Middle East Technical University | Department of Computer Engineering

CENG 232

Logic Design

Spring '2016-2017

Lab 5

(Individual Work)

Due date: 28 May 2017, 23:59
No late submissions!

1 Introduction

In this laboratory, you will finish the implementation of our basic computer architecture. First, you will implementing the **control subsystem**. Then, you will generate **processor module** which puts together the implementations of **data and control subsystems**. Then, as the final part of the homework, you will implement **computer** module. This module basically combines **memory and processor modules**. When the implementation of the **computer module** finished, you will have a ready to run computer which can fetch the instructions from the memory, decode them and calculate the result. In this lab, Fpga part is excluded, which means that you won't be testing your implementations with physical boards. Testing with test benches will be enough.

2 Lab Work

You have already finished data subsystem and memory subsystem. You will add one more key subsystem implementation to your design which is **control subsystem**. Then, in the other parts you will just combine the existing modules to provide a hierarchical computer design. In the following sections; first, general structure of the data (instruction formats, instruction sets etc.) is explained and then, general structures of the memory is provided. Next, control subsystem and the parts you are required to implement are presented. After that, connections that you should make is explained in processor and computer subsections.

2.1 Instruction Formats

An instruction is represented by one word (32 bits). These 32 bits have different meaning according to the formats depicted in Figure 1. In these formats, instructions are divided into the following fields:

- An opcode field (Opcode) which is six bits wide and specifies the operation performed by the instruction (up to 64 operations);
- Two register fields (RT/RS and RA) which are five bits wide. Each specifies one of the 32 general-purpose registers. Registers RA and RS contain operands and register RT specifies the destination register; and a 16-bit field that may have different interpretations (depending on the opcode):

– a 5-bit register field (RB) which specifies an additional register used by the instruction (the rest of the bits are unused); or

– a 16-bit value (SI or UI), which specifies the value of an operand used by the instruction. This value can be interpreted as a signed integer, an unsigned integer, or a bit vector, depending on the opcode; or

– a 16-bit signed displacement (D) used to calculate an address in a memory instruction or in a branch;

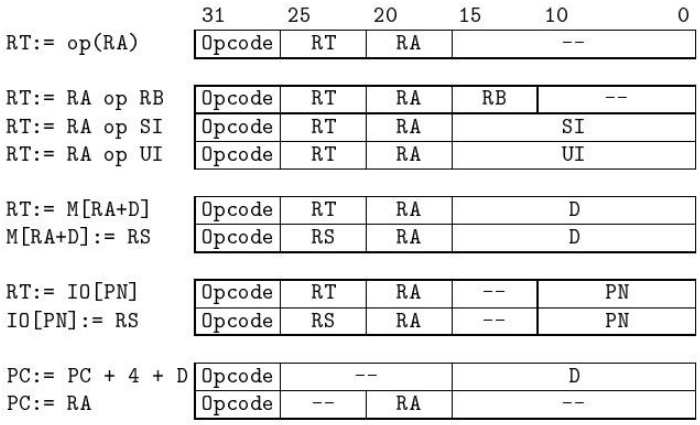

Figure 1: Instruction formats.

Note that symbols RT, RS, RA, RB are used to denote the corresponding instruction fields as well as the contents of the registers whose address is specified in those instruction fields; the actual meaning of the symbol is inferred from the context. On the other hand, SI, UI, D, PN denote the contents of the corresponding fields within the instruction, whereas symbols PC, IR, CR denote the contents of those registers.

2.2 Instruction Set

The processor has six groups of instructions, which are listed in Figures 2 3 and 4. For each instruction, the table lists the name, the opcode, the function, whether it sets the Condition Register (CR), and the representation used in an assembly-language program, as described later. The corresponding instruction groups are:

- Unary operations, which have a single operand and produce the corresponding result, including setting the condition register CR.
- Binary operations, which have two operands and produce the corresponding result, also setting CR.
- Memory operations, which access the memory location whose address (referred to as the effective address EA) corresponds to the lowemost 24 bits from the result of adding the sign-extended value specified in field D to the contets of the register specified in field RA. The length of the operand transferred can be one byte or one word (four bytes) (denoted as Mem(RA+D,1) and Mem(RA+D,4), respectively, in the table).
- I/O operations: In this assignment we will not deal with I/O operations.

The instruction set listed in Figure 2, 3 and 4 provides a complete description of the capabilities of the processor; any computation sequence executed by this system must be represented as a program written using these instructions.

