CSCE 230 – Group Labs (the Project)

1. Submission Deadlines
   - Group Lab 1 – 100 points – Do not need your DE1 board
     - Due: 11:59PM, **Nov 11 (Sunday)**, and no late submission will be accepted
   - Group Lab 2 – 100 points – Do not need your DE1 board
     - Due: 11:59PM, **Nov 25 (Sunday)**
     - Late submission is accepted until Dec 6 (Thursday) with penalty of 5 points per day
     - **This is the most difficult part in the project, please start as early as possible.**
   - Group Lab 3 – 100 points – Do not need your DE1 board
     - Due: 11:59PM, **Dec 2 (Sunday)**
     - Late submission is accepted until Dec 6 (Thursday) with penalty of 5 points per day
   - Optional Group Lab 4 – Bonus 45 points – Need to burn your processor on your DE1 board
     - Due: 11:59PM, **Dec 6 (Thursday)**, and no late submission will be accepted
2. Overview
   - You will design a processor, called a Simple Processor, which is very similar to a Nios Processor that we have studied in the class.
   - For a group with two students, one or both students should answer the questions and submit the files on Canvas.
3. Differences between a Nios Processor and a Simple Processor

	a Nios Processor	a Simple Processor
Word size	32 bits	16 bits
Addressing	Byte addressing that assigns successive addresses to successive bytes. Specifically, Address of the first word is 0, address of the second word is 4, address of the third word is 8, ...	Word addressing that assigns successive addresses to successive words. Specifically, Address of the first word is 0, address of the second word is 1, address of the third word is 2, ...
Number of registers in the register file	32 registers, r0, r1, ..., r31, each with 32 bits, r0 is always 0	8 registers, r0, r1, ..., r7, each with 16 bits, r0 is always 0
Return address register (ra)	r31	r7
Instruction size	A word of 32 bits	A word of 16 bits
Instruction formats	3 formats: R type, I type, and J type.	4 formats: R type, I type, B type, and J type. (explained below)
ALU flags	ALU flags are generated and then checked in stage 3 of the same instruction.	ALU flags are generated and saved to register STATUS, and then checked in stage 3 of the next (or later) instruction. (Refer to the cmp and branching instructions for more details)

4. Instruction set architecture (ISA)
   4.1. Four types of instruction formats
      4.1.1. R type, such as add, sub, and, or, xor, cmp, jmp, callr, ret
      - Instructions with only register operands.
      - 3-bit operation code (OP_code) is always 0b000, where 0b indicates a binary number.
      - 4-bit operation extension (OPX) indicates the specific type of an R-type instruction.
      - 3-bit D is the index of the destination register RD.
      - 3-bit T is the index of the second source register RT.
      - 3-bit S is the index of the first source register RS.

S	T	D	OPX	OP_code
15:13	12:10	9:7	6:3	2:0
				0b000

      4.1.2. I type, including ldw, stw, addi, ori, orhi
      - Instructions with a 7-bit immediate operand and at most two register operands.
      - 3-bit OP_code is between 0b001 and 0b101 and indicates the specific type of an I-type instruction.
      - 7-bit immediate operand, called imm7
      - 3-bit D is the index of register RD.
      - 3-bit S is the index of register RS.

S	D	7-bit immediate operand	OP_code
15:13	12:10	9:3	2:0

      4.1.3. B type, such as br, beq, bne, bge, ...
      - Branching instructions with a 9-bit immediate operand and no register operands.
      - 3-bit OP_code is always 0b110.
      - 4-bit OPX indicates the branching condition
      - 9-bit immediate operand, called imm9

9-bit immediate operand	OPX	OP_code
15:7	6:3	2:0
		0b110

      4.1.4. J type, including call.
      - Call instructions with a 13-bit immediate operand and no register operands.
      - 3-bit OP_code is always 0b111.
      - 13-bit immediate operand, called imm13

13-bit immediate operand	OP_code
15:3	2:0
	0b111

4.2. Arithmetic instructions

4.2.1. add

- Instruction type: R
- OP_code = 0b000, OPX = 0b0000
- Operation: RD ← RS + RT
- Example: add r2, r3, r4
- Format for the example

