PROJECT 2

Title: General Purpose Processor
Point Value: 100
Due Date: 4/4/2021 @ 11.59pm

The Problem

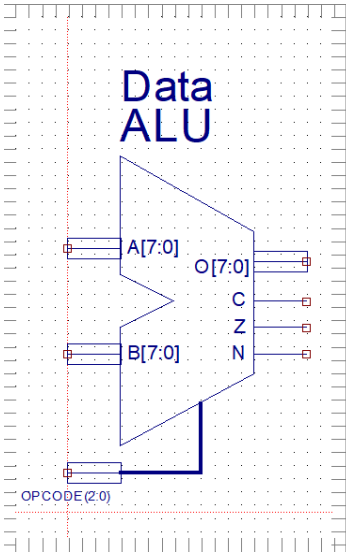
In this project, you will be completing the design of a simple general purpose processor using Xilinx ISE Schematic Capture.

DISCLAIMER: The processor we are working with is a very basic 8 bit single cycle processor with no data memory. While it is a good representation of a general purpose processor and is good for education, commercial processors are typically more complex.

A skeleton has been provided that contains various aspects of the processor already designed. You will be completing the Arithmetic Logic Unit, Register File and Instruction Memory of the simple processor.
Below is a description of these units as it pertains to this processor. You will need to use these descriptions as well as the information in the architecture section to fully build these units.

Arithmetic Logic Unit (ALU):

As covered in class, the arithmetic unit does the arithmetic and logic lifting for the processor. This ALU is an 8 bit ALU and can do 8 different operations as described in the table below. The ALU was previously built by you in project 1. In this project you will be using the hardware you built in project 1 by moving it into the space for the ALU. You can use the method shown in the video regarding sharing files.
You will also need to connect the carry out from the adder, the borrow out from the subtractor and multiplier overflow signals to create the C (carry) flag.



Opcode	Operation
000	O = B
001	O = A ^ B (bitwise xor)
010	O = A AND B (bitwise and)
011	O = A OR B (bitwise or)
100	O = A + B (addition)
101	O = A - B (subtract)
110	O(7) = 0, 0(6:0) = B (7:1) (logical shift right)
111	O = A * B (multiplication)

In addition to the operations, the ALU will also help generate flags based on the output. You do not have to create the flags **but will need to create the input values for the carry flag multiplexor.**

ALU inputs:

A(7:0):
8 bit data to be operated on. This comes from the register file, read data a(R_data_a).

B (7:0):
8 bit data to be operated on. This can come from the register file, read data b (R_data_b) or the data input to the processor.

Opcode (2:0):
This picks the operation to be done. The opcode comes from instruction bits 6 down to 4.

ALU Outputs:

O (7:0):
The result of the ALU operation.

C:
The carry flag, generated when we have over flow from the add, subtract or multiply operations. Connect your carry out, borrow out and multiplier overflow signals to the corresponding input on the C flag multiplexor.

N:
The sign flag, generated when we have a negative result i.e if the most significant bit is a 1. You do not have to create this.

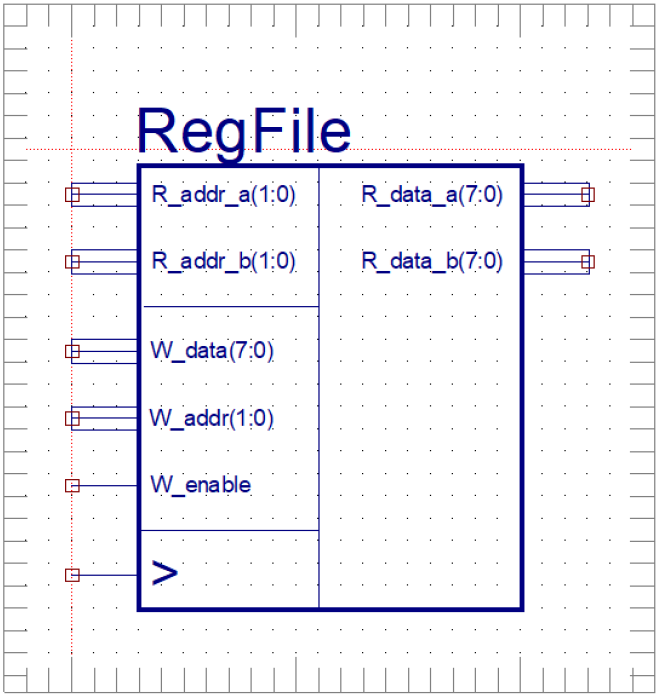
Z:
Zero flag, generated when we have a zero result. You do not have to create this.

