Project 3

Due September 2, 2024 at 9:00 PM

This project description is subject to change at any time for clarification. For this project you will be working with a partner.

Desired Outcomes

- Exposure to using circuit simulator (Logisim Evolution)
- Exposure to interrupt/system call mechanism
- An understanding of how carry-look ahead is implemented
- An understanding of how to develop a simple RISC style CPU

Project Description

You will be using Logisim Evolution for this project. You may use any of the built in components of Logisim, except for those in the Arithmetic group. All class projects will be run through MOSS like software to determine if students have excessively collaborated. Excessive collaboration, or failure to list external sources will result in the matter being referred to Student Judicial Affairs.

You will be developing a two cycle (Fetch-Decode, Execute-Writeback) 15-bit RISC-V like CPU for this project. The CPU will contain 8, 15-bit general purpose registers R0 – R7 (though R0 is hardwired to zero). A 15-bit program counter PC, 15-bit instruction buffer IB, 15-bit save restore program counter, an 8-bit flags register, and an 8-bit save restore flags register. The flags register has two empty bit, and six flags: Interrupt I, Zero Z, Negative N, Overflow O, Carry C, Interrupt Enable E. All instructions are 15-bits and are described in the following section.

CPU Inputs:

- CLK – The CPU Clock
- IRQ – Interrupt request
- DATAIN – The 15-bit data path in from memory and I/O

CPU Outputs:

- ADDR – The 15-bit memory address to be read or written
- RE – The read enable to memory, high when data is to be read from memory
- WE – The write enable to memory, high when data is to be written to memory
- DATAOUT – The 15-bit data path out to memory and I/O, this should be tri-state buffered

CPU Debug Outputs:

- PC – The 15-bit program counter
- IB – The 15-bit instruction buffer
- SRP – The 15-bit save restore program counter
- SRF – The 8-bit save restore flags register
- R0 – R7 – The 8, 15-bit general purpose registers

- I – Interrupt flag
- Z – Zero flag
- N – Negative flag
- O – Overflow flag
- C – Carry flag
- E – Interrupt Enable flag

The flags register layout is shown below.

Bit	7	6	5	4	3	2	1	0
Value	-*	I	E	-*	O	C	N	Z

- Z – Zero flag
- N – Negative flag
- C – Carry flag
- O – Overflow flag
- E – Interrupt Enable flag
- I – Interrupt flag
- * – Hardwired Zero

Your CPU will be connected to a small memory that will be used for both data and instructions. You will be provided several small assembly programs, an assembler, and a given test circuit in order to test your CPU. The adder for your ALU must be implemented using multiple 5-bit carry look-ahead units or built as a prefix adder.

When an interrupt occurs (either IRQ with E flag set, or SWI instruction), during the writeback stage:

- The address 0x7FFF from memory updates the PC
- The E flag is cleared
- The SRP is updated with PC for IRQ interrupt or PC + 1 for SWI
- The SRF is updated with the flags register

An assembler has been provided as a Python 3 script. There are several assembly programs that have been provided for you to test. The output dat files from the assembler can be loaded into the memory to execute.

To receive full credit your CPU does not need to implement all the instructions/functionality. The required instructions that must be implemented are R-Type instructions (ADD, SUB, AND, OR, XOR, SLL, SRL, and SRA), I-Type Instructions (ADDI, LW, ANDI, ORI, XORI, SLLI, SRLI, SRAI, and JALR), B-Type Instructions (BZF, BNZF, BNF, BNNF, BCF, and BNCF), S-Type Instruction (SW), and U-Type Instruction (LUI). The interrupt and flag related instructions RTI, SWI, SSRP, SSRF, SF, and SFI can be implemented for extra credit. As the supporting instructions are extra credit, the proper response to interrupts will also be extra credit.

