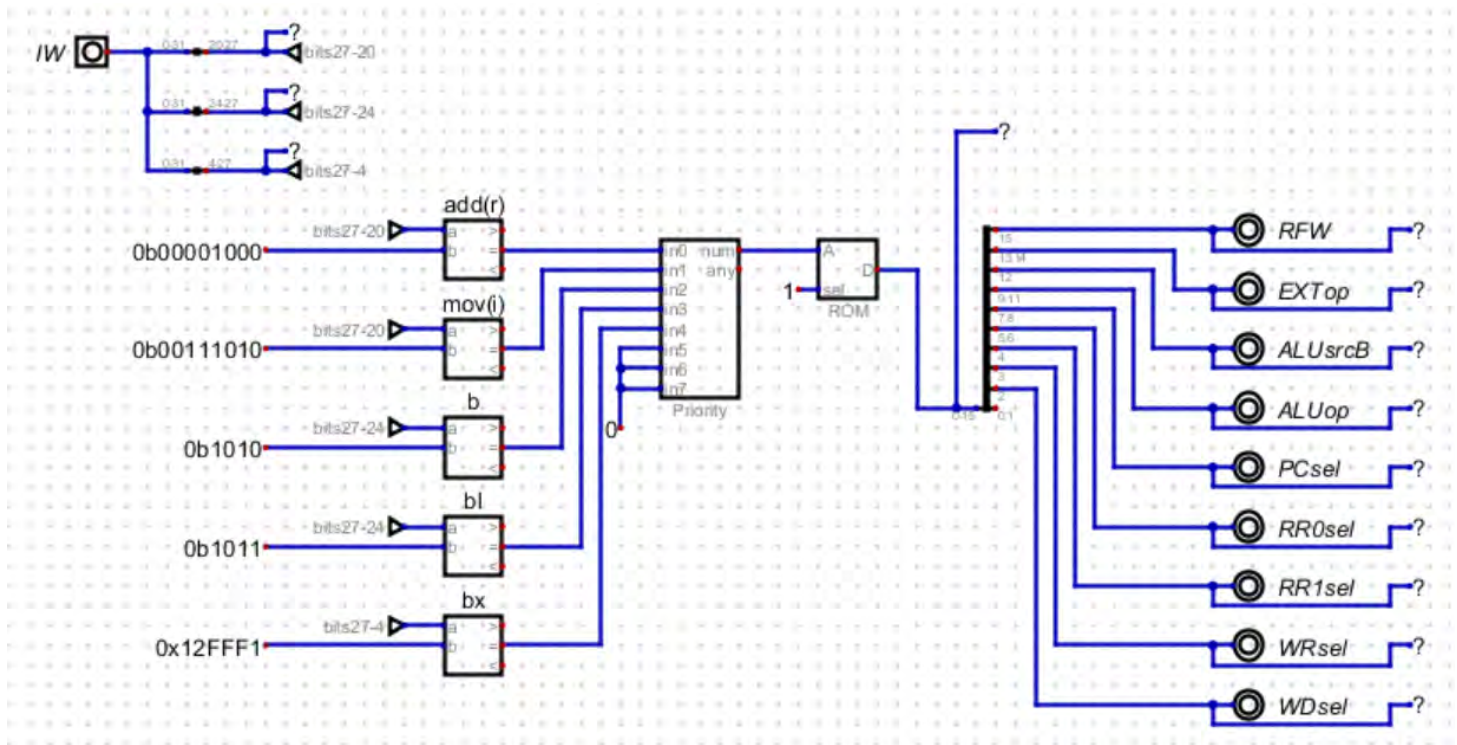
v2.0 raw

 minimize
of hex file

value at that ROM address is the 15-bit value containing the control bits. If your decoder has 32 or less outputs, your ROM will need 5 address bits ($2^5 = 32$). Your ROM will need enough data bits to hold all of the control bits.

