EE2401 微算機系統 Fall 2019

HW#6 (ARM) (11/12/2019)

Due date: 11/28/2019. Severe penalty will be given to late homework.

Note:

(a) The homework will be graded based on your **answer**. Please read class handouts (ARM-1)

(b) You are required to **type** your homework (first the problem then your solution) by using a **word processor** and submit in .docx (or .pdf) format under a filename **EE2401f19-hw6-student_no-vn.docx (or .pdf)**, where **student_no** is your student number, e.g., **107061xxx** and **vn** is your version number, e.g., **v3**. You should upload your .doc file in **iLMS** by the specified deadline whenever you have a newer version. Follow the iLMS upload homework process to upload your file.

(c) The homework will be graded based on your **latest version**. Old version(s) will be discarded.

(d) Each homework assignment will have full score of 100 points. 5 points will be deducted if you do not comply with the naming convention. Severe grade penalty will be given to late homework. 20 points will be taken off per day after deadline till zero point. ***Copying is violating the regulations and is definitely not allowed!***

(e) Please treat the above requirements as a kind of training in writing a decent homework report. If you have any problem regarding this homework, **please feel free to consult with TA or teacher**. If you think the time is too short to accomplish this homework, please let me know in class.

1. (MU0 Extension Design) (70%)

The original 8 instructions of MU0 are shown below. In the class, we discussed two versions of RTL organizations and associated state transition table for MU0, shown in Fig.1-Table 1 and Fig.2-Table 2, respectively. The minimum extension that is useful is to introduce **a new 12-bit index register (X)** and some **new instructions** that allow X to be **initialized** and **used in load, store, add and sub instructions**. Recall that we have **8 unused opcodes** in the original design, so we would add up to eight new instructions before we run out of space. Assume we want to **add only seven new instructions** for indexing operations as described below:

The basic set of indexing operations:

```
LDX  S      ;X := mem16[S]
LDA  S,X    ;ACC := mem16[S+X]
STO  S,X    ;mem16[S+X] :=ACC
ADD  S,X    ;ACC := ACC + mem16[S+X]
SUB  S,X    ;ACC := ACC - mem16[S+X]
```

More for modifying X:

```
INX         ;X := X+1
DEX         ;X := X-1
```

In this homework problem, we are going to add these extensions to the two versions of MU0 implementation.
For version 1 (2 clock cycles for each instruction),

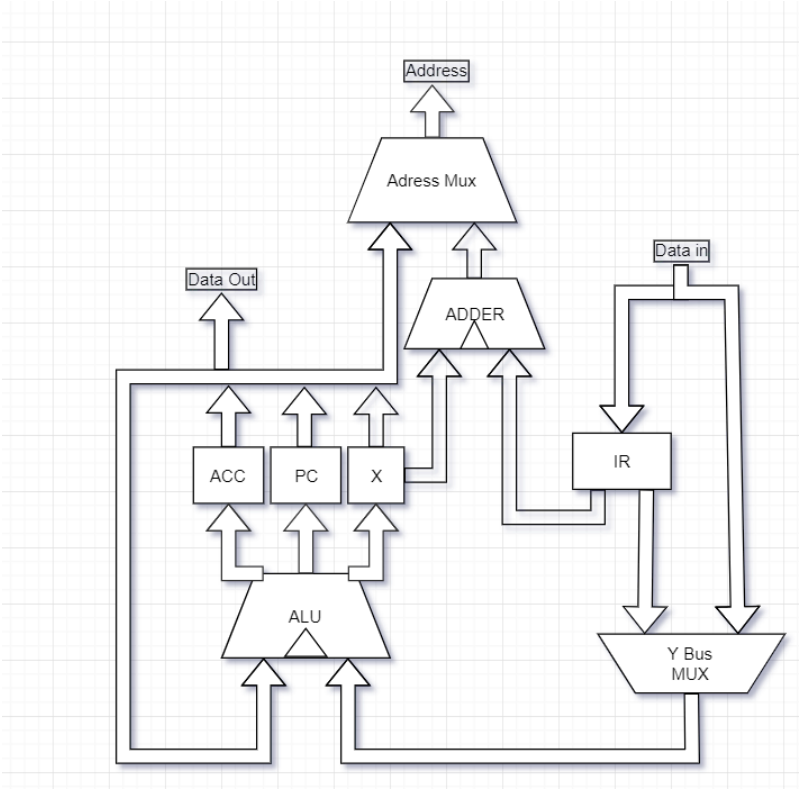
(a) (15%) Modify the RTL organization shown in Fig.1 to include the X register, indicating the new control signals required.

For version 1, it will first fetch, then decode and execute.

We include the x register above the ALU, and add an additional adder to modify the output instruction. The x register can accept input from the ALU and output data to the additional adder or to the data bus.

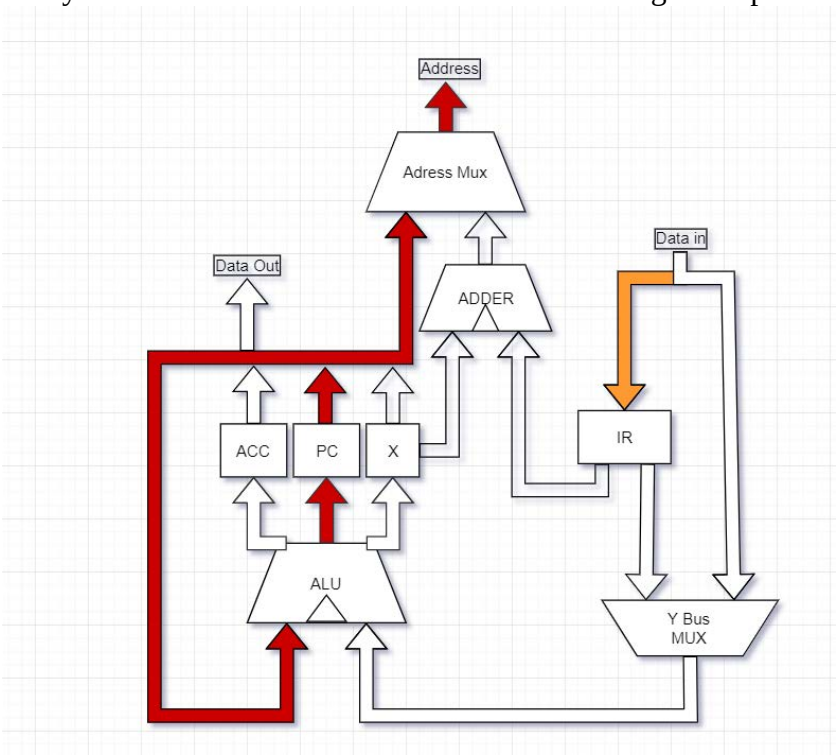
We will add an additional decrement function to the ALU (X = X – 1)

Overall design:



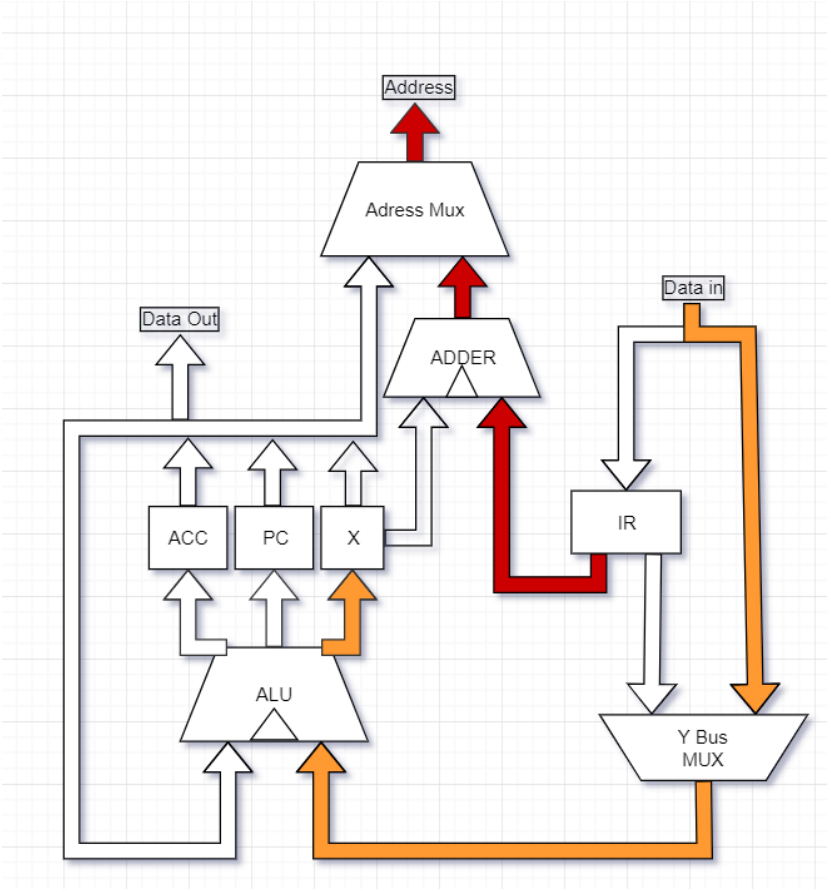
Let's try to verify each additional instruction.
We will use the rainbow color to indicate the order, i.e. red, orange, yellow…… and so on.

Every instruction has the same “instruction fetching” data path.



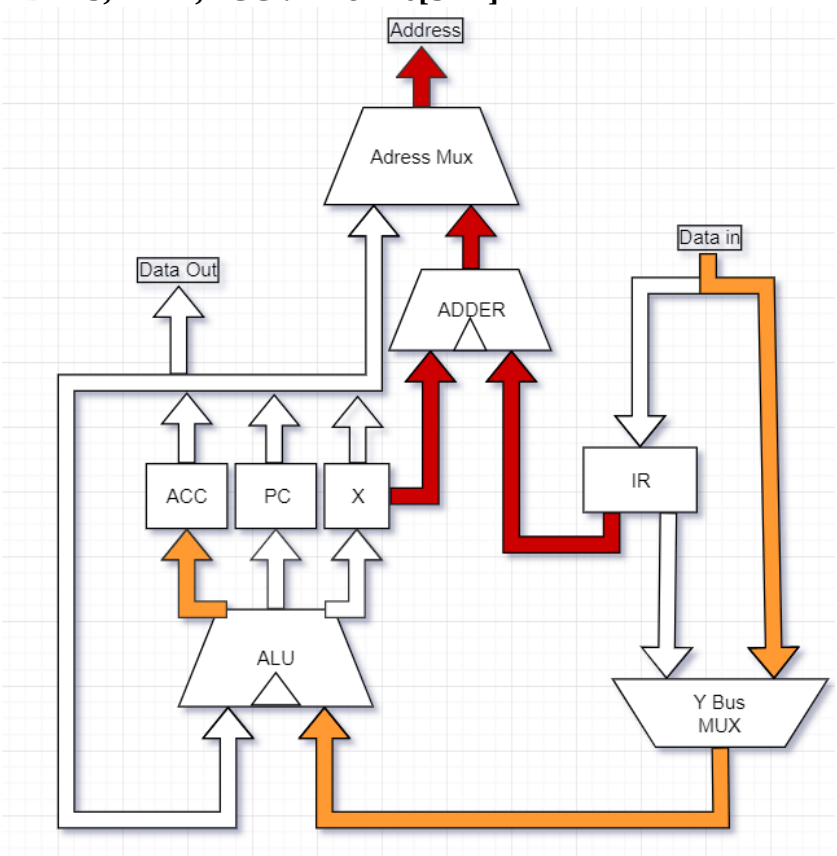
The program counter will +1 first and then go to memory to fetch the instrction. Then, the instrction will be read and go into the instrction register.

LDX S ;X := mem16[S]



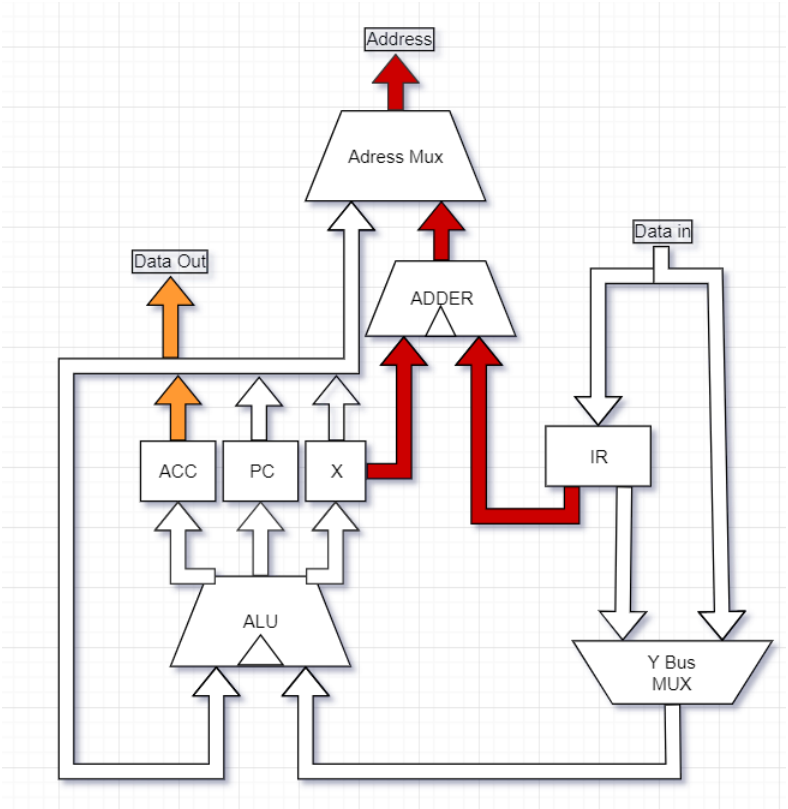
Instruction will be decoded, and sent to the address. Then the memory data will be sent in and store inside X.