In this lab you will implement only 4 of the instructions in control subsystem; adi, xor, ldw, stb.

Name	Opcode	Function	CR	Assembly Language
No-op	000000	no operation		nop
NOT	000010	RT:= not(RA)	Y	not RT,RA
Left shift	000100	RT:= lshift(RA)	Y	lsh RT,RA
Right shift	000110	RT:= rshift(RA)	Y	rsh RT,RA
Left rotate	001000	RT:= lrot(RA)	Y	lrt RT,RA
Right rot.	001010	RT:= rrot(RA)	Y	rrt RT,RA
Add	010000	RT:= RA + RB	Y	add RT,RA,RB
Add immed.	010001	RT:= RA + SI	Y	adi RT,RA,SI
Subtract	010010	RT:= RA - RB	Y	sub RT,RA,RB
Sub. immed.	010011	RT:= RA - SI	Y	sbi RT,RA,SI
AND	010100	RT:= RA and RB	Y	and RT,RA,RB
AND immed.	010101	RT:= RA and UI	Y	ani RT,RA,UI
OR	010110	RT:= RA or RB	Y	or RT,RA,RB
OR immed.	010111	RT:= RA or UI	Y	ori RT,RA,UI
XOR	011000	RT:= RA xor RB	Y	xor RT,RA,RB
XOR immed.	011001	RT:= RA xor UI	Y	xri RT,RA,UI

Figure 2: Instruction set.

Name	Opcode	Function	CR	Assembly Language
Load byte	100000	RT(7 to 0):= Mem(RA+D,1)		ldb RT,D(RA)
Load word	100001	RT(31 to 0):= Mem(RA+D,4)		ldw RT,D(RA)
Store byte	100010	Mem(RA+D,1):= RS(7 to 0)		stb RS,D(RA)
Store word	100011	Mem(RA+D,4):= RS(31 to 0)		stw RS,D(RA)
I/O Rd byte	100100	RT(7 to 0):= IO(PN,1)		irb RT,PN
I/O Rd word	100101	RT(31 to 0):= IO(PN,4)		irw RT,PN
I/O Wr byte	100110	IO(PN,1):= RS(7 to 0)		iwb RS,PN
I/O Wr word	100111	IO(PN,4):= RS(31 to 0)		iww RS,PN

Figure 3: Instruction set.

2.3 Data Representation

As it can be observed from the instruction types, a 32-bit data item can represent any of the following data types, the interpretation being given by the type of instruction that uses it:

- Logical data (bit-vectors) used for logical operations, such as AND and XOR.
- Integers (signed) in twos complement notation; this is used for all arithmetic operations.
- Unsigned integers used for representation of addresses.

In addition, memory operations use 8-bit (byte) data as unsigned or logical values.

Name	Opcode	Function	CR	Assembly Language
Branch	111000	PC:= PC + 4 + D		br D
Branch indirect	111001	PC:= RA		bri RA
Branch if N=0	110000	If N=0 then PC:= PC+4+D		brp D
Branch if N=1	110001	If N=1 then PC:= PC+4+D		brn D
Branch if Z=0	110010	If Z=0 then PC:= PC+4+D		bnz D
Branch if Z=1	110011	If Z=1 then PC:= PC+4+D		brz D
Branch if C=0	110100	If C=0 then PC:= PC+4+D		bnc D
Branch if C=1	110101	If C=1 then PC:= PC+4+D		brc D
Branch if V=0	110110	If V=0 then PC:= PC+4+D		bnv D
Branch if V=1	110111	If V=1 then PC:= PC+4+D		brv D

Figure 4: Instruction set.

2.4 Specification of the Memory Contents

For simulation purposes, we fill the initial contents of the memory with a program; moreover we defined a memory smaller than the maximum size defined in Lab-4. Please note that, we may change the memory content while evaluating your modules.

```
reg [7:0] mem[55:0];
initial
begin
//program
mem[3]  = 8'b01100000; mem[2]  = 8'b00000000; mem[1]  = 8'b00000000; mem[0]  = 8'b00000000;
mem[7]  = 8'b01000100; mem[6]  = 8'b00100000; mem[5]  = 8'b00000000; mem[4]  = 8'b00110010;
mem[11] = 8'b10000110; mem[10] = 8'b10000001; mem[9]  = 8'b00000000; mem[8]  = 8'b00000000;
mem[15] = 8'b10000110; mem[14] = 8'b10100001; mem[13] = 8'b00000000; mem[12] = 8'b00000100;
mem[19] = 8'b01000100; mem[18] = 8'b01000000; mem[17] = 8'b00000000; mem[16] = 8'b00111111;
mem[23] = 8'b10001000; mem[22] = 8'b01000001; mem[21] = 8'b00000000; mem[20] = 8'b00000000;
//data
mem[51] = 8'b00110011; mem[50] = 8'b00001111; mem[49] = 8'b11110000; mem[48] = 8'b11001100;
mem[55] = 8'b00110011; mem[54] = 8'b00001111; mem[53] = 8'b11110000; mem[52] = 8'b11001100;
end
```

