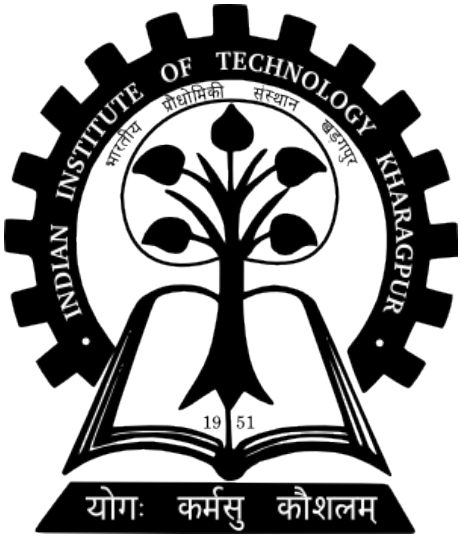


Indian Institute of Technology Kharagpur

Department of Computer Science and Engineering

Design of a one instruction CPU

Hardik Soni
20CS30023

Assignment 1

Computer Organization and Architecture (CS31007)

September 18, 2024

Contents

1 Assignment Statement

1.1 Problem Statement

1. Study the one instruction CPU ISAs and select one.
2. Code one multiplication or division routine using the chosen instruction
3. Design the CPU data paths for your one instruction CPU
4. Develop the controller specifications to orchestrate the data path components so that the required instruction is properly executed

1.2 Abstract

1.2.1 One Instruction Set Computer (OISC)

OISC is the One Instruction Set Computer (by analogy with RISC and CISC), a machine providing only one instruction. The abbreviation URISC (Ultimate RISC) has been used in some publications with the same meaning as OISC.

A general focus of OISCs is to be the extremely simple and yet meaning full (i.e. to allow memory, loops and computation as you know it). A significant divide has emerged between two schools of instruction set design, commonly referred to as RISC (reduced instruction set computer architecture) and CISC (complex instruction set computer architecture). The distinction between these schools existed long before the titles were given. This contrast is particularly clear when comparing the Data General Nova (RISC) and DEC PDP-11 (CISC) architectures created in the late 1960s.

2 Explanation of Chosen Instruction

To implement the OISC(One Instruction Set Architecture), the Instruction I have chosen is **Subtract and branch if less than or equal to zero(subleq)**.

2.1 Subtract and branch if less than or equal to zero(subleq):-

The subleq instruction subtracts the contents at address a from the contents at address b, stores the result at address b, and then:-

- if the result is not positive, transfers control to address c.
- if the result is positive, execution proceeds to the next instruction in sequence.

2.1.1 PsuedoCode

```
Instruction subleq a, b, c
    Mem[b] = Mem[b] - Mem[a]
    if (Mem[b] <= 0)
        goto c
```

By making the third operand equal to the address of the subsequent instruction in the sequence, conditional branching can be prevented. The suppression of this operand is inferred if the third operand is not written.

With two operands and an internal accumulator, a variation is also feasible in which the accumulator is deducted from the memory address designated by the first operand. The second operand indicates the branch address, and the outcome is recorded in both the accumulator and the memory location:

```
Instruction subleq2 a, b
    Mem[a] = Mem[a] - ACCUM
    ACCUM = Mem[a]
    if (Mem[a]     0)
        goto b
```

2.2 Synthesized instructions

2.2.1 JMP c

Unconditional branch:

```
subleq Z, Z, c
```

2.2.2 ADD a, b

Addition can be performed by repeated subtraction, with no conditional branching; e.g., the following instructions result in the content at location a being added to the content at location b:

```
subleq a, Z
subleq Z, b
subleq Z, Z
```

The first instruction subtracts the content at location a from the content at location Z (which is 0) and stores the result (which is the negative of the content at a) in location Z. The second instruction subtracts this result from b, storing in b this difference (which is now the sum of the contents originally at a and b); the third instruction restores the value 0 to Z.

2.2.3 MOV a, b

A copy instruction can be implemented similarly; e.g., the following instructions result in the content at location b getting replaced by the content at location a, again assuming the content at location Z is maintained as 0:

```
subleq b, b
subleq a, Z
subleq Z, b
subleq Z, Z
```

2.2.4 BEQ b, c

```
  subleq b, Z, L1
  subleq Z, Z, OUT
L1:
  subleq Z, Z
  subleq Z, b, c
OUT:
  ...
```

3 Explanation of Multiplication Routine Used and Encoding of MUL a, b, c as subleq

I have encoded the instruction MUL a b c wherein the OISC will compute the multiplication of a and b, and then store into the memory location pointed to by 'c'.

3.1 Explanation and Pseudo-Code

The Algorithms works in this Fashion:-

3.1.1 Implementation Details

- NR is a reserved memory location to store -1.
- Z, c, d refer to temporary utility memory locations, used for the program.
- a, b refer to the numbers to be multiplied.

1. First the program checks whether or not a is non-negative or not, it it is, it proceeds L2 or else L1.
2. L1 reverses the sign of both a and b, i.e, a = -a and b = -b.
3. L2 moves the value of a to c and b to d respectively.