You can unzip the given tgz file with utilities on your local machine, or if you upload the file to the CSIF, you can unzip it with the command:

```
tar -xzvf proj3given.tgz
```

You **must** submit the circuit file, README.md file, and `.git` directory in a tgz archive. You can tar gzip a directory with the command:

```
tar -zcvf archive-name.tgz directory-name
```

You should avoid using existing circuits as a primer that are currently available on the Internet. You **MUST** specify in your README.md file any sources of circuits that you have viewed to help you complete this project. Your README file **MUST** have both partner's name and SID number, a brief description of what circuits work/don't work, and a list of sources you used for designing of your circuit (you do not need to list the book or lecture notes it is assumed these have been used). You **MUST** properly document **ALL** uses of Generative AI following the guidelines outlined in the Generative AI Restrictions. All class projects will be submitted to MOSS like software to determine if students have excessively collaborated. Excessive collaboration, or failure to list external code sources will result in the matter being referred to Student Judicial Affairs.

Grading

The point breakdown can be seen in the table below. Make sure your circuit executes correctly in Logisim Evolution 3.8.0 as that is where it is expected to execute. You will make an interactive grading appointment to have your assignment graded. You must have a working webcam for the interactive grading appointment. Project submissions received 24hr prior to the due date/time will received 10% extra credit. The extra credit bonus will drop off at a rate of 0.5% per hour after that, with no additional credit being received for submissions within 4hr of the due date/time.

Points	Description
10	Has git repository with appropriate number of commits
10	Has README.md with proper documentation
10	ADDER15 implemented as specified
10	ALU15 implemented as specified
5	REGS15 implemented as specified
5	IMM15 implemented as specified
5	FLAGS8 implemented as specified
10	R-Type instructions execute correctly
5	I-Type instruction execute correctly
10	B-Type instructions execute correctly
10	S-Type and U-Type instructions execute correctly
10*	Student understands all circuits they have provided
Extra Credit	
10	Interrupt instructions and behavior is correct

* Students who are unable to demonstrate understanding of their circuit could receive negative points and resulting in score as low as zero overall regardless of functioning of circuit submitted.

** Groups where partner workload is not balanced may have adjustments in their scores.

An approach you may want to take for this project is as follows:

1. Implement the 15-bit adder by either starting with the 5-bit carry look-ahead or doing a full prefix adder. The ADDER15 subcircuit interface has been provided.
2. Implement the 15-bit ALU using the adder from step 1. The ALU15 subcircuit interface has been provided.
3. Implement register file. The REGS15 subcricuit interface has been provided. The I/O are:
    a. CLK – the clock signal input
    b. SelD – selector for write target register
    c. W – enable for the write update
    d. D – 15-bit input for the write value
    e. A, B – two 15-bit output ports
    f. SelA, SelB – two selectors for the register outputs
    g. R0 – R7 – outputs of the current register values
4. Implement immediate generator. The IMM15 has been provided for you. This will take in the instruction and create the sign extended immediate value.
5. Create a CPU that will read an instruction into the IB based upon the PC, and then does nothing on the writeback stare. The PC should be incremented in the writeback stage so that the next memory address will be read. This will guarantee that the loading and incrementing of the PC is correct. You can use the rtype0.asm program to test this as it is essentially NOP instructions.
6. Implement part of the decoder to support R-Type and I-Type instructions. The DECODER15 interface has been provided. The I/O are:
    a. OPCODE – the 5-bit opcode of the instruction
    b. SWI – output that is true if the opcode is SWI
    c. STORE – output that is true if the opcode is SW
    d. LOAD – output that is true if the opcode is LW
    e. ASEL – output that selects the input for the ALU A input
    f. BSEL – output that selects the input for the ALU B input
    g. ALUOP – outputs the ALU operation
    h. REGDATASRC – outputs the source for the register write
    i. REGW – outputs true if the register should be written back
    j. FLAGSRC – outputs the source for the flags register
    k. FLAGWM – outputs the flags write mask
    l. SRPW – outputs true if the SRP register should be updated
    m. SRFW – outputs true if the SRF register should be updated
7. Add the ALU, register file, immediate generator, and decoder to allow for register/register instructions, and for loading immediate values. Chapter 7.3 may be helpful in understanding how the components interact. You should test the rtype0.asm, itype.asm, and loading.asm programs for this. These will test general instructions without changing control flow.
8. Implement the flags register. The FLAGS8 subscircuit interface has been provided. The FLAGS8 interface has been provided. The inputs are:
    a. IRQ – the IRQ input
    b. NF – the 8-bit next flag input
    c. WM – the 8-bit write mask input
    d. CLK – the clock signal