S	T	D	OPX	OP_code
15:13	12:10	9:7	6:3	2:0
3	4	2	0b0000	0b000

4.2.2. addi

- Instruction type: I
- OP_code = 0b011
- Operation: RD ← RS + σ(imm7). The last term is the 16-bit sign-extension of the 7-bit immediate operand.
- Example: addi r2, r3, 1
- Format for the example

S	D	7-bit immediate operand	OP_code
15:13	12:10	9:3	2:0
3	2	1	0b011

4.2.3. sub

- Instruction type: R
- OP_code = 0b000, OPX = 0b0001
- Operation: RD ← RS - RT
- Example: sub r2, r3, r4
- Format for the example

S	T	D	OPX	OP_code
15:13	12:10	9:7	6:3	2:0
3	4	2	0b0001	0b000

4.3. Logic instructions

4.3.1. and

- Instruction type: R
- OP_code = 0b000, OPX = 0b0010
- Operation: RD ← RS and RT
- Example: add r2, r3, r4
- Format for the example

S	T	D	OPX	OP_code
15:13	12:10	9:7	6:3	2:0
3	4	2	0b0010	0b000

4.3.2. or

- Instruction type: R
- OP_code = 0b000, OPX = 0b0011
- Operation: RD ← RS or RT
- Example: or r2, r3, r4
- Format for the example

S	T	D	OPX	OP_code
15:13	12:10	9:7	6:3	2:0
3	4	2	0b0011	0b000

4.3.3. ori

- Instruction type: I
- OP_code = 0b100
- Operation: RD ← RS or (0b000000000:imm7). The last term is the 16-bit zero-extension of the 7-bit immediate operand.
- Example: ori r2, r3, 1
- Format for the example

S	D	7-bit immediate operand	OP_code
15:13	12:10	9:3	2:0
3	2	1	0b100

4.3.4. orhi

- Instruction type: I
- OP_code = 0b101
- Operation: RD ← RS or (imm7:0b000000000). Note that, there are 9 zeros.
- Example: orhi r2, r3, 1
- Format for the example

S	D	7-bit immediate operand	OP_code
15:13	12:10	9:3	2:0
3	2	1	0b101

4.3.5. xor

- Instruction type: R
- OP_code = 0b000, OPX = 0b0100
- Operation: RD ← RS xor RT
- Example: xor r2, r3, r4
- Format for the example

S	T	D	OPX	OP_code
15:13	12:10	9:7	6:3	2:0
3	4	2	0b0100	0b000

4.3.6. nand

- Instruction type: R
- OP_code = 0b000, OPX = 0b0101
- Operation: RD ← RS nand RT = not (RS and RT)
- Example: nand r2, r3, r4
- Format for the example

S	T	D	OPX	OP_code
15:13	12:10	9:7	6:3	2:0
3	4	2	0b0101	0b000

4.3.7. nor

- Instruction type: R
- OP_code = 0b000, OPX = 0b0110
- Operation: RD ← RS nor RT = not (RS or RT)
- Example: nor r2, r3, r4
- Format for the example

S	T	D	OPX	OP_code
15:13	12:10	9:7	6:3	2:0
3	4	2	0b0110	0b000

4.3.8. xnor

- Instruction type: R
- OP_code = 0b000, OPX = 0b0111
- Operation: RD ← RS xnor RT = not (RS xor RT)
- Example: xnor r2, r3, r4
- Format for the example

S	T	D	OPX	OP_code
15:13	12:10	9:7	6:3	2:0
3	4	2	0b0111	0b000

4.4. Data copy instructions

4.4.1. ldw

- Instruction type: I
- OP_code = 0b001
- Operation: RD ← Mem16[RS + σ(imm7)].
- Example: ldw r2, 1(r3)
- Format for the example