To add your Project 1 ALU to Project 2:

1. Navigate to the folder where your project 1 ALU is located.
2. Copy all the schematic and symbol files (.sch and .sym) of the components you created and used in the project 1 ALU and paste them in the project 2 skeleton folder.
3. Next open the Project 2 Skeleton in XIlinx.
4. Right click on the File that is currently the Top Module. (As if you were going to create a new source)
5. Click "Add Source"
6. Navigate to the folder where your project 2 skeleton Add all the .sch files you used to create your project 1 ALU.
7. Then recreate your project 1 ALU in the project 2 ALU schematic.

Register File:

The register file for this processor contains our general purpose registers and is made of one write port, a bank of 4 registers (r0-r3) and two read ports. These will be used by the processor instructions to hold data as it is operated on.



Inputs:

Write Address (W_addr [1:0]):

The 2 bit address for the register to write data to after an instruction.
This address is also the same as the read address a (R_addr_a).

Write data (W_data[7:0]):
Data to be written to a register described by the write address.

Write Enable (W_enable):
Enables saving the write data to the write register. This ensures that we only save data in a register during an instruction that actually needs to save data. For example an add instruction needs to save data but a jump instruction does not need to save any data

Read address a (R_addr_a[1:0]):
2 bit address of register to read data from. This is also the write address.

Read address b (R_addr_b[1:0]):
Second 2 bit address of register to read data from.

Clock (>):
Clock input for the register file. This clock is tied to all the registers and should be the same clock for the entire processor.

Outputs:

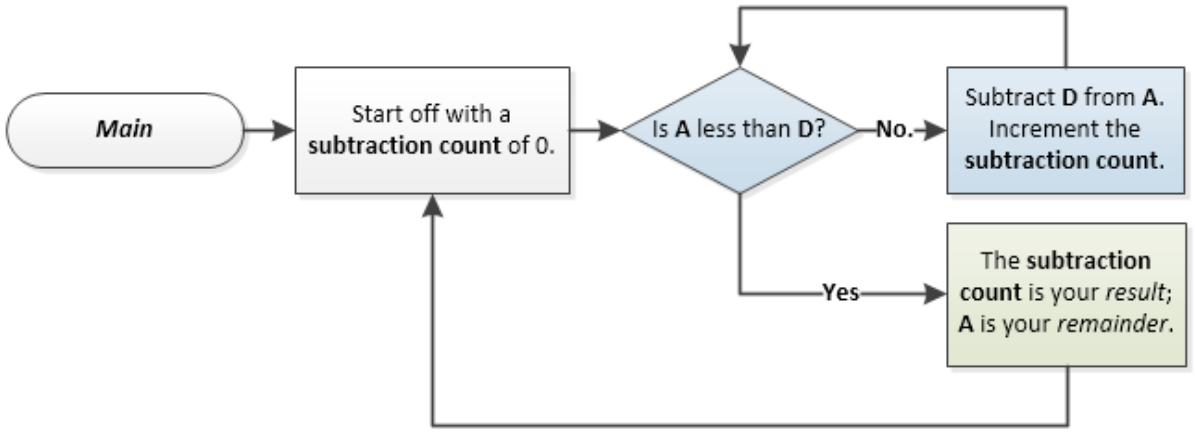
Read data a (R_data_a [7:0]):
Data from the register described by read address a.
Note: Depending on the instruction, the data in this register will get replaced as it also acts as the destination register.