   e. F – the 8-bit flags output
9. Expand CPU to implement the branching and jump instructions. You should test the sinwave.asm and then the subroutine.asm programs. These will test that the branching/jumping is working as well as the basic ALU capability.
10. (EXTRA CREDIT) Implement the interrupts, interrupt instructions, and swap register instructions. The interrupt.asm program will test the interrupt support.

Table 1. General Instruction Format

Type	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	-	rs2			rs1			rd			opcode				
I	imm[3:0]				rs1			rd			opcode				
B	imm[9:0]										opcode				
S	imm[3]	rs2			rs1			imm[2:0]			opcode				
J	imm[6:0]							rd			opcode				
U	imm[10:4]							rd			opcode				

Table 2. Individual Instruction Format

Inst	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD	-	rs2			rs1			rd			00000				
SUB	-	rs2			rs1			rd			00001				
AND	-	rs2			rs1			rd			00010				
OR	-	rs2			rs1			rd			00011				
XOR	-	rs2			rs1			rd			00100				
SLL	-	rs2			rs1			rd			00101				
SRL	-	rs2			rs1			rd			00110				
SRA	-	rs2			rs1			rd			00111				
ADDI	imm[3:0]				rs1			rd			01000				
LW	imm[3:0]				rs1			rd			01001				
ANDI	imm[3:0]				rs1			rd			01010				
ORI	imm[3:0]				rs1			rd			01011				
XORI	imm[3:0]				rs1			rd			01100				
SLLI	imm[3:0]				rs1			rd			01101				
SRLI	imm[3:0]				rs1			rd			01110				
SRAI	imm[3:0]				rs1			rd			01111				
BZF	imm[9:0]										10000				
BNZF	imm[9:0]										10001				
BNF	imm[9:0]										10010				
BNNF	imm[9:0]										10011				
BCF	imm[9:0]										10100				
BNCF	imm[9:0]										10101				
RTI	-										10110				
SWI	-										10111				
SSRP	-				rs1			rd			11000				
SSRF	-				rs1			rd			11001				
SF	-				rs1			rd			11010				
JALR	imm[3:0]				rs1			rd			11011				
JAL	imm[6:0]							rd			11100				
SFI	imm[6:0]							rd			11101				
SW	imm[3]	rs2			rs1			imm[2:0]			11110				
LUI	imm[10:4]							rd			11111				

Table 3. Pseudo Instructions

Pseudo Instruction	Base Instruction	Meaning
`NOP`	`ADDI x0, x0, 0`	No operation
`LI rd, imm`	*`Multiple Sequences`*	Load immediate
`MV rd, rs`	`ADDI rd, rs, 0`	Copy registers
`NOT rd, rs`	`XORI rd, rs, -1`	One's complement
`NEG rd, rs`	`SUB rd, x0, rs`	Two's complement
`BEQ rs1, rs2 target`	`SUB x0, rs1, rs2` `BZF target`	Branch if rs1 = rs2
`BNE rs1, rs2 target`	`SUB x0, rs1, rs2` `BNZF target`	Branch if rs1 ≠ rs2
`BLT rs1, rs2 target`	`SUB x0, rs1, rs2` `BNF target`	Branch if rs1 < rs2
`BGE rs1, rs2 target`	`SUB x0, rs1, rs2` `BNNF target`	Branch if rs1 ≥ rs2
`BLTU rs1, rs2 target`	`SUB x0, rs1, rs2` `BCF target`	Branch if rs1 < rs2, unsigned
`BGEU rs1, rs2 target`	`SUB x0, rs1, rs2` `BNCF target`	Branch if rs1 ≥ rs2, unsigned
`J target`	`JAL x0, target`	Jump
`JR rs`	`JALR x0, rs`	Jump register