The memory contents given above corresponds to the following instructions:

```
0x000000: xor R0,R0,R0   ; R0 = 0
0x000004: adi R1,R0,50   ; R1 = 50
0x000008: ldw R20,0(R1)  ; R20= Mem(50,4)= Mem(48,4)
0x00000C: ldw R21,4(R1)  ; R21= Mem(54,4)= Mem(52,4)
0x000010: adi R2,R0,63   ; R2 = 63
0x000014: stb R2,0(R1)   ; Mem(50,1) = 63

0x000048: 0x330FF0CC
0x000052: 0x330FF0CC
```

Figure 5: Instructions generated from the above memory contents.

2.5 Control Subsystem

The control subsystem produces the control signals so that the instruction loop is performed in the data subsystem. The control subsystem is a canonical sequential network, as depicted in Figure 6.

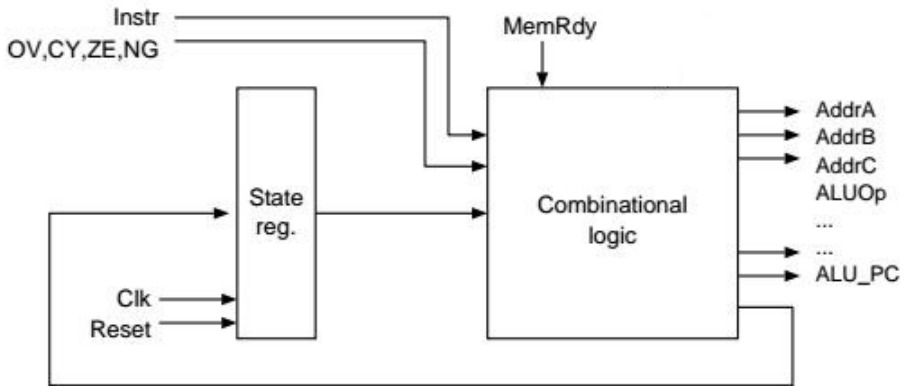

Figure 6: Control subsystem.

The first question to be answered in the design of the control subsystem is the number of clock cycles for the instruction loop. Activities are separated into different cycles (control states) for the following reasons:

1. A resource is used more than once in the instruction. This is the case of the ALU, which is used to increment PC and also to perform the operation. It is also the case of the memory in Load and Store operations as it is used twice, as for instruction fetch and once for data read/write.

2. Because different instructions take different times, dividing the execution in several cycles reduces the execution time of the faster instructions. For example, the main delays involved in the execution part of an Add instruction are delay of reading the operands from RF, the delay of performing the operation in ALU, and the delay of storing the result in RF. On the other hand, the memory instruction Load has these same delays plus the delay of reading the operand from memory. Consequently, it is convenient to divide the execution of the Load instruction into two parts, so that the execution of the Add instruction is done in one cycle whereas the Load is done in two cycles.

The exact number of cycles used for an implementation and the activities that are included in each cycle are complex design decisions. The objectives are using the resources effectively and achieving the best execution time (for the instruction mix of typical programs). In our illustration implementation we use three states, as shown by the state diagram depicted in Figure 7. The activities in each of the states are:

Fetch (Status=001): The instruction is fetched from memory and stored in IR. The address of the next sequential instruction is calculated in the ALU and stored in PC.

Execute (Status=010): The instruction is decoded, the operands are fetched from registers, the operation is performed in the ALU and the result is stored in the destination register. In memory instructions (Load and Store) as well as in branches, the effective address is calculated.

Memop (Status=011): In memory instructions, a memory operation (read, write) is performed. Moreover, in Load the data is stored in RF.

As a consequence of this division, two clock cycles are required for instructions other than memory instructions, and three clock cycles for memory instructions.