LDA S,X ;ACC := mem16[S+X]



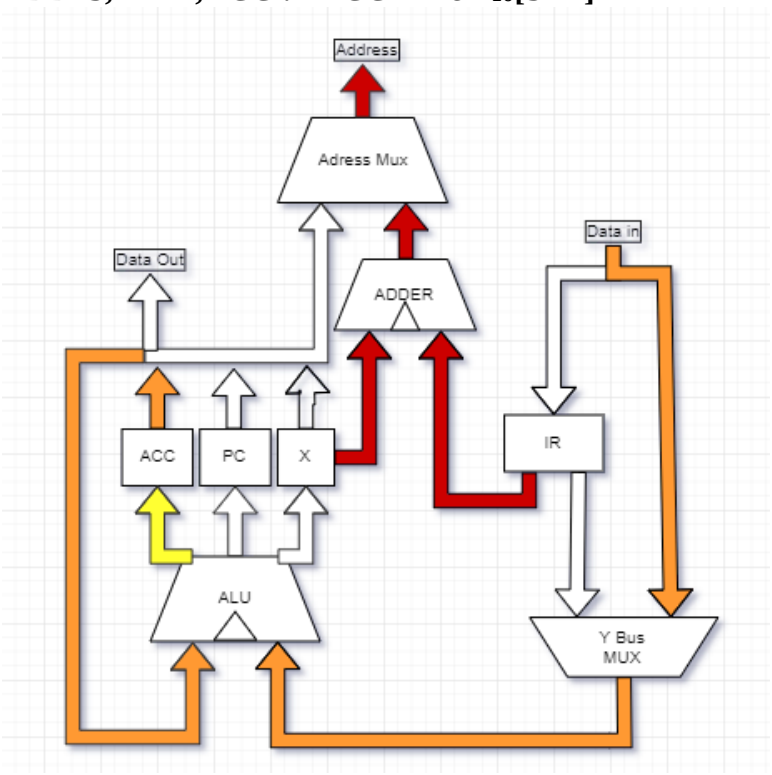
X will be added to the decoded instruction, in order to get the memory in S + X. Then, the data will be sent in and stored inside ACC.

STO S, X ;mem16[S+X] :=ACC



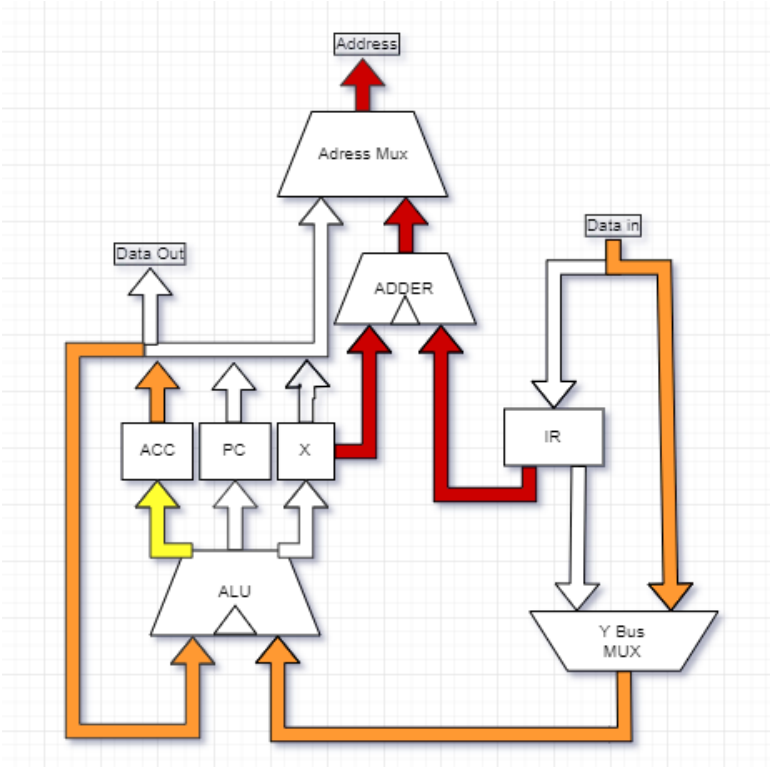
Since we want the data in the position of S+X, we have to add X and the decoded instruction to fetch the correct position. Once the position is found, the data in ACC will be stored in S + X.

ADD S,X ;ACC := ACC + mem$_{16}$[S+X]



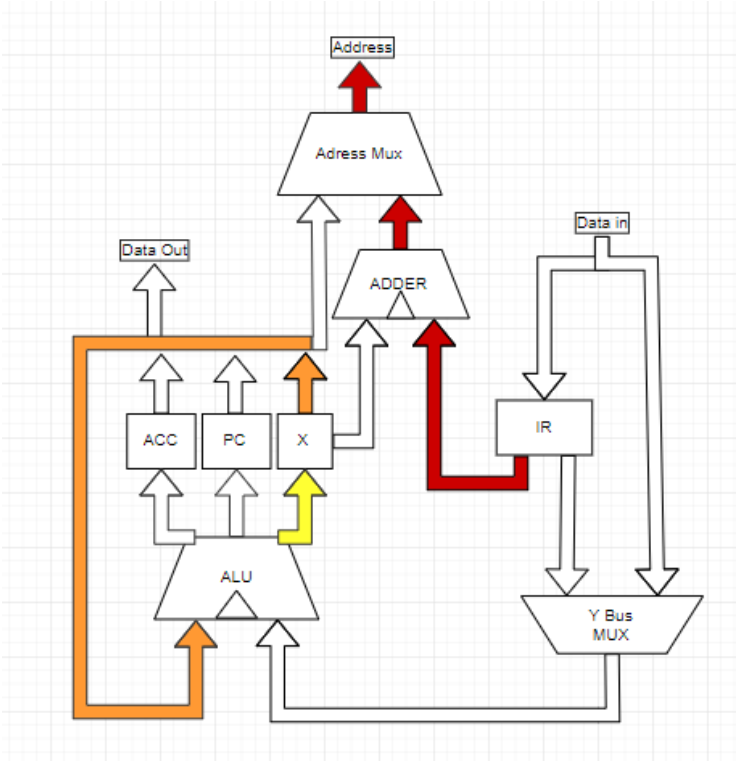
Similar to previous, the position is set at S + X, hence the decoded instruction should be added by X. Then, we will add the data inside the assigned position with the ACC. After the addition by the ALU, the result will be stored to the ACC.