Read data b (R_data_b [7:0]):
Data from register described by read address b.

Instruction Memory:

The instruction memory for this processor is a read only memory (ROM) that gets initialized with data and can only be read from. In our case this initialization will happen as we design the rest of the processor in xilinx.
You will **NOT** be designing the instruction rom for the processor **BUT** you will be writing a machine code program. The skeleton comes with a machine code program preloaded to give you an example.

That program is described by the flowchart below:



This program above performs division on two 8 bit numbers by using the repeated subtraction algorithm. You need to convert this flow chart into instructions for our processor and then into machine code and compare what you converted to what is in the instruction ROM to give you an understanding of what is going on.
Once you have studied how the instructions become machine code, you will need to convert the program described below into machine code and enter it into the Instruction rom. You will need to overwrite the current instructions in there to do this.

Program:

```
START:    MIN=127
LOOP:     If MIN==0, Goto END
          Load X (X comes from the input to the processor)
          P=|X| (absolute value of x)
          If P < MIN, MIN=P
          Goto LOOP
END:      Goto END
```

How to enter instructions into the Instruction ROM:

Once you have your instructions broken down into assembly, you will need to change them to machine code (binary) as discussed in class.
In order to enter the binary into the instruction rom in Xilinx you will need to follow these steps:

- Convert the binary to HEX.
- Go to the Instruction Rom schematic. You can get there by either:
    - Double Clicking on the Instruction PROM schematic in the Design Window.
    - Pushing into the GPP symbol then the Instruction ROM symbol.

        - You push into a symbol by right clicking on it -> Symbol -> Push into symbol.
    - **Note:** In the Instruction ROM, the inputs to the MUX represent the instruction to be executed. The Select signal for the mux is the instruction address aka the instruction to be executed next. This is fed from the PC(Program Counter).
    - The instructions are in order so the instruction tied to input 0 of the mux would be your first instruction and input 1 would be your second and so on.
- Once in the ROM double click on the constant box tied to the input of the instruction you need to edit.
- Edit the CValue in the pop window with the HEX value of the instruction you need.
- Save and synthesize the entire project. You do not need to synthesize just the Instruction ROM.

Testing

It is encouraged that you test your work as you go along. To aid in this, the skeleton comes with three separate test benches (simulation files) that you can use to test your designs.

ALU Test Bench

This is the same test bench from project 1 and is labelled: *ALU_ALU_sch_tb*
This testbench steps through all the opcode values in order starting from 000 up to 111.
The values for A and B are currently both set to the binary value “10101010”.

Register File Test Bench

The register file testbench is labelled: *RegFile_RegFile_sch_tb*
This test bench tests whether or not the registers load when the write enable is high and whether the correct data is read from the correct register.

The testbench oscillates the write enable signal, while stepping through all the registers addresses for both read addresses as well as the write address.
The data to be input (wr_data), changes by an increment of 8 on each iteration.

When ready to test, run the simulation, and compare the address inputs with data outputs. Although not set up by default in the waveform window, you can also add the registers themselves to the waveform.
To do this:

- Run the reg file simulation
- Once the waveform window opens, go to the window on the left hand side and click on the arrow right next to regfile_regfile_sch
- Click on UUT
- In the window between the one you are currently in and the waveform window, look for the names of your register symbols. Drag and drop these into the waveform

- Hit the reset button at the top of the waveform window (blue refresh arrow button) then run the simulation again using the blue play button **with an hourglass** next to it. **(Do not use the blue play button without the hour glass)**
- You should now be able to see the contents of the different registers as they change

Do not change anything in the source code for this simulation file

GPP Test Bench

This test bench tests the entire processor once the register file and ALU are done. It is named: *GPPTestCircuit_GPPTestCircuit_sch_tb*.
It is set up to test the default program that came with the skeleton, however it can be changed to test the program you are supposed to write.
Instructions on how to make that change are in the test bench itself and you can get to it by double clicking on the name of the test bench.