ADD (Add)

Description

Add, adds two registers together and stores the value into a destination register.

```
ADD rd, rs1, rs2
```

```
rs1 + rs2 → rd
```

```
PC + 1 → PC
```

Flags Affected: Z, N, O, and C.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD	-	rs2			rs1			rd			00000				

SUB (Subtract)

Description

Subtract, subtracts register from another and stores the value into a destination register.

```
SUB rd, rs1, rs2
```

```
rs1 - rs2 → rd
```

```
PC + 1 → PC
```

Flags Affected: Z, N, O, and C.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SUB	-	rs2			rs1			rd			00001				

AND (And)

Description

And, ands two registers together and stores the resulting value back into a destination register.

`AND rd, rs1, rs2`

`rs1 & rs2 → rd`

`PC + 1 → PC`

Flags Affected: `Z`, and `N`.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AND	-	rs2			rs1			rd			00010				

OR (Or)

Description

Or, ors two registers together and stores the resulting value back into a destination register.

`OR rd, rs1, rs2`

`rs1 | rs2 → rd`

`PC + 1 → PC`

Flags Affected: `Z`, and `N`.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OR	-	rs2			rs1			rd			00011				

XOR (Or)

Description

Xor, xors two registers together and stores the resulting value back into a destination register.

```
XOR rd, rs1, rs2
```

```
rs1 ^ rs2 → rd
```

```
PC + 1 → PC
```

Flags Affected: `Z`, and `N`.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
XOR	-	rs2			rs1			rd			00100				

SLL (Shift Left Logical)

Description

Shift left logical, rotates the bits of the source register `rs1` by the number of bits specified by the low 4 bits of `rs2` and stores the value into the destination register. Zeros are shifted in, and the most significant bits are shifted out the carry bit.

```
SLL rd, rs1, rs2
```

```
rs1 << rs2 → rd
```

```
PC + 1 → PC
```

Flags Affected: `Z`, `N`, `C`.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SLL	-	rs2			rs1			rd			00101				

SRL (Shift Right Logical)

Description

Shift right logical, rotates the bits of the source register `rs1` by the number of bits specified by the low 4 bits of `rs2` and stores the value into the destination register. Zeros are shifted in, and the least significant bits are shifted out the carry bit.

```
SLL rd, rs1, rs2

rs1 >> rs2 → rd

PC + 1 → PC
```

Flags Affected: `Z`, `N`, `C`.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRL	-	rs2			rs1			rd			00110				

SRA (Shift Right Arithmetic)

Description

Shift right arithmetic, rotates the bits of the source register `rs1` by the number of bits specified by the low 4 bits of `rs2` and stores the value into the destination register. The most significant bit is maintained, and the least significant bits are shifted out the carry bit.

```
SLL rd, rs1, rs2

rs1 >> rs2 → rd

PC + 1 → PC
```

Flags Affected: `Z`, `N`, `C`.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRL	-	rs2			rs1			rd			00111				

ADDI (Add Immediate)

Description

Add Immediate, adds an immediate 4-bit signed extended value to a register and stores it back into a destination register.

```
ADDI rd, rs1, imm

rs1 + imm → rd

PC + 1 → PC
```

Flags Affected: `Z`, `N`, `O`, and `C`.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDI	imm[3:0]				rs1			rd			01000				

LW (Load Word)

Description

Load word, loads data from main memory into a register. The address of memory is specified by the register `rs1`. A 4-bit signed extended immediate offset `imm` is added to the address of the source.

```
LW rd, rs1 + imm

Mem(rs1 + imm) → rd

PC + 1 → PC
```

Flags Affected: None

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LW	imm[3:0]				rs1			rd			01001				

ANDI (And Immediate)

Description

And immediate, ands an immediate 4-bit signed extended value to a register and stores it back into a destination register.

```
AND rd, rs1, imm

rs1 & imm → rd

PC + 1 → PC
```

Flags Affected: Z, and N.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ANDI	imm[3:0]				rs1			rd			01010				

ORI (Or Immediate)