The combinatorial logic receives as inputs the value in the state register as well as the condition signals from the data section (signals Instr, Z, N, C, V, MemRdy). Based on these inputs, this subsystem produces the signals that control the operation of the entire system, including the generation of the input to the state register (value of the next state). The values of these signals are shown in the state diagram depicted in Figure 7. Most of these signals depend only on the value of the state register, so they are available a short delay after the state register changes value; let us call this **Ctrl_delay**. On the other hand, decoding the instruction requires a few levels of logic, so that the corresponding delay is longer; we refer to this as **Dec_delay**.

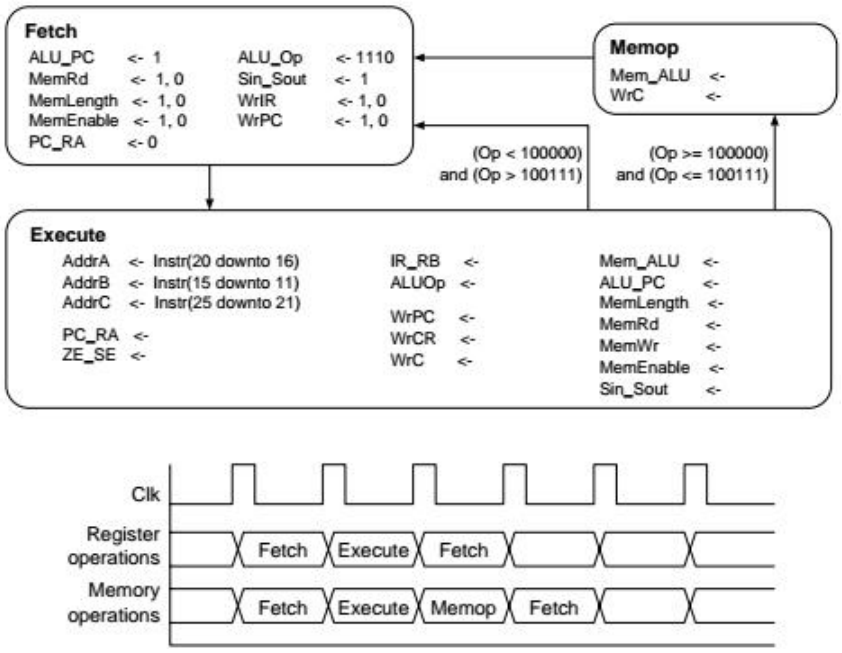

Figure 7: State diagram and timing for control subsystem of the processor.

In fetch/memop states, you should consider the delays given below: To read from memory, you should set Rd=1 after **MemRd_delay**(2.5ns) and Rd=0 after **MemRd_pulse**(5.5ns) To write to memory, you should set Wr=1 after **MemWr_delay**(2.5ns) and Rd=0 after **MemWr_pulse**(5.5ns)

The signature of the module is given below:

```
module Ctrl_Subsystem(
input [31:0] Instr,
input ZE, NG, CY, OV ,
output reg [4:0] AddrA, AddrB, AddrC,
output reg [3:0] ALUOp,
output reg WrC, WrPC, WrCR, WrIR,
output reg Mem_ALU, PC_RA, IR_RB,
output reg ALU_PC, ZE_SE, Sin_Sout,
output reg MemRd,MemWr,
output reg MemLength,
output reg MemEnable,
input MemRdy,
output reg [2:0] Status, // 0:p_reset, 1:fetch, 2:execute, 3:memop
input Clk, Reset
    );
```

PS: In this lab you will implement only 4 of the instructions in control subsystem; xor, adi, ldw, stb.

1. **xor: RT = RA xor RB**
   (a) The RT value denotes the DataC value in Register File whose address is AddrC.
   (b) The RA value denotes the DataA value in Register File whose address is AddrA.
   (c) The RB value denotes the DataB value in Register File whose address is AddrB.

2. **adi: RT = RA + SI**

   (a) The RT value denotes the DataC value in Register File whose address is AddrC.

   (b) The RA value denotes the DataA value in Register File whose address is AddrA.

   (c) SI is a 32-bit value, which is the sign-extended version of right-most 16 bits of the instruction.

3. **ldw: RT[31:0] = Mem[RA+D,word]**

   (a) The RT value denotes the DataC value in Register File whose address is AddrC.

   (b) The RA value denotes the DataA value in Register File whose address is AddrA.

   (c) D is a 32-bit value, which is the zero-extended version of right-most 16 bits of the instruction.

4. **stb: Mem[RA+D,byte] = RS[7:0]**

   (a) The RA value denotes the DataA value in Register File whose address is AddrA.