S	D	7-bit immediate operand	OP_code
15:13	12:10	9:3	2:0
3	2	1	0b001

4.4.2. stw

- Instruction type: I
- OP_code = 0b010
- Operation: Mem16[RS + σ(imm7)] ← RD.
- Example: stw r2, 1(r3)
- Format for the example

S	D	7-bit immediate operand	OP_code
15:13	12:10	9:3	2:0
3	2	1	0b010

4.5. Control transfer instructions

4.5.1. cmp

- Instruction type: R
- OP_code = 0b000, OPX = 0b1000
- Operation: RS – RT, and then save ALU flags N, C, V, Z to register STATUS as bits 3, 2, 1, 0, respectively. Note that, cmp is the *only* instruction that writes ALU flags to register STATUS.
- Example: cmp r3, r4
- Format for the example

S	T	D	OPX	OP_code
15:13	12:10	9:7	6:3	2:0
3	4	0	0b1000	0b000

4.5.2. branching instructions

- Instruction type: B
- OP_code = 0b110
- Branching conditions

OPX	Instruction	Condition	sign	ALU Flags
0b0000	br	unconditional		
0b0001	beq	RS = RT		Z = 1
0b0010	bne	RS ≠ RT		Z = 0
0b0011	bgeu	RS >= RT	unsigned	C = 1
0b0100	bltu	RS < RT	unsigned	C = 0
0b0101	bgtu	RS > RT	unsigned	(C = 1) and (Z = 0)
0b0110	bleu	RS <= RT	unsigned	(C = 0) or (Z = 1)
0b0111	bge	RS >= RT	signed	N = V
0b1000	blt	RS < RT	signed	N ≠ V
0b1001	bgt	RS > RT	signed	(N = V) and (Z = 0)
0b1010	ble	RS <= RT	signed	(N ≠ V) or (Z = 1)
others	not used	not used		

- Operation: PC ← PC + **1** + σ(imm9) if condition is true; PC ← PC + **1**, otherwise.

- A branching instruction is used together with a cmp instruction.
  - The cmp instruction compares two registers RS and RT and saves the ALU flags to register STATUS.
  - The branching instruction checks its condition using the ALU flags saved in register STATUS.
    - If the condition is true, the processor executes the instruction at the LABEL;
    - otherwise the processor executes the next instruction.
- Note that,
  - A branching instruction in a Nios Processor is equivalent to a cmp instruction and a branching instruction in a Simple Processor.
  - A Simple Processor uses word addressing, and thus the address difference between two consecutive instructions is only 1. This is why PC + 1.
- Example:
  - Address=0x0F cmp r3, r4
  - Address=0x10 beq LABEL
  - ...
  - Address=0x15 LABEL:
- Format for the beq instruction example

9-bit immediate operand	OPX	OP_code
15:7	6:3	2:0
0x15-(0x10+1) = 4 = 0b000000100	0b0001	0b110

4.5.3. jmp

- Instruction type: R
- OP_code = 0b000, OPX = 0b1001
- Operation: PC ← r2.
- Example: jmp r2
- Format for the example

S	T	D	OPX	OP_code
15:13	12:10	9:7	6:3	2:0
2	0	0	0b1001	0b000

4.5.4. callr

- Instruction type: R
- OP_code = 0b000, OPX = 0b1010
- Operation: r7 ← PC + 1, PC ← r2
- Example: callr r2
- Format for the example

S	T	D	OPX	OP_code
15:13	12:10	9:7	6:3	2:0
2	0	7	0b1010	0b000

4.5.5. call

- Instruction type: J
- OP_code = 0b111
- Operation: r7 ← PC + 1, PC ← ($PC_{15:13}$:imm13)
- Example:
  - Address=0x000 call LABEL
  - ...
  - Address=0x100 LABEL:
- Format for the example

13-bit immediate operand	OP_code
15:3	2:0
(Address of LABEL)$_{12:0}$=0x100	0b111