Description

Or, ors an immediate 4-bit signed extended value to a register and stores it back into a destination register.

```
ORI rd, rs1, imm

rs1 | imm → rd

PC + 1 → PC
```

Flags Affected: Z, and N.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OR	imm[3:0]				rs1			rd			01011				

XORI (Xor Immediate)

Description

Xor immediate, xors an immediate 4-bit signed extended value to a register and stores it back into a destination register.

```
XOR rd, rs1, imm

rs1 ^ imm → rd

PC + 1 → PC
```

Flags Affected: `Z`, and `N`.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
XORI	-	rs2			rs1			rd			01100				

SLLI (Shift Left Logical Immediate)

Description

Shift left logical immediate, rotates the bits of the source register `rs1` by the number of bits specified by the 4-bit immediate `imm` and stores the value into the destination register. Zeros are shifted in, and the most significant bits are shifted out the carry bit.

```
SLL rd, rs1, imm

rs1 << imm → rd

PC + 1 → PC
```

Flags Affected: `Z`, `N`, `C`.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SLLI	imm[3:0]				rs1			rd			01101				

SRLI (Shift Right Logical Immediate)

Description

Shift right logical immediate, rotates the bits of the source register `rs1` by the number of bits specified by the 4-bit immediate value `imm` and stores the value into the destination register. Zeros are shifted in, and the least significant bits are shifted out the carry bit.

```
SRLI rd, rs1, imm
```

```
rs1 >> imm → rd
```

```
PC + 1 → PC
```

Flags Affected: `Z`, `N`, `C`.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRLI	imm[3:0]				rs1			rd			01110				

SRAI (Shift Right Arithmetic Immediate)

Description

Shift right arithmetic immediate, rotates the bits of the source register `rs1` by the number of bits specified by the 4-bit immediate value `imm` and stores the value into the destination register. The most significant bit is maintained, and the least significant bits are shifted out the carry bit.

```
SRAI rd, rs1, imm
```

```
rs1 >> imm → rd
```

```
PC + 1 → PC
```

Flags Affected: `Z`, `N`, `C`.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRAI	imm[3:0]				rs1			rd			01111				

BZF (Branch Zero Flag)

Description

Branch zero flag, branches the Program Counter `PC` to the relative value of the immediate offset if the zero flag is set; otherwise, the Program Counter is incremented by one.

```
BZF #imm

IF Z-flag set PC + imm → PC

ELSE PC + 1 → PC
```

Flags Affected: None.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BZF	imm[9:0]										10000				

BNZF (Branch Not Zero Flag)

Description

Branch not zero flag, branches the Program Counter `PC` to the relative value of the immediate offset if the zero flag is not set; otherwise, the Program Counter is incremented by one.

```
BNZF #imm

IF Z-flag clear PC + imm → PC

ELSE PC + 1 → PC
```

Flags Affected: None.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BNZF	imm[9:0]										10001				

BNF (Branch Negative Flag)

Description

Branch negative flag, branches the Program Counter `PC` to the relative value of the immediate offset if the negative flag is set; otherwise, the Program Counter is incremented by one.

```
BNF #imm

IF N-flag set PC + imm → PC

ELSE PC + 1 → PC
```

Flags Affected: None.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BNF	imm[9:0]										10010				

BNNF (Branch Not Negative Flag)

Description

Branch not negative flag, branches the Program Counter `PC` to the relative value of the immediate offset if the negative flag is not set; otherwise, the Program Counter is incremented by one.

```
BNNF #imm

IF N-flag clear PC + imm → PC

ELSE PC + 1 → PC
```

Flags Affected: None.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BNNF	imm[9:0]										10011				

BCF (Branch Carry Flag)

Description

Branch carry flag, branches the Program Counter `PC` to the relative value of the immediate offset if the carry flag is set; otherwise, the Program Counter is incremented by one.

```
BCF #imm

IF C-flag set PC + imm → PC

ELSE PC + 1 → PC
```

Flags Affected: None.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BCF	imm[9:0]										10100				

BNCF (Branch Not Carry Flag)