SUB S,X ;ACC := ACC - mem$_{16}$[S+X]



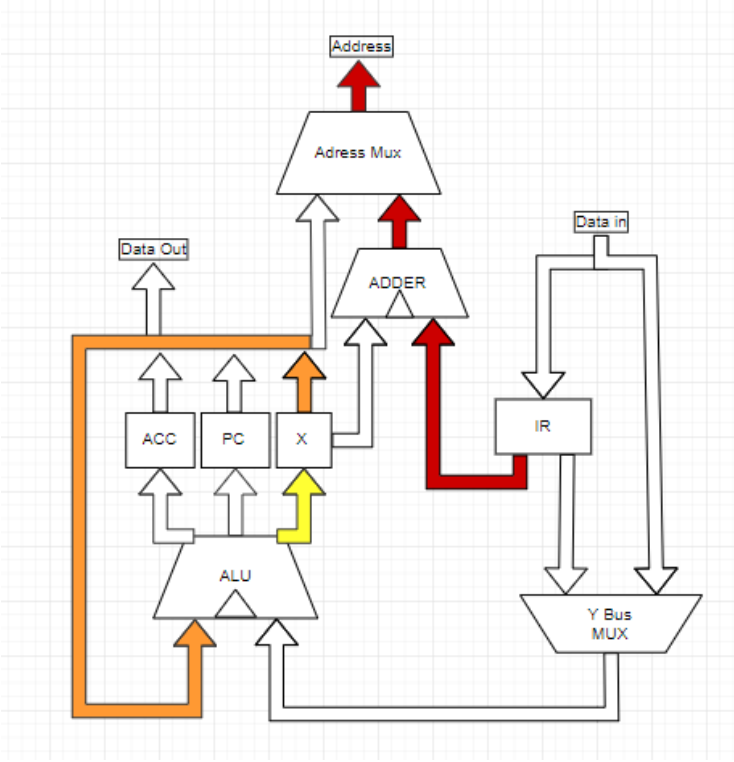
Similar to previous, but what the ALU will perform is a subtraction.

INX ;X := X+1



We first decode the instruction, then X will be sent to the data bus, and sent to the ALU. The ALU will perform an increment, and finally sent back to X.

DEX ;X := X-1



Similar to the previous, but the ALU will perform a decrement (The decrement function will be added to our ALU, we will discuss it further in the control logic table)

(b) (20%) Modify the control logic in Table 1 to support indexed addressing.

We have to include several control logics for the X register, XEn for the clock enable and X1OE for the output enable to the address bus, X2OE for the output enable to the adder.

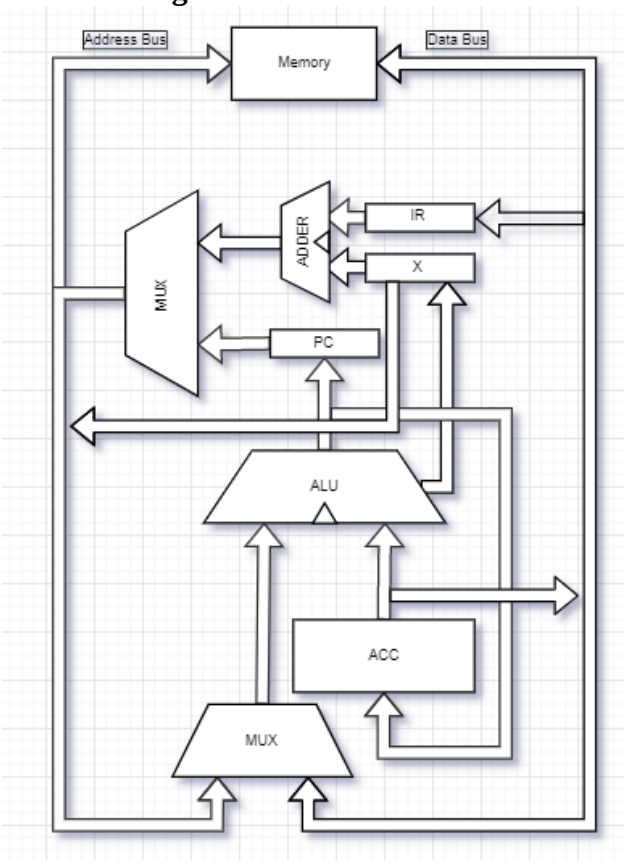
	State	F[3:0]	Next State	IREn	PCEn	AccEn	M[2:0]	AccOE	PCOE	Ydin	Asel	DoutOE	Read	Write	Xen	X1OE	X2OE
LDX S	1	1000	0	0	0	0	0	0	0	1	1	0	1	0	1	0	0
LDA S, X	1	1001	0	0	0	1	0	0	0	1	1	0	1	0	0	0	1
STO S, X	1	1010	0	0	0	0	ZZ	1	0	Z	1	1	0	1	0	0	1
ADD S, X	1	1011	0	0	0	1	1	1	0	1	1	0	1	0	0	0	1
SUB S, X	1	1100	0	0	0	1	11	1	0	1	1	0	1	0	0	0	1
INX	1	1101	0	0	0	0	10	0	0	0	1	0	0	0	1	1	0
DEX	1	1110	0	0	0	0	100	0	0	0	1	0	0	0	1	1	0

For version 2 (2 clock cycles for some instructions and 1 clock cycle for the remaining instructions),

(c) (15%) Modify the RTL organization shown in Fig.2 to include the X register, indicating the new control signals required.

For version 2, it will first decode and execute the instruction (the current instruction was fetched in the previous stage), then fetch the next instruction.

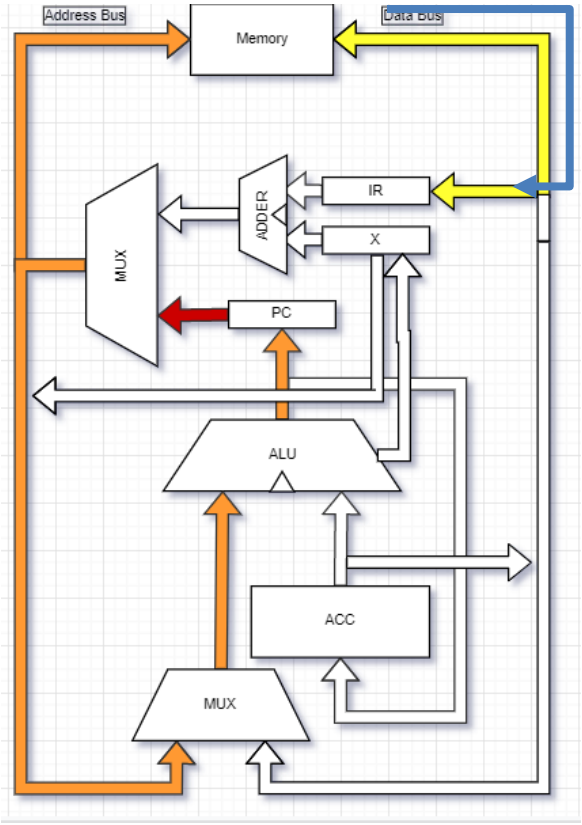
Overall Design:



The x register can accept input from the ALU, and send the data either to the adder or directly to the address bus. An adder will be added in order to execute the indexed addressing instruction.