   (b) D is a 32-bit value, which is the zero-extended version of right-most 16 bits of the instruction.

   (c) The RS value denotes the DataB value in Register File whose address is AddrB. (RS[7:0] is the right-most 8 bits of RS)

2.6 Operation of the Computer And Cycle Time

1. **Fetch** (see Figure 8):

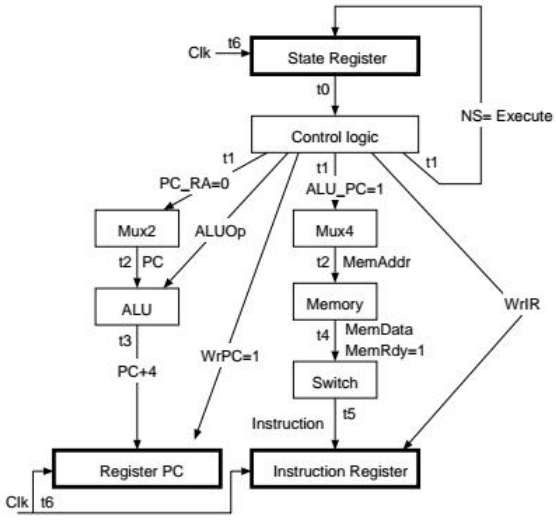

Figure 8: Dependencies for state fetch.

   (a) The iteration begins at time $t_0 = t_{clk} + Ctrl\_delay(0.5)$, when the output of the state register has been set to the value Fetch, that is, state=001.

   (b) At $t_1 = t_0 + Ctrl\_delay$, the control subsystem activates the control signals to fetch the next instruction (ALU_PC=1, MemLength=1, MemEnable=1, Sin_Sout=0, WrIR=1), and to increment PC (PC_RA=0, ALUop=1110, WrPC=1). In addition, the control logic produces the input to State Register (next state=Execute =010). Signal MemRd=1 becomes stable at $t_0 + memRd\_delay(2.5)$.

   (c) At $t_2 = t_1 + Mux\_delay$, Mux4 transmits the value in PC to MemAddrBus, and Mux2 transmits the value in PC to the ALU input A.

(d) At $t_3 = t_2 + ALU\_delay$, the ALU delivers the result from the operation A+4. This value is now available at the input of PC.
PS:ALU operates when there is a change in A, B or ALUop. If Mux2 does not change the value of A, then t_3 can be calculated as $t_3 = t_1 + ALU\_delay$

(e) At $t_4 = t_0 + memRd\_delay(2.5ns : ControlSubsystem) + Tmem(8ns : Memory)$, the signal MemRdy=1 and MemData=Instruction.

(f) At $t_5 = t_4 + Switch\_delay$, the instruction appears at the output of the switch module and is ready to be loaded in IR.

(g) At $t_6 \geq t_5$ the clock pulse arrives, triggering the update of IR, PC, and Status. Note that the signal MemRdy = 1 is not used, because we have assumed that the access to memory always fits in one cycle.

2. **Execute** (see Figure 9):

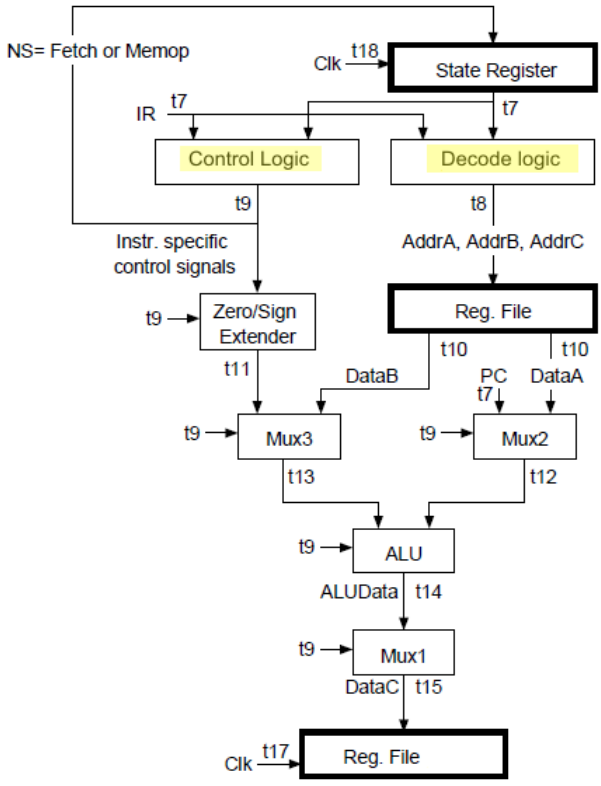

Figure 9: Dependencies for state execute.