To create the data in the ROM, you can copy/paste the concatenated control bits from your spreadsheet into a .hex file, and load that into your ROM. While you could just type directly into the ROM's data editor, we suggest creating a .hex file to minimize work when you add control bits or find bugs in your spreadsheet. Example of hex file format:

```
v2.0 raw
8000
9600
50A0
D0AC
100
```



We provide you with a Python script (mkd.hex) for the decode ROM. First, create a file out control bits from your spreadsheet. You can create the decode ROM hex file like this:

✗ to create the is of the



your spreadsheet into a .hex file, and load that into your ROM. While you could just type directly into the ROM's data editor, we suggest creating a .hex file to minimize work when you add control bits or find bugs in your spreadsheet. Example of hex file format:

```
v2.0 raw
```

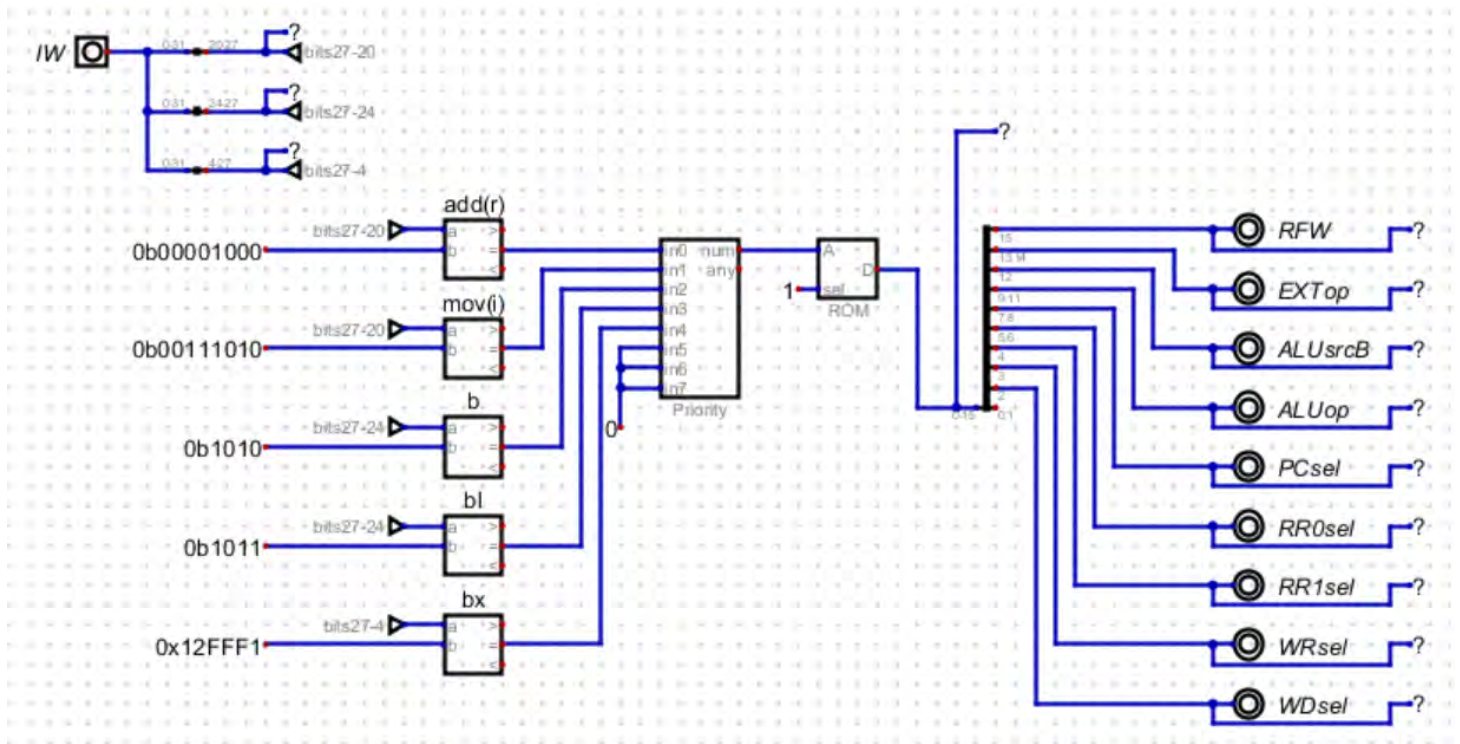
```
8000
```

```
9600
```

```
50A0
```

```
D0AC
```

```
100
```



We provide you with a Python script (mkdecoderom.py) that makes it easy to create the .hex for the decode ROM. First, create a file that contains the binary versions of the output control bits from your spreadsheet. You can create the decode ROM hex file like this:

```
$ python3 mkdecoderom.py > decode_rom.hex
```

```
<paste your output control bits
```

```
^D
```



ARM Processor Implementation

Due

1. Mon Nov 22nd at 11:59 PM to your Project06 Github repo.
2. Submit all of the .dig files and .hex files required to run your simulation
3. Submit a PDF for your Decoder spreadsheet
4. Interactive grading will take place on Tue Nov 23rd. You will sign up for a grading slot using the signup sheet link provided in Piazza or Campuswire.