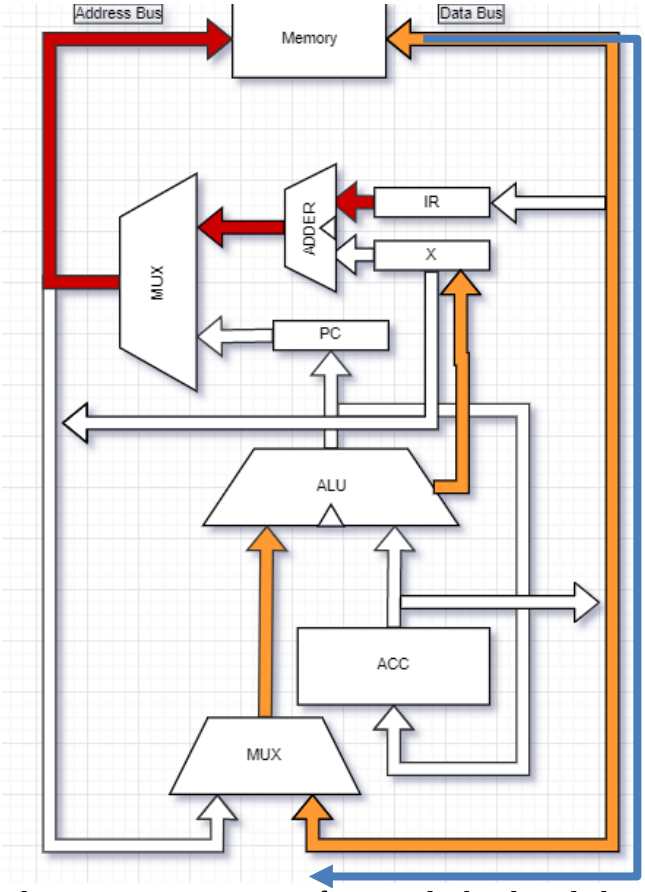
We will add an additional decrement function to the ALU (X = X – 1)
We then verify all the additional instructions.

Instruction Fetching:



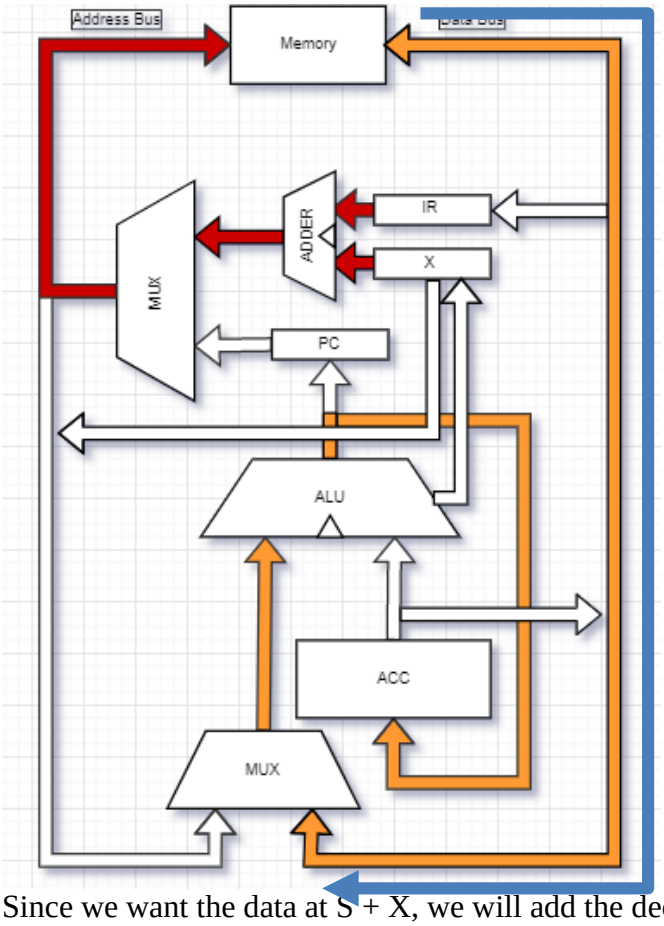
Similar to the version 1, all the “fetching instruction” data path are the same.
The program counter sends the current position to fetch the correct instruction, and the program counter will be added 1. Then, the correct instruction will be sent to the instruction register.

LDX S ;X := mem16[S]



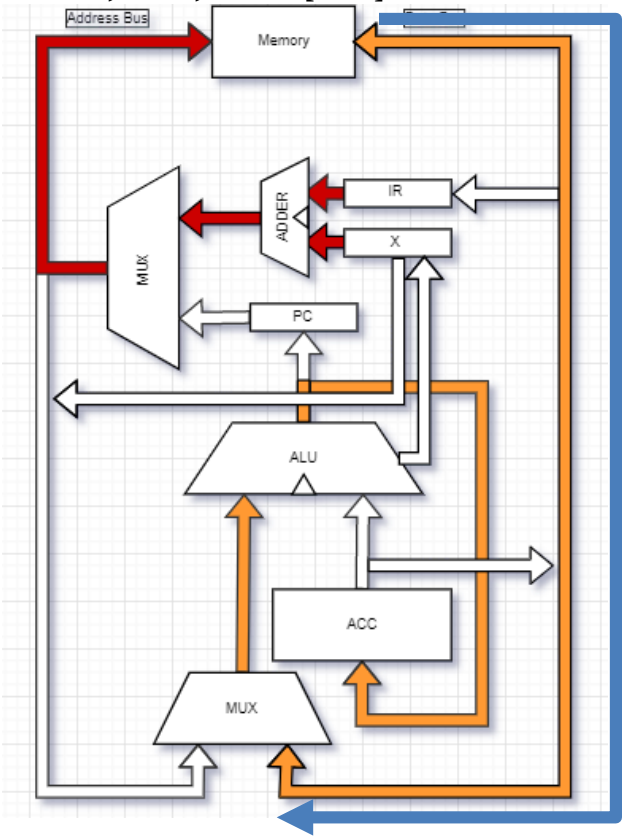
The instruction register first sends the decoded instruction, then send the required memory data to register X.

LDA S,X ;ACC := mem16[S+X]



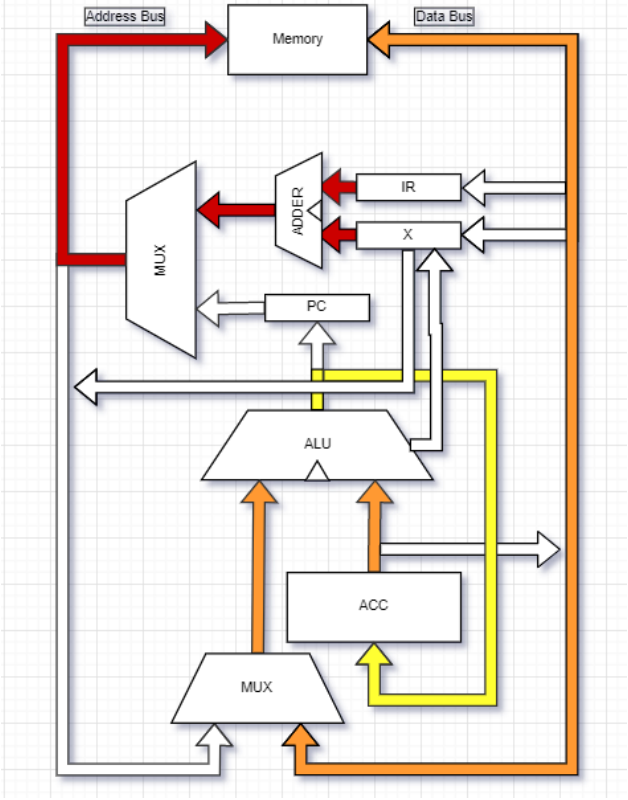
Since we want the data at S + X, we will add the decoded instruction with the X register in order to fetch the required data. Then the required data will be sent to ACC.