(a) At $t_7 = t_{clk} + Reg\_delay$, signals IR_out=Instruction, PCout=updatedPC, and SR_out=Execute.

(b) At $t_8 = t_7 + Dec\_delay(3)$, control signals AddrA, AddrB, and AddrC aree transmitted to the data section.

(c) At $t_9 = t_7 + Ctrl\_delay(0.5)$, the decoding of the instruction is completed. Signals PC_RA, ZE_SE, IR_RB, ALUop, ALU_PC, WrCR, WrC, and WrPC are set according to the instruction being executed. Also the input value for SR (next state value = Memop = 100 or Fetch = 001) is produces, depending whether the instruction is a memory operation or not.

(d) At $t_{10} = t_8 + RF\_delay$, signals DataA and DataB receive the values stored in the registers addressed by signals AddrA and AddrB.

(e) At $t_{11} = t_9 + Ext\_delay$, the output of the Zero/Sign Extender becomes stable.
PS: Extender operates when there is a change in X_in or ZE_SE. If ZE_SE does not change in execute section then t_{11} can be calculated as $t_{11} = t_7 + Ext\_delay$

(f) At $t_{12} = t_{10} + Mux\_delay$ the output from Mux2 becomes stable; this assumes that $t_{10} > t_9$

(g) At $t_{13} = t_{11} + Mux\_delay$, the output from Mux3 becomes stable, as previously, this assumes that $t_{10} > t_9$. This event completes the signals required by the ALU to perform its operation.

(h) At $t_{14} = max(t_{12}, t_{13}) + ALU\_delay$, the ALU delivers the results from its operation (data and condition). At this time the value is available also at the input of PC.

(i) At $t_{15} = t_{14} + Mux\_delay$, Mux1 transmits signal ALUdata to the input of RF (signal DataC). At this time, it is also available at MemAddr.

(j) At $t_{17} \geq t_{15}$ the next clock pulse arrives, triggering the update of CR, RF, or PC (for branches), and SR.

3. **Memop** (see Figure 10):

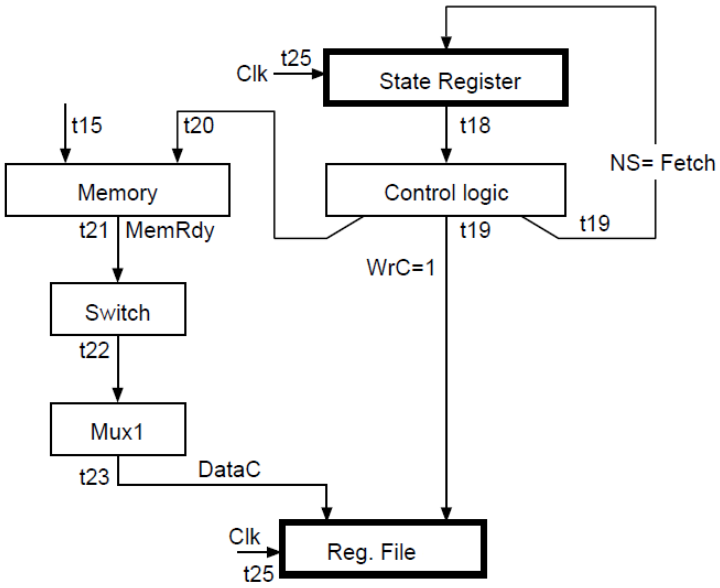

Figure 10: Dependencies for state Memop.

(a) At $t_{18} = t_{clk} + Ctrl\_delay(0.5)$, State = Memop.

(b) At $t_{19} = t_{18} + Ctrl\_delay$, the control logic sets the signals ALU_PC, Mem_ALU, MemLengh, MemEnable, Sin_Sout, and WrC, depending whether the operation is a Load or a Store. MemRd or MemWr becomes stable at $t_{20} = t_{18} + MemRd\_delay(2.5)$ or $t_{20} = t_{18} + MemWr\_delay(2.5$

(c) At $t_{21} = t_{20} + Tmem(8ns : memory)$, signal MemRdy=1 and the memory access is completed.

(d) At $t_{22} = t_{21} + Switch\_delay$, the output from the switch module becomes stable.

(e) At $t_{23} = t_{22} + Mux\_delay$, the output from Mux1 becomes stable and ready to be written in RF.