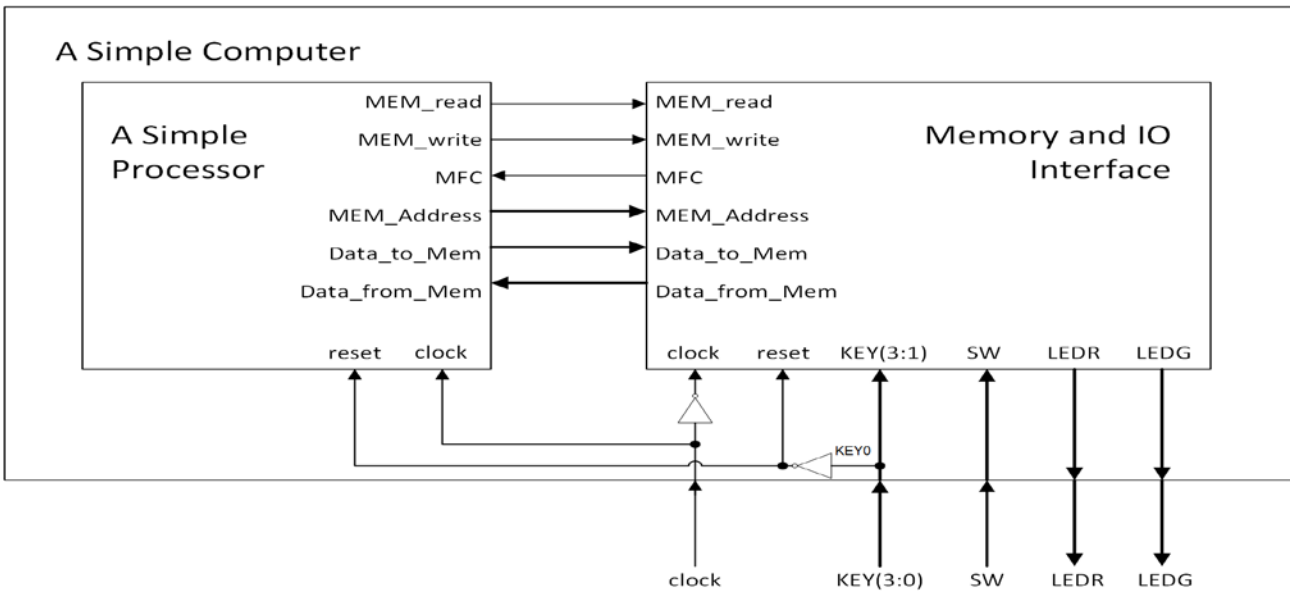
4.5.6. ret

- Instruction type: R
- OP_code = 0b000, OPX = 0b1011
- Operation: PC ← r7
- Example: ret
- Format for the example

S	T	D	OPX	OP_code
15:13	12:10	9:7	6:3	2:0
7	0	0	0b1011	0b000

5. Computer organization

5.1. A simple computer consists of a simple processor and a memory and IO interface.



5.2. VHDL designs

- A Simple Computer: SimpleComputer.vhd, which is complete and is the **top-level entity** for Quartus.
- A Simple Processor: SimpleProcessor.vhd. which is **incomplete** and is what you need to complete in this project.
- Memory and I/O:
  - MemoryIOInterface.vhd, which is complete.
  - MainMemory.vhd, which is complete.
  - MemoryInitialization.mif, which is **incomplete** and is where you need to write the binary/hexadecimal encodings of an assembly program. This mif file will be automatically read by MainMemory.vhd to initialize the initial values of memory.