STO S,X ;mem16[S+X] :=ACC



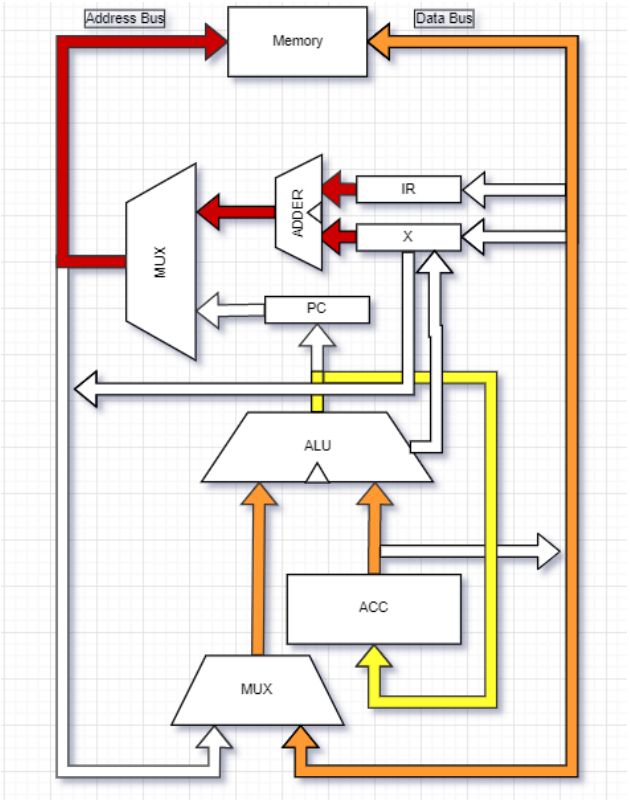
Similarly, X and IR will be added together to fetch the correct memory position, then the data in ACC will be send to that position.

ADD S,X ;ACC := ACC + mem16[S+X]



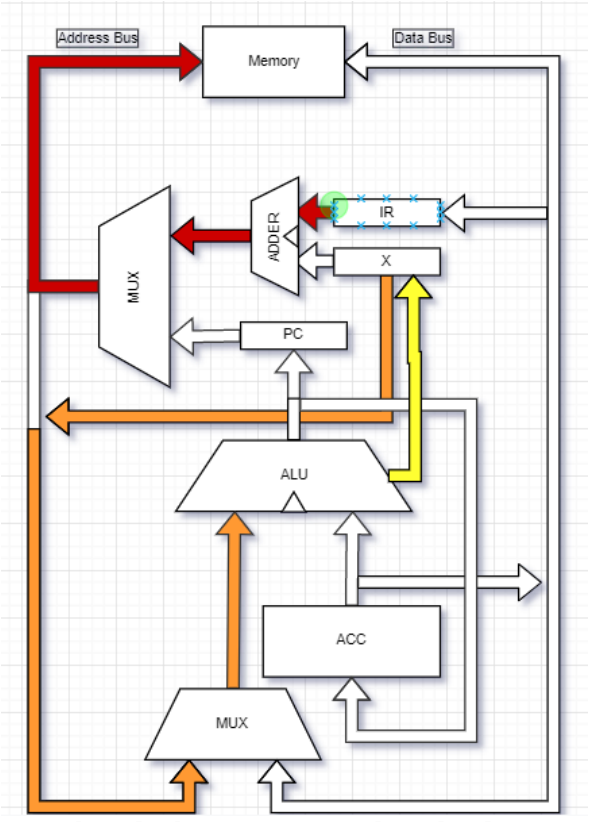
Similar to previous, first we get the required data in memory S + X using the adder, then the required data comes out from the memory. Next, the data will be added with the data in the ACC using the ALU. Finally, store the result inside the ACC.

SUB S,X ;ACC := ACC - mem16[S+X]



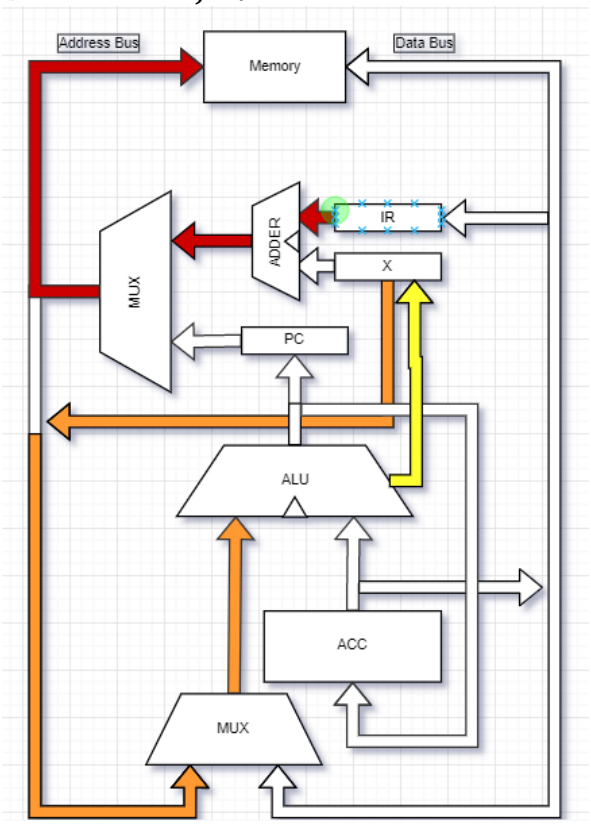
Similar to previous, but use the subtraction function inside the ALU.

INX ;X := X+1



First decode the instruction, then the data inside X register will be send to ALU to increment and will be sent back to X as a result.

DEX ;X := X-1



Similar to previous, but using a decrement function with the ALU.

(d) (20%) Modify the control logic in Table 2 to support indexed addressing.

Instruction	Opcode	Reset	Ex/ft	ACCz	ACC15	Asel	Bsel	ACCce	Pcce	Irce	ACCoe	ALUfs	MEMrq	RnW	Ex/ft
Reset	xxxx	1	x	x	x	0	0	1	1	1	0	0	1	1	0
LDX S	1000	0	0	x	x	1	x	0	0	0	0	B	1	1	1
	1000	0	x	1	x	0	0	0	1	1	0	B+1	1	1	0
LDA S, X	1001	0	0	x	x	1	1	1	0	0	0	B	1	1	1
	1001	0	x	1	x	0	0	0	1	1	0	B+1	1	1	0
STO S, X	1010	0	0	x	x	1	1	1	0	0	0	B	1	0	1
	1010	0	x	1	x	0	0	0	1	1	0	B+1	1	1	0
ADD S, X	1011	0	0	x	x	1	1	1	0	0	1	A+B	1	1	1
	1011	0	x	1	x	0	0	0	1	1	0	B+1	1	1	0
SUB S, X	1100	0	0	x	x	1	1	1	0	0	1	A-B	1	1	1
	1100	0	x	1	x	0	0	0	1	1	0	B+1	1	1	0
INX	1101	0	0	x	x	1	0	0	0	0	0	B	1	1	1
	1101	0	x	1	x	0	0	0	1	1	0	B+1	1	1	0
DEX	1110	0	0	x	x	1	0	0	0	0	0	B-1	1	1	1
	1110	0	x	1	x	0	0	0	1	1	0	B+1	1	1	0