(f) At $t_{25} \geq t_{23}$, the next clock pulse arrives, tiggering the update of RF, if the instruction is Load, and Status.

The critical path that determines the minimum clock period corresponds to the maximum of the critical paths among all the states. We now show an example of the calculation of the minimum clock period.

We calculate the minimum clock period using the values in Figure 11 for the delay of the different modules:

Register	`Reg_delay`	t_R	2 ns (setup and propagation delay)
Register file	`RF_delay`	t_{RF}	4 ns
ALU	`ALU_delay`	t_{ALU}	6 ns
Multiplexer	`Mux_delay`	t_{mux}	0.5 ns
Zero/sign ext.	`Ext_delay`	t_{ZSE}	0.5 ns
Switch	`Switch_delay`	t_{sw}	0.5 ns
Control delay	`Ctrl_delay`	t_{ctl}	0.5 ns
Decode delay	`Dec_delay`	t_{dec}	3 ns
Memory	`Mem_delay`	t_{mem}	8 ns (static memory)

Figure 11: Delays

$t_{fetch} = t_R + t_{ctl} + t_{mux} + t_{mem} + t_{sw} = 2 + 0.5 + 1 + 8 + 0.5 = 12ns$
$t_{exec} = t_R + t_{ctl} + t_{RF} + t_{mux} + t_{ALU} + t_{mux} + t_{RF} = 2 + 0.5 + 4 + 0.5 + 6 + 0.5 + 4 = 17.5ns$
$t_{memop} = t_R + t_{ctl} + t_{mem} + t_{sw} + t_{mux} + t_{RF} = 2 + 0.5 + 8 + 0.5 + 0.5 + 4 = 15ns$

2.7 Events and Flow of Data During Execution

Let us consider the events and flows of data that take place in the data subsystem during the Execute part of the instruction loop (e.i., the instruction fetch has already been performed and the instruction has been stored in IR).

Let us first consider an arithmetic-logic instruction; the sequence of events (as depicted in Figure 12) are as follows:

1. Signal *Instr* (the output from IR) is sent to the control subsystem for decoding.

2. After the instruction decode delay, the signals that control the modules are activated by the control subsystem; this includes the signals to access RF and the signals to control the ALU, the multiplexers, and the storing of the results into the registers.

3. After the RF read delay, the contents of the registers used as operands by the instruction become available and are fed to the ALU through multiplexers Mux2 and Mux3.

4. After thr ALU delay, the ALU delivers the results from the operation (data and condition); the data result is stored in the RF (through Mux1), and the condition result is stored in CR. These storing operations are synchronized with the clock signal, so that the values must be stable earlier than the stup time of the registers.

   The flow is similar for instructions that have an immediate operand SI or UI, except that the second operand is obtained from IR and extended accordingly.

   In memory operations, the address is calculated in the ALU and sent through the address bus. For a Load instruction the data come through the data bus and are written in RF through port C. In a Store operation, the data are obtained from RF through port B.

All these steps compromise the **critical path** in the execution step of an instruction: the instruction is decoded, the operands read, the operation performed, and the results transmitted to the input of the registers before the arrival of the next clock pulse (complying with the setup time of the registers). Moreover, in memory operations the access to memory has to be included. Depending on the implementation of the control subsystem, this critical path might be required to fit in one or in several clock cycles. We consider one such implementation now.

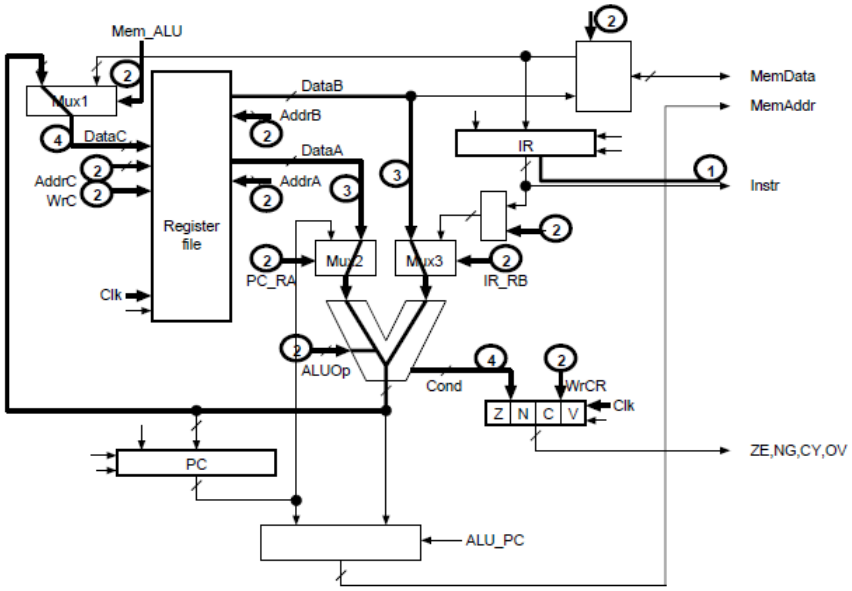

Figure 12: Sequence of events in data subsystem for arithmetic-logic instruction.