5.3. Computer Interface

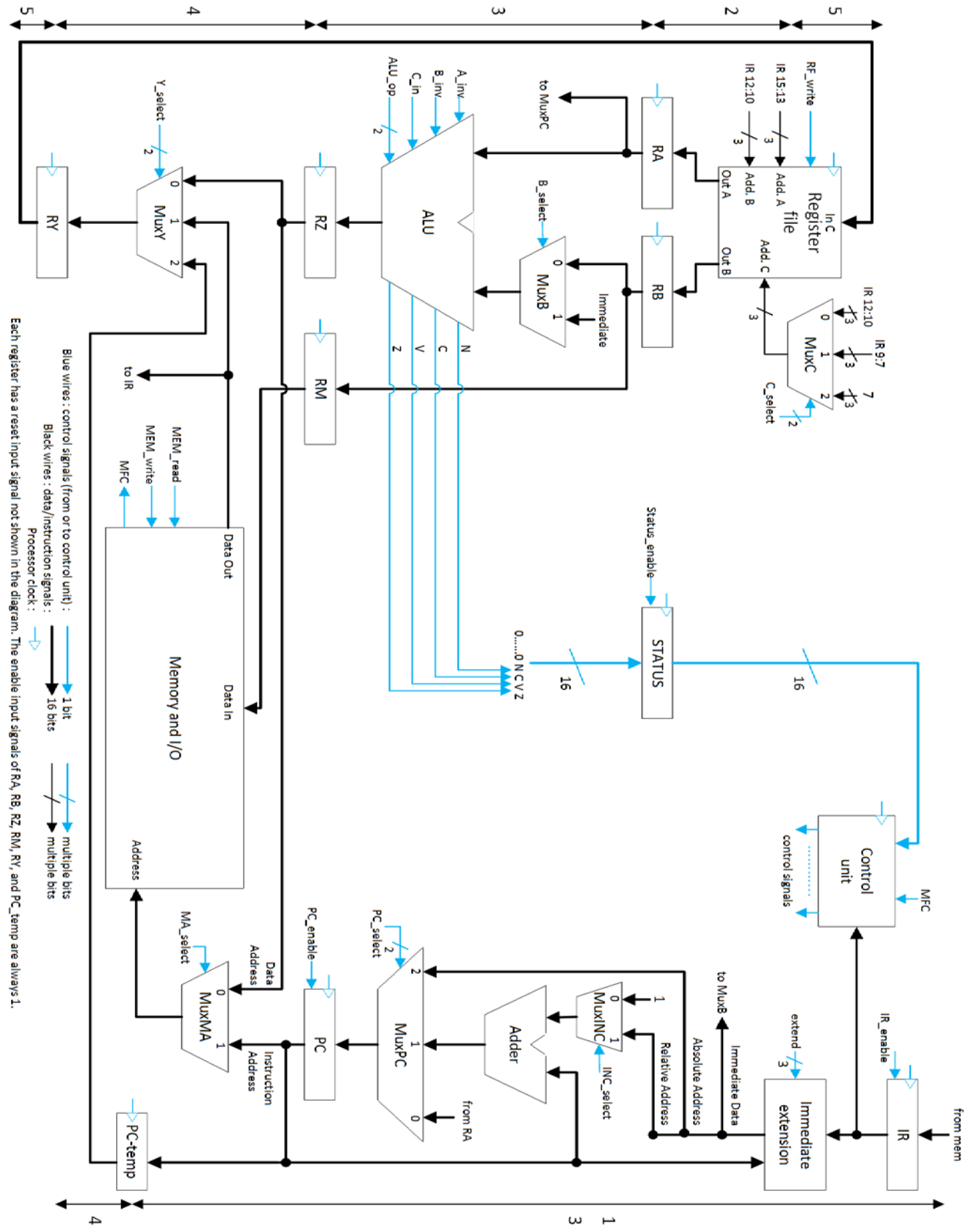
- The processor clock input port is directly connected to the computer clock input port, and is triggered by a rising edge of the computer clock. The memory and IO clock input port is connected to the output of a NOT gate, and thus is triggered by a falling edge of the computer clock.
- When KEY0 is pushed (i.e. KEY0=0), the computer is reset. Therefore, the inverted KEY0 is connected to the reset input ports of the processor and memory and IO interface.