Original 8 instructions of MU0

Instruction	Opcode	Effect
LDA S	0000	ACC := mem_{16}[S]
STO S	0001	mem_{16}[S] := ACC
ADD S	0010	ACC := ACC + mem_{16}[S]
SUB S	0011	ACC := ACC - mem_{16}[S]
JMP S	0100	PC :=S
JGE S	0101	If ACC ≥ 0 PC := S
JNE S	0110	If ACC ≠ 0 PC := S
STP	0111	Stop

Original MU0 RTL Organization

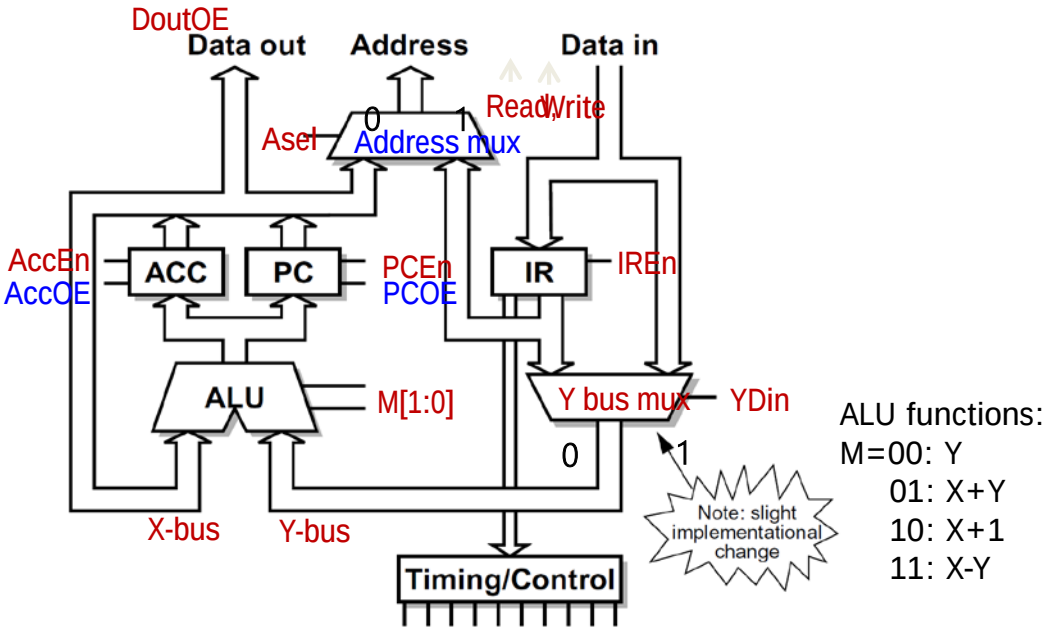


Fig.1 MU0 v1

Table 1 Original control logic for MU0 v1

Flag N, Z State: 0—fetch, 1--execute

	State	F[2:0]	Next state	IREn	PCEn	AccEn	M[1:0]	AccOE	PCOE	YDin	Asel	DoutOE	Read	Write
	0	δδδ	1	1	1	0	10	0	1	δ	0	0	1	0
LDA	1	000	0	0	0	1	00	0	0	1	1	0	1	0
STO	1	001	0	0	0	0	δδ	1	0	δ	1	1	0	1
ADD	1	010	0	0	0	1	01	1	0	1	1	0	1	0
SUB	1	011	0	0	0	1	11	1	0	1	1	0	1	0
JMP	1	100	0	0	1	0	00	0	0	0	δ	0	0	0
JGE	1	101	0	0	$\overline{N}$	0	00	0	0	0	δ	0	0	0
JNE	1	110	0	0	$\overline{Z}$	0	00	0	0	0	δ	0	0	0
STP	1	111	1	0	0	0	δδ	0	0	δ	δ	0	0	0

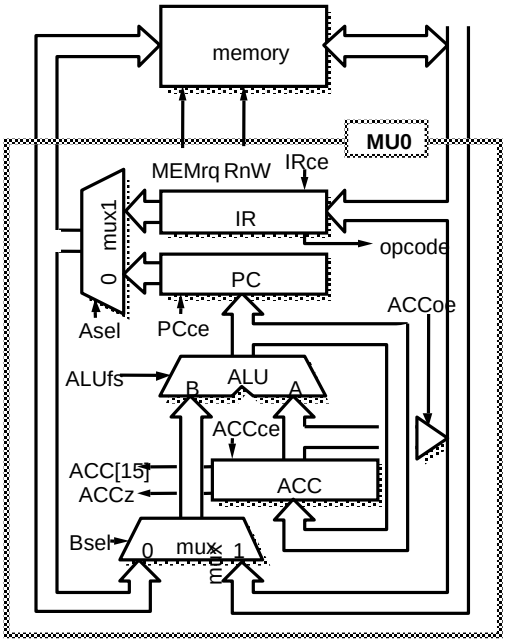


Fig. 2 MU0 v2

Table 2. Original control logic for MU0 v2

Inputs						Outputs									
Instruction	Opcode	Reset	Ex/ft	ACC15 ACCz		Asel	Bsel ACCce		PCce IRce		ACCoe	ALUfs	MEMrq RnW		Ex/ft
Reset	xxxx	1	x	x	x	0	0	1	1	1	0	= 0	1	1	0
LDA S	0000	0	0	x	x	1	1	1	0	0	0	= B	1	1	1
	0000	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
STO S	0001	0	0	x	x	1	x	0	0	0	1	x	1	0	1
	0001	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
ADD S	0010	0	0	x	x	1	1	1	0	0	0	A+B	1	1	1
	0010	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
SUB S	0011	0	0	x	x	1	1	1	0	0	0	A–B	1	1	1
	0011	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
JMP S	0100	0	x	x	x	1	0	0	1	1	0	B+1	1	1	0
JGE S	0101	0	x	x	0	1	0	0	1	1	0	B+1	1	1	0
	0101	0	x	x	1	0	0	0	1	1	0	B+1	1	1	0
JNE S	0110	0	x	0	x	1	0	0	1	1	0	B+1	1	1	0
	0110	0	x	1	x	0	0	0	1	1	0	B+1	1	1	0
STP	0111	0	x	x	x	1	x	0	0	0	0	x	0	1	0

2. (10%) Please give a formula and briefly explain the terms in the formula for describing the CMOS circuit power. Meanwhile, please write down the four strategies for low power design.

The total power dissipation P_c, of a CMOS circuit, neglecting the short-circuit and leakage components, is therefore given by summing the dissipation of every gate g in the circuit C:

$$P_c = \frac{1}{2} \cdot f \cdot {V_{dd}}^2 \cdot \sum_{g \epsilon C} A_g \cdot C_L^g$$

Where f is the clock frequency, A_g is the gate activity factor (reflecting the fact that not all gates switch every clock cycle) and C is the load capacitance. Note that within this summation clock lines, which make two transitions per clock cycle, have an activity factor of 2.