2.8 Implementation of the Processor

In processor module, you will write the interconnection between control subsystem and data subsystem, which is given in Figure 13 .

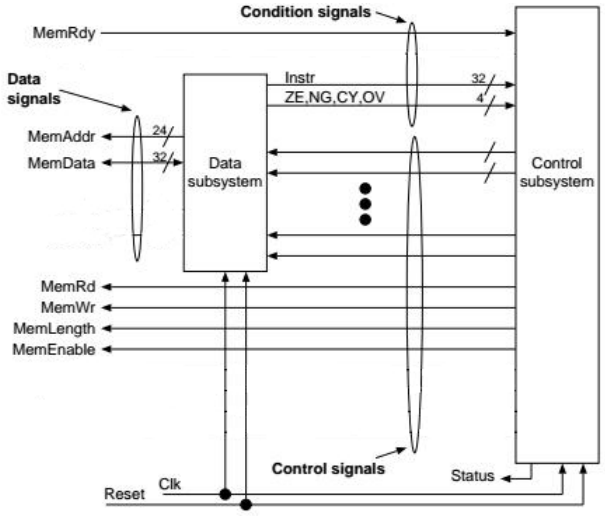

Figure 13: Interconnections in processor module.

The signature of the module is given below:

```
module Processor(
output  [23:0] MemAddr  , // memory address bus
input [31:0] fromMemData  , // data bus from memory
output  [31:0] toMemData, // data bus to memory
output  MemLength, // memory operand length
output  MemRd, // memory read control signal
output  MemWr, // memory write control signal
output  MemEnable, // memory enable signal
input MemRdy, // memory completion signal
output  [2:0] Status, // processor status signal 0:p_reset, 1:fetch, 2:execute, 3:memop
input Reset , // reset signal
input Clk // clock signal
    );
```

2.9 Implementation of the Computer

In computer module, you will write the interconnection between memory and processor, which is given in Figure 14 .

The signature of the module is given below:

```
module Computer(
input Reset,
input Clk
);
```

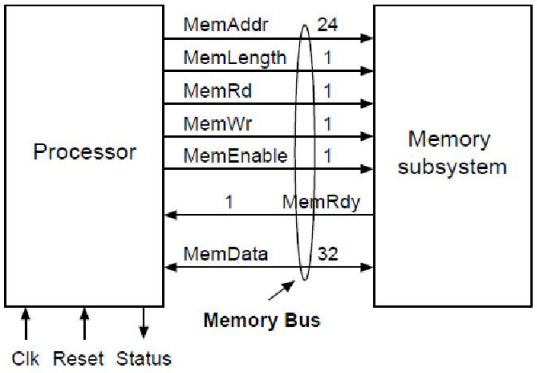

Figure 14: General structure of the computer.

3 Deliverables

- This homework is supposed to be done **individually**, not with your partner. Any kind of cheating is not allowed.
- Implement the modules in 3 files (`Computer.v` ,`Processor.v` ,`Ctrl_Subsystem.v`) and compress them into a single file named `lab5.zip`. **Do not submit your testbenches or additional files.**
- It is your responsibility to write a testbench for the modules. You may share your testbenches on the newsgroup.
- Submit the zipped file through the COW system before the given deadline.
- Use the newsgroup metu.ceng.course.232 for any questions regarding the homework.

4 Hints

- **Hint 1:** In simulation mode, debugging your code helps you to find errors. `https://www.xilinx.com/itp/xilinx10/isehelp/ism_p_using_breakpoints.htm`
- **Hint 2:** In simulation mode, Click View− >Panels− >Memory to see the contents of the memory or Register File.
- **Hint 3:** We will provide you a **vhdl** version of the microcomputer system. You can use this vhdl code as a reference when you are implementing. It is not a must but it can definitely help. So, you are suggested to benefit from this compressed folder.