5.4. Main memory and I/O ports

Base Address	End Address	Memory or I/O	16-bit data at an address
0x0000	0x0FFF	Main memory	
0x1000	0x1000	LEDR: LED Red (Write only)	000000:LEDR9:LEDR8:...:LEDR0
0x1010	0x1010	LEDG: LED Green (Write only)	00000000:LEDG7:LEDG6:...:LEDG0
0x1040	0x1040	SW: Slider Switch (Read only)	000000:SW9:SW8:...:SW0
0x1050	0x1050	KEY: Push Button (Read only)	000000000000:KEY3:KEY2:KEY1:0

6. Processor architecture

6.1. The processor architecture diagram



6.2. Diagram differences between a Nios Processor and a Simple Processor

- Addresses A and B of Register File
- All inputs of MuxC
- STATUS register
- extend input signal of Immediate Extension
- Input0 of MuxINC
- Input and output of Control Unit

6.3. VHDL designs

6.3.1. The simple processor

- SimpleProcessor.vhd, which is **incomplete** but already contains all necessary components and internal signals. You only need to write the implementation between "begin" and "end" of the architecture. You may add other internal signals or output signals into SimpleProcessor.vhd for debugging, but do not create any other new VHDL files in this project.

6.3.2. Register file

- RegisterFile8by16Bit.vhd, which is complete

6.3.3. Other registers

- Please use Reg16Bit.vhd as a component to implement all other registers including PC, IR, STATUS, RA, RB, RZ, RM, RY, PC_temp.

6.3.4. ALU

- ALU.vhd, which is complete

6.3.5. Immediate extension

- Immediate.vhd, which is complete**. This design is very different from the immediate design that we have studied in the class, so please read this file carefully in order to determine which extension to use for various instructions**.

6.3.6. PC adder

- Adder16Bit.vhd, which is complete and is used both as a PC Adder and in the ALU.

6.3.7. Control unit

- ControlUnit.vhd, which is **incomplete** but already contains the current state logic and the next state logic. You only need to write the output logic. You may add other internal signals or output signals into ControlUnit.vhd for debugging, but again do not create any other new VHDL design files in this project.

6.3.8. Various multiplexers

- Mux4Input3Bit.vhd, Mux2Input16Bit.vhd, Mux4Input16Bit.vhd, and Mux8Input16Bit.vhd, which are all complete and are all the multiplexers that we need in this project.

7. Group Lab 1
   7.1. Deadline
      - Please check the first page for the deadline.
   7.2. Lab objective
      - Familiar with the instruction formats and encoding
   7.3. Lab task
      7.3.1. Please convert the following three Simple Processor programs to hexadecimal encodings
      7.3.2. Assembly program 1 (used in group lab 2 later)

```
      add   r0, r0, r0        # at address 0
      addi  r1, r0, 1         # at address 1
      ori   r2, r0, 2         # at address 2
      xor   r3, r1, r2        # at address 3
      add   r4, r1, r3        # at address 4
      or    r5, r1, r4        # at address 5
      nor   r6, r2, r4        # at address 6
      sub   r7, r6, r2        # at address 7
      orhi  r7, r3, 3         # at address 8
```

      7.3.3. Assembly program 2 (used in group lab 3 later)

```
      add   r0, r0, r0        #0:
      addi  r6, r0, 0x20      #1:
      addi  r2, r0, 2         #2:
      call  sum               #3: r7 used as ra
end:  br end                  #4:
sum:  addi  r6, r6, -4        #5: r6 is used as sp
      stw   r4, (r6)          #6: push r4
      addi  r3, r0, 0         #7:
      addi  r4, r0, 1         #8:
loop: add   r3, r3, r4        #9:
      addi  r4, r4, 1         #A:
      cmp   r2, r4            #B:
      bge   loop              #C:
      ldw   r4, (r6)          #D: pop r4
      addi  r6, r6, 4         #E:
      ret                     #F: return to ra
```