For the default program, the inputs are set to 10 and 2. Thus it is testing 10/2 and the result should be stored in R0 when the program finishes.

For the program you write, the sequence of inputs in the testbench is as follows:
MIN =127
X= 10
X=16
X=127
X=5
X=3
X=-2
X=16
X=1
X=16
X=4
X=0
X=4

You can change the inputs to what you want to test in the test bench by changing the binary value of Input in the sequence.

Implementation Details & Hints:

- **Review the Architecture Section.**
- **You must use schematics. You CANNOT use VHDL or Verilog in your design.**
- **You should design using components. Build and design the components one step at a time from the bottom up.**
- **You cannot use any of the inbuilt adders, subtractors or other xilinx arithmetic built in components. Flip flops (FD/*FDE*), D- Latches(LD), Multiplexors and decoders (d2_4E) are ok.**
    - **You CANNOT use the multi bit flip flops**
    - **You CANNOT use the RegParallelLoad, Mux 2_8, Mux 2_4, Mux16_4, Mux 16_9, Adder_4, Mux8_8, reg16 or reg2 components**
    - **Any multibit component needed has to be built by you**

- A skeleton has been provided. In addition to the empty ALU and Register File, the skeleton also contains simulation files for the ALU, Register File and the processor as a whole.
    - The ALU simulation file tests all possible operations on two inputs.
    - The Register file, tests loading values into all the registers and reading values from them.
    - The general simulation file, tests the first algorithm as described above. It also tests the second algorithm that you are required to build.
- **DO NOT** change the inputs or outputs in the skeleton.
- **DO NOT** change any of the schematics outside of the ones required for the project.
    - If you make any changes, redownload the skeleton and copy your work over as needed.
- You will need to zip up the entire folder your project is located in and submit it on blackboard labelled lastname_lastname_lastname_lastname_project2.zip
    - Each member of the team needs to have their lastname included on the submission folder.

    - Only one member of the team needs to submit
- Any late submissions will incur a penalty of 10 points per every 6 hours they are late.

Grading Breakdown:

The 100 total points for this part will be broken down into expected functionality, and how organized the design is.

- [75] Design functionality:
    - [10] ALU
    - [35] Register File
    - [30] New program in instruction rom

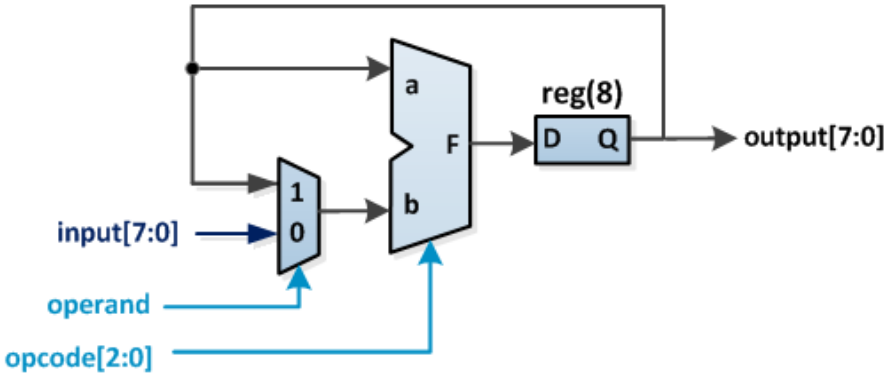
- [25] Style and Submission
    - [8] Use of components
    - [7] Schematic is organized and easy to follow
    - [10] Followed submission requirements

General Purpose Processor(GPP) Architecture:

This section describes the architecture for the GPP that will be used in this project. As outlined earlier, this is a very simple processor.

Arithmetic Logic Unit (ALU):

The ALU for this processor is an 8 bit ALU and can perform 8 operations as described in the table below.