Requirements


1. You will implement a single-cycle microarchitecture for a subset of the ARMv7 instruction set architecture in Digital
 1. You may use any of Digital's library of components
 2. We will introduce some new and useful tools and techniques for Digital in lecture
2. You need to pass unit tests and complete program tests provided below. You also need to get your Project02 `sort/find_max_index` and your Project03 `merge/merge_sort` implementations. Note that the complete program requirements are the as for Project04. The unit tests will help you incrementally build and test your processor. We will provide a starter instruction memory with the unit tests and the complete program tests from Project04. You will need to add your `sort/find_max_index` and `merge/merge_sort` to the give instruction memory.
3. In order to make your programs runnable on your processor you must do the following:
 1. You will add an assembly language `main` to set up at least five parameters for your functions
 2. You will add the end marker (add `r0` to zero) so that the program should stop
- ⓘ Ensure that you do not have `.global` or `.func` directives in your programs



3. Ensure that you do not have `.global` or `.func` directives in your programs
4. Your processor implementation will include the following major sub-circuits:
 1. The Program Counter (PC) will be one 32-bit register with enable (EN) and clear (CLR) inputs.
 2. Machine code programs will be stored in a ROM components, just as we did in Project05. Like in Project05 your instruction memory will be able to select the program you want to execute.
 3. A Register File that can support reading 3 registers in a single clock cycle: RD0, RD1, and RD2. This is a modified version of the Register File from Lab06 and Lab07.
 4. The Arithmetic Logic Unit (ALU) will perform data processing tasks such as `add`, `sub`, `mul`, and `mov`. The ALU will also compute NZCV values for subtraction (as a result of a `cmp` instruction).
 5. The Sign Extension Unit (Extender) will perform 8-bit zero extension, 12-bit zero extension, and 24 bit sign extension (and multiply by 4).
 6. A Shift Unit (Shifter) will support shifts in `mov` and `SDT` instructions.
 7. Data Memory. We will use a Digital RAM component for data memory. This will be used for stack memory by our test programs. We will configure the RAM component to hold 32 bit word values. This will make it easy to support `ldr` and `str`.
 8. The Control Unit will decode machine code instructions and support conditional branch execution. As with Project05, the Control Unit will be the most complex part.
 9. The Data Path will connect data between the various sub-circuits
 10. The Control Path will connect the Control unit to various sub-circuits and multiplexers

Given



-  We will discuss the major sub-circuits in lecture and you will have hands-on time to develop and ask questions



Given

1. We will discuss the major sub-circuits in lecture and you will have hands-on time to develop and ask questions
2. We have compiled an implementation guide for Project06, available in three parts in the Resources section of this web site.
3. We will provide a starter instruction memory with the unit tests and complete program tests.
4. We will provide a Python script for creating a decode ROM hex file.

Tests

Project06 tests:

<https://github.com/USF-CS315-0304-F21/tests/tree/main/project06>

Project06 given (tests source and instruction memory):

<https://github.com/USF-CS315-0304-F21/project06-given>

Unit Tests

00-dp-reg

01-dp-imm

02-dp-lsl-imm

03-dp-lsr-reg

04-dp-asr-imm

05-mem-reg

06-mem-imm

07-mem-shift-reg

08-blbx

09-b

10-beq

11-ble

12-bgt

13-lui

14-rsb





00-dp-reg

01-dp-imm

02-dp-lsl-imm

03-dp-lsr-reg

04-dp-asr-imm

05-mem-reg

06-mem-imm

07-mem-shift-reg

08-blbx

09-b

10-beq

11-ble

12-bgt

13-mul

14-rsb

Complete Program Tests

15-quadratic

16-get_bitseq

17-max3

18-fib-rec

Your Programs

19-sort_s (and find_max_index_s)

20-merge_sort_s (and merge_s)

Rubric

For interactive grading you should be able to run the autograder tests and manually execute your sort_s and merge_sort_s programs.

(30 points) Unit tests

(20 points) Complete program tests

(10 points) Your sort_s