      7.3.4. Assembly program 3 (used in optional group lab 4 later)

```
      add   r0, r0, r0        #0:
      orhi  r1, r0, 8         #1: LEDR Address
      ori   r2, r1, 0x10      #2: LEDG Address
      ori   r3, r1, 0x40      #3: SW Address
      ori   r4, r1, 0x50      #4: KEY Address
      addi  r5, r0, 1         #5:
loop: ldw   r6, (r3)          #6: Get current SW status
      stw   r6, (r1)          #7: Set LEDR according to SW
      stw   r5, (r2)          #8: Set LEDG according to r5
      ldw   r6, (r4)          #9: get current KEY status
      cmp   r6, r7            #A: cmp with previous KEY status
      beq   loop              #B: No change in KEY, then goto loop
      add   r7, r6, r0        #C: save current KEY status to r7
      addi  r5, r5, 1         #D: increase r5 by 1
      br    loop              #E: goto loop
```

8. Group Lab 2 (**this is the most difficult part in the whole project**)

8.1. Deadline

- Please check the first page for the deadline.

8.2. Lab objective

- Implement the datapath and all arithmetic and logic instructions

8.3. Lab task

8.3.1. Project files

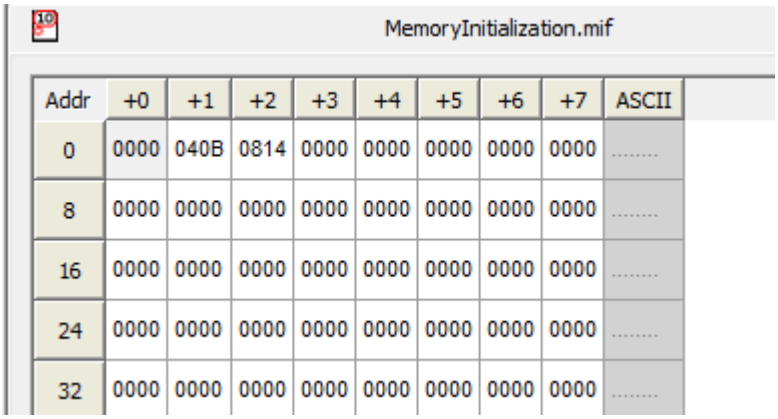
- Please create a Quartus project with all the provided files, including all the vhd files, MemoryInitialization.mif file, and task.do file.
- *If you are using the lab computers, please create a new project directory in C:\temp instead of your Z: drive (after the lab, please backup your files to your Z: drive though); otherwise, Quartus does not work correctly (e.g, does not read mif files).*
- Please select VHDL 2008 as follow. Otherwise, you may see error messages like "Use of non globally static actual... requires VHDL 2008" when running ModelSim
  - Choose menu "Assignment"
  - Select "Setting" to open the Setting window
  - Click "VHDL input" on the left side, and then select "VHDL 2008".
- Please carefully read the provided files. We have already studied all of them, except the MainMemory and IO interface.

8.3.2. Files to modify

- SimpleProcessor.vhd: please connect all the components according to the processor architecture diagram.
- ControlUnit.vhd: Please set the output control signals for all the arithmetic and logic instructions. Note: simply copying and pasting the VHDL code from the lecture slides will not work due to the difference between a Nios processor and a Simple processor.
- MemoryInitialization.mif: Please write the hexadecimal encodings of **program 1** in group lab 1 to MemoryInitialization.mif. Note: this mif file will be automatically read by MainMemory.vhd to initialize the initial values of main memory, so you **do not need to do anything to load it**.
- Do not create any other new VHDL design files, or modify any other VHDL files except for debugging purpose.

8.3.3. MemoryInitialization.mif

- The following is a screenshot of MemoryInitialization.mif. You can double-click a cell to enter the hexadecimal encoding of an instruction at that address.
- For example, address 0: 0x0000 is the hexadecimal encoding of "add r0, r0, r0".
- For example, address 1: 0x040B is the hexadecimal encoding of "addi r1, r0, 1"
- For example, address 2: 0x0814 is the hexadecimal encoding of "ori r2, r0, 2"



MemoryInitialization.mif

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	0000	040B	0814	0000	0000	0000	0000	0000	.....
8	0000	0000	0000	0000	0000	0000	0000	0000	.....
16	0000	0000	0000	0000	0000	0000	0000	0000	.....
24	0000	0000	0000	0000	0000	0000	0000	0000	.....
32	0000	0000	0000	0000	0000	0000	0000	0000	.....