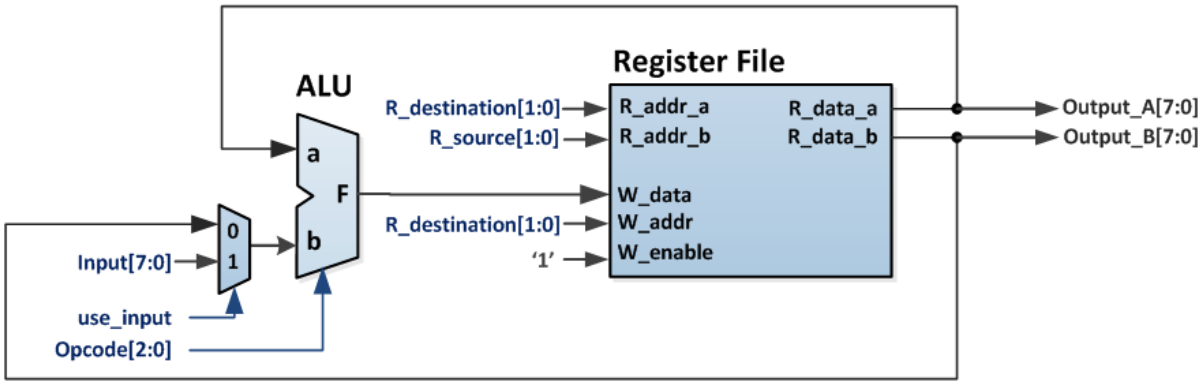


Opcode	Operation	Description
000	O = B	Output is = equal to input B
001	O = A ^ B (bitwise xor)	Xor of each bit in A with the corresponding bit in B
010	O = A AND B (bitwise and)	AND of each bit in A with the corresponding bit in B
011	O = A OR B (bitwise or)	OR of each bit in A with the corresponding bit in B
100	O = A + B (addition)	8 bit addition with a final carry out bit
101	O = A - B (subtract)	8 bit subtraction with a borrow out bit
110	O(7) = 0, 0(6:0) = B (7:1) (logical shift right)	Logical shift of the data at input B right by one bit
111	O = A * B (multiplication)	8 Multiplication. To make our design simpler, we will only keep the lower 8 bits of the

		result and signal an overflow

The B input for the ALU will take an address from a register or the input input. The input-input is the I/O input for this processor. It takes 8 bit data from the outside and feeds it into the ALU. This data will be used for operations or fed into a register and is chosen by the operand select signal. This is also the use-input bit in the machine code for the instruction.

Register File:



The register file is made of a bank of 4 8-bit registers named r0, r1, r2 and r3.
It has one write port and two read ports as well as a write enable signal that enables the register described by the write address.

Instruction ROM:

The instruction rom contains instructions for the processor. The only input it receives is the program counter (PC) which picks the value of the instruction to be executed next. The output is the 9 bit instruction.
The instruction rom for this processor takes upto 16 instructions.

Program Counter (PC) and Control ALU:

The PC is a 4 bit register that contains the value of the next instruction to be executed.
It is controlled by the control ALU, which either increments the PC or sets the PC to an instruction address based off a branch (jump) instruction.
The Control ALU takes in the C, Z and N flags, the instruction bits (3:0), instruction bits (7:4) and outputs the address of the instruction to be executed next.
C flag is the carry flag and is high when there is carry out or borrow out or multiplication overflow
N flag is the sign flag and is the most significant bit of the output
Z flag is the zero flag and is high when the result is 0

Instruction bits (3:0) are the last four bits of the instruction. These bits are the instruction address when executing a jump instruction

Instruction bits (7:4) is the opcode for the control ALU. These bits represent the current instruction being executed which helps decide whether the PC will be incremented by 1 or it will get the address (instruction 3:0) if it is a jump.

Instruction:

The instructions for this GPP are 9 bits long.