The typical gate load capacitance is a function of the process technology and therefore not under the direct control of the designer. The remaining parameters in Equation 3 suggest various approaches to low-power design. These are listed below with the most important first:

(1) Minimize the power supply voltage, V_{dd}
The quadratic contribution of the supply voltage to the power dissipation makes this an obvious target. This is discussed further below.
(2) Minimize the circuit activity, A
Techniques such as clock gating fall under this heading. Whenever a circuit function is not needed, activity should be eliminated.
(3) Minimize the number of gates.
Simple circuits use less power than complex ones, all other things being equal,

since the sum is over a smaller number of gate contributions.

(4) Minimize the clock frequency, f.
Avoiding unnecessarily high clock rates is clearly desirable, but although a lower clock rate reduces the power consumption it also reduces performance, having a neutral effect on power-efficiency (measured, for example, in MIPS – Millions of Instructions Per Second – per watt). If, however, a reduced clock frequency allows operation at a reduced V_{dd}, this will be highly beneficial to the power-efficiency.

3. (10%) Briefly explain CISC and RISC. Discuss the pros and cons of CISC and RISC as much as you can.

CISC:
CISC stands for "Complex Instruction Set Computer", it has the opposite behavior of the RISC. The common CISC characteristics are:
(1) Dense code, simple compiler
(2) Powerful instruction set, variable format, multi-word instructions
(3) Multi-cycle execution, low clock rate.

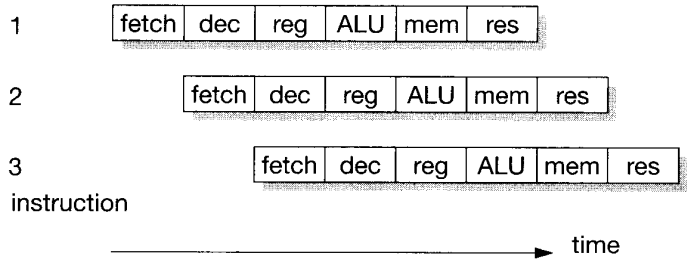
The advantages of CISC:
(1) Microprogramming is easy to implement, using the powerful instruction set.
(2) Great code density
The drawbacks of CISC:
(1) The performance of the machine slows down due to the amount of clock time taken by different instructions will be dissimilar
(2) Only 20% of the existing instruction is used in a typical programming event, even though there are various specialized instructions, in reality, are not used frequently.
(3) The conditional codes are set by the CISC instructions as a side effect of each instruction which takes time for setting – and, as the subsequent instruction changes the condition code bits – so, the compiler has to examine the condition code bits before this happens.

RISC:
RISC stands for "Reduced Instruction Set Computer", it has the opposite behavior of the CISC. The common RISC characteristics are:
(1) A fixed instruction size with few formats
(2) A load-store architecture where instructions that process data operate only on registers and are separate from instructions that access memory.
(3) A large register bank of thirty-two 32-bit registers, all of which could be used for any purpose, to allow the load-store architecture to operate efficiently.
(4) Single cycle instruction
The advantages of RISC:
(1) A smaller die size – RISC is simpler to CISC hence will require fewer transistors and less silicon area.
(2) A shorter development time – Since the RISC is simpler in the architecture prospect, it will take less design effort and therefore have a lower design cost.
(3) A higher performance – A simpler processor ought to allow a high clock rate.
The drawbacks of RISC:
(1) Poor code density compared with CISC's
(2) Cannot execute x86 code

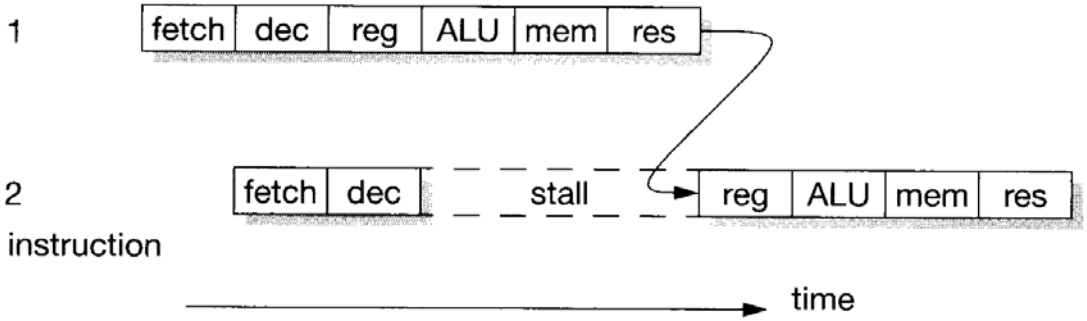
4. (10%) Assume a six-stage pipeline of executing an instruction as follows:

Fetch: (Fetch the instruction from memory).
Dec: (Decode it to see what sort of instruction it is).
Reg: (Access any operands that may be required from the register bank).
ALU: (Combine the operands to form the result or a memory address).
Mem: (Access memory for a data operand, if necessary).
Res: (Write the result back to the register bank).

In principle, such a pipeline should deliver a **six times speed-up** compared with non-pipelined instruction execution as shown below. However, in practice, pipeline could be stalled due to some reasons like **pipeline hazards**.

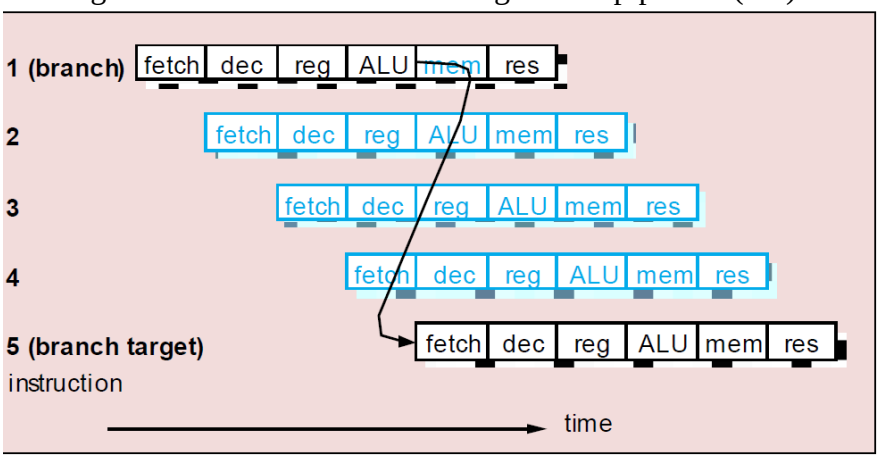


(i) Please explain by drawing a figure like the above one the **read-after-write pipeline hazard** for this six-stage pipeline. (5%)



The Read-after-write(RAW) hazard refers to a situation where an instruction refers to a result that has not yet been calculated or retrieved. Like the figure above, instruction 2 depends on the result of instruction 1, but instruction 1 hasn't finish its execution, causing a pipeline stall.

(ii) Repeat (i) for the **pipeline stall caused by branch instruction**, assuming the branch target is calculated in the ALU stage of the pipeline. (5%)



When using branching in pipelines, each branch will be executed whether or not the branch is the target. We can see that the target is the last to be executed, hence this will cause the worst case for the branch, the other non-target instruction will be abandoned acting as a “pipeline stall”.