8.4. Self test

- Please run ModelSim with the provided testing script task.do, and verify that the assembly program executes correctly as follows
  - r1 = 0x0001 after running the instruction at address 1
  - r2 = 0x0002 after running the instruction at address 2
  - r3 = 0x0003 after running the instruction at address 3
  - r4 = 0x0004 after running the instruction at address 4
  - r5 = 0x0005 after running the instruction at address 5
  - r6 = 0xFFF9 after running the instruction at address 6
  - r7 = 0xFFF7 after running the instruction at address 7
  - r7 = 0x0603 after running the instruction at address 8

8.5. Hints for debugging

- If the assembly program does not execute correctly, please
  - first, check whether the binary/hexadecimal encoding of each instruction is correct. The correct encodings will be available on Canvas after the due date of group lab 1.
  - second, check whether PC is updated correctly after each instruction. PC should start from zero, and then increase by one after stage 1 of each arithmetic or logic instruction. If not, please check the components responsible for updating PC. You may need to add more debugging signals.
  - next, check whether the correct instruction is loaded to IR right after stage 1. For example, right after stage 1 of the first instruction, IR=0x0000. Right after stage 1 of the second instruction, IR=0x040B. If not, please check the values of input ports and output ports of MemoryIOInterface. You need to add more debugging signals to check their values.
  - Finally, check whether the inter-stage registers and control signals for each stage have the correct values. You need to add more debugging signals to check their values.

9. Group Lab 3
   9.1. Deadline
      - Please check the first page for the deadline.
   9.2. Lab objective
      - Implement all data copy and control transfer instructions
   9.3. Lab task
      9.3.1. Project files
         - Re-use your Quartus project files from last group lab.
      9.3.2. Files to modify
         - ControlUnit.vhd: Please set the output control signals for all data copy and control transfer instructions.
         - SimpleProcessor.vhd: Modify only if necessary. For example, if you made mistakes in last group lab.
         - MemoryInitialization.mif: Please write the hexadecimal encodings of **program 2** in group lab 1 to MemoryInitialization.mif.
   9.4. Self test
      - Please run ModelSim with the provided testing script task.do, and verify that the assembly program executes correctly as follows
         - r6 = 0x0020 after running the instruction at address 1
         - r2 = 0x0002 after running the instruction at address 2
         - PC = 0x0005 after running the instruction at address 3
         - r6 = 0x001C after running the instruction at address 5
         - r2 = 0x0002, r3 = 0x0001, r4 = 0x0002 after running the instruction at address 0xC for the first time
         - r2 = 0x0002, r3 = 0x0003, r4 = 0x0003 after running the instruction at address 0xC for the second time
         - r4 = 0x0000 after running the instruction at address 0xD
         - r6 = 0x0020 after running the instruction at address 0xE
         - PC = 0x0004 after running the instruction at address 0xF
         - The program finally enters an infinite loop at address 4

10. Optional Group Lab 4

10.1. Deadline

- Please check the first page for the deadline.

10.2. Lab objective

- Load and test our Simple Computer design on your DE1 board

10.3. Lab task

10.3.1. Project files

- Re-use your Quartus project files from last group lab.
- Automatic pin assignment: menu "Assignment" → "Import Assignment" → select file "Project_pin_assignments.csv"→ click the "OK" button

10.3.2. Files to modify

- ControlUnit.vhd: Modify only if necessary.
- SimpleProcessor.vhd: Modify only if necessary.
- MemoryInitialization.mif: Please write the hexadecimal encodings of **program 3** in group lab 1 to the MemoryInitialization.mif.

10.4. Self test

- After you have successfully loaded the design onto your DE1 board, please *first* press KEY0 to reset the Simple Computer.
- This assembly program sets red LEDs (LEDR) according to the positions of Slider Switches (SW), and sets green LEDs (LEDG) according to a binary number which is increased by any change (press or release) of KEY3, KEY2, and KEY1.

10.5. **Please return your DE1 board to the TAs in the labs**.