They are split into two categories, Data instructions and Control Instructions.
Data instructions work with the ALU and always save the data into a register file.
Control instructions do not save data to the register file and the bits reserved for the register file in the data instructions are used as the target address. These are the branch/jump instructions.
The instruction machine code is broken down as below:

	INSTRUCTION [8]	INSTRUCTION [7]	INSTRUCTION [6:4]	INSTRUCTION [3:2]	INSTRUCTION[1:0]
	USE_INPUT	w_enable_not			
Data Instruction	USE_INPUT	0	OPCODE	RD	RS
Control Instruction	X	1	OPCODE	Branch/Jump	Target Address

Machine Code		**Assembly Language** *use_input = 0,* *use_input = 1*	**Instruction Effect** *use_input = 0,* *use_input = 1*	**Flags Affected**
Instruction[7]	**Instruction[6:4]**			
write_enable_not	*opcode*			
0	`000`	`MOV Rd, Rs` `LD Rd, Input`	`Rd <= Rs` `Rd <= Input`	
0	`001`	`XOR Rd, Rs` `XOR Rd, Input`	`Rd <= Rd ^ Rs` `Rd <= Rd ^ Input`	`Z, N`

0	010	AND Rd, Rs AND Rd, Input	Rd <= Rd & Rs Rd <= Rd & Input	Z, N
0	011	OR Rd, Rs OR Rd, Input	Rd <= Rd | Rs Rd <= Rd | Input	Z, N
0	100	ADD Rd, Rs ADD Rd, Input	Rd <= Rd + Rs Rd <= Rd + Input	C, Z, N
0	101	SUB Rd, Rs SUB Rd, Input	Rd <= Rd - Rs Rd <= Rd - Input	C, Z, N
0	110	LSR Rd, Rs LSR Rd, Input	Rd <= Rs >> 1 Rd <= Input >> 1	C, Z, N

0	111	MUL Rd, Rs
		
		MUL Rd, Input
			Rd <= Rd * Rs
			
			Rd <= Rd * Input
				C, Z, N

1	000	JMP ADDR JMP ADDR	PC <= ADDR PC <= ADDR	
1	001	BCC ADDR ;Branch if C clear BCC ADDR	PC <= ADDR if C=0 PC <= ADDR if C=0	
1	010	BCS ADDR ;Branch if C set BCS ADDR	PC <= ADDR if C=1 PC <= ADDR if C=1	
1	011	BZC ADDR ;Branch if Z clear BZC ADDR	PC <= ADDR if Z=0 PC <= ADDR if Z=0	
1	100	BZS ADDR ;Branch if Z set BZS ADDR	PC <= ADDR if Z=1 PC <= ADDR if Z=1	

1	101	BNC ADDR ;Branch if N clear BNC ADDR	PC <= ADDR if N=0 PC <= ADDR if N=0	
1	110	BNS ADDR ;Branch if N set BNS ADDR	PC <= ADDR if N=1 PC <= ADDR if N=1	
1	111	*Unused / don't care.*	*Undefined / don't care.*	

Instruction example
Assembly:
ADD R0 R3
Machine Instruction:
0 0100 00 11

R0 = R0 + R3

Bits 1:0 are the read address b. In this example they contain the number 3, representing register R3.

Bits 3:2 are the read address a. In this example they contain the number 0, representing register R0. This is also the destination register where the result of the instruction is stored.

Bits 6:4 are the ALU opcode. This picks the operation in the ALU, in this case, addition (100). Compare this value with the opcode values from the ALU table.

Bit 7 is the write enable for the register file. This bit is inverted and thus a 0 is an enable and a 1 is not. Bit 7 is a 0 for all data operations.

Bit 8 is the use input bit. This decides whether the ALU will use a register or the input for the B input for ALU.

More examples:

Assembly	Machine Instruction	
ADD R1 Input	1 0100 01 XX	Add what is on the input to whatis in R1 and store it in R1. The last two bits in this case do not matter
MUL R2 R3	0 0111 10 11	Multiply r3 and r2 and save the result in r2
BCS 8	X 1010 10 00	Jump to instruction number 8 if the carry flag is a ‘1’. Bit 8 in this case does not matter. Note: that the last 4 bits which were used as registers in the data instructions are now used as the address we are jumping to.
BZC 10	X 1011 10 10	Jump to instruction at number 10 if the zero flag is ‘0’. Note: that the last 4 bits are used as the address we jump to