Description

Branch not carry flag, branches the Program Counter `PC` to the relative value of the immediate offset if the carry flag is not set; otherwise, the Program Counter is incremented by one.

```
BNCF #imm

IF C-flag clear PC + imm → PC

ELSE PC + 1 → PC
```

Flags Affected: None.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BNCF	imm[9:0]										10101				

RTI (Return From Interrupt)

Description

RTI returns from an interrupt. The PC is updated with the value of the SRP, and the flags are updated with the value of SRF.

```
RTI

SRP → PC

SRF → F
```

Flags Affected: All.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RTI	-										10110				

SWI (Software Interrupt)

Description

SWI generates a software interrupt that will be processed regardless of the E flag value. The interrupt vector is stored in memory location 0x7FFF (maximum address) and is fetched from the memory location to be loaded into the PC. The PC + 1 (address of the instruction after SWI) is stored in the SRP, the current flags register is stored in SRF, and the E flag is cleared.

```
SWI

Mem(0x7FFF) → PC

PC + 1 → SRP

F → SRF
```

Flags Affected: E.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SWI	-										10111				

SSRP (Swap Save Restore PC)

Description

Swap save restore `PC`, moves the source register into the `SRP`, and moves the `SRP` register into the destination register.

```
SSRP rd, rs1
```

```
rs1 → SRP
```

```
SRP → rd
```

```
PC + 1 → PC
```

Flags Affected: None.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SSRP	-				rs1			rd			11000				

SSRF (Swap Save Restore Flags)

Description

Swap save restore flags, moves the low byte of the source register into the `SRF` register, and moves the `SRF` register into the low byte of the destination register.

```
SSRF rd, rs1
```

```
rs1_7..0 → SRF
```

```
SRF → rd_7..0
```

```
PC + 1 → PC
```

Flags Affected: None.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SSRF	-				rs1			rd			11001				

SF (Swap Flags)

Description

Swap flags, moves the low byte of the source register into the flags register, and moves the flags register into the low byte of the destination register.

`SF rd, rs1`

$F \rightarrow rd_{7..0}$

$rs1_{7..0} \rightarrow F$

`PC + 1 → PC`

Flags Affected: All.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SF	–				rs1			rd			11010				

JALR (Jump And Link Register)

Description

Jump and link register, sets the Program Counter `PC` to the value specified in the register `rs1` plus the 4-bit sign extended immediate `imm`. and stores the previous `PC` + 1 in the destination register.

`JALR rd, rs1 + imm`

`rs1 + imm → PC`

`PC + 1 → rd`

Flags Affected: None.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JALR	imm[3:0]				rs1			rd			11011				

JAL (Jump And Link)

Description

Jump and link, sets the Program Counter `PC` to the offset specified byt the in the register `rs1` plus the 7-bit sign extended immediate `imm`. and stores the previous `PC` + 1 in the destination register.

`JAL rd, imm`

`PC + imm → PC`

`PC + 1 → rd`

Flags Affected: None.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JAL	imm[6:0]							rd			11100				

SFI (Swap Flags Immediate)

Description

Swap flags immediate, moves the immediate value into the flags register, and moves the flags register into the low byte of the destination register.

`SF rd, imm`

`F → rd`$_{7..0}$

`imm → F`

`PC + 1 → PC`

Flags Affected: None.

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SFI	imm[6:0]							rd			11101				

SW (Store Word)

Description

Store word, stores data from a register to main memory. The address of memory is specified by the source register `rs1`. A 4-bit signed extended immediate offset `imm` is added to the address of the source.

```
SW rs2, rs1 + imm

rs2 → Mem(rs1 + imm)

PC + 1 → PC
```

Flags Affected: None

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SW	imm[3]	rs2			rs1			imm[2:0]			11110				

LUI (Load Upper Immediate)

Description

Load Upper Immediate loads a sign extended immediate into the register specified by rd.

```
LUI rd, imm

imm → rd

PC + 1 → PC
```

Flags Affected: None

Opcode

Bit	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JAL	imm[10:4]							rd			11111				