3.2 Code

```
# here NR is the standard memory location reserved for storing -1
subleq Z, Z                 # Z <- 0
subleq c, c                 # c <- 0
subleq a, Z, L2             # jumps to L2 if a >= 0
#   a <- -a
#   b <- -b
```

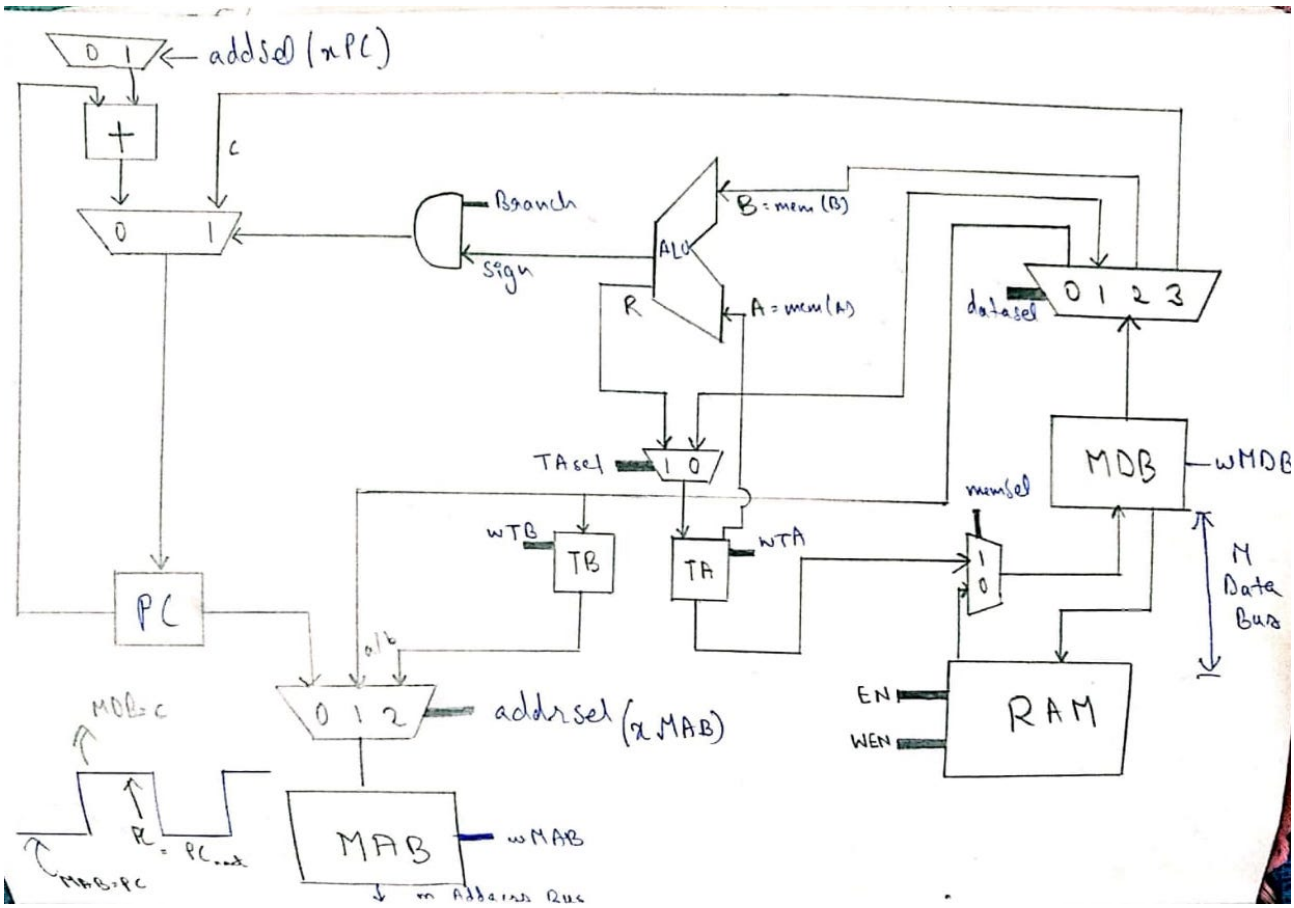
```
1  L1:                      #   Swaps the signs of a and b and
       proceeds to L2
2      subleq a, Z                  # Z <- -a
3      subleq a, a                  # a <- 0
4      subleq Z, c                  # c <- a
5      subleq c, a                  # a <- -a
6      subleq Z, Z                  # Z <- 0
7      subleq c, c                  # c <- 0
8      subleq b, Z                  # Z <- -b
9      subleq b, b                  # b <- 0
10     subleq Z, c                  # c <- b
11     subleq c, b                  # b <- -b
12 subleq c, c                  # c <- 0
13 subleq Z, Z                  # Z <- 0
14 L2:                      #   Stores a and b in c and d respectively
       and proceeds to E1
15     subleq a, Z                  # Z <- -a
16     subleq Z, c                  # c <- a
17     subleq Z, Z                  # Z <- 0
18     subleq d, d                  # d <- 0
19     subleq b, Z                  # Z <- -b
20     subleq Z, d                  # d <- b
21     subleq b, b                  # b <- 0
22 #    c <- a
23 #    d <- b
24 #    b <- 0
25 E1:
26     subleq Z, Z                  # Z <- 0
27     subleq NR Y                  # Y <- -(-1) = 1
28     subleq Y, a, Exit            # jumps to Exit if a-1 <= 0
29     subleq d, Z                  # Z <- -d
30     subleq Z, b                  # b <- b+d
31     subleq Z, Z                  # Z <- 0
32     subleq Z, Z, E1              # jumps to E1
33 #    b <- b*a
34 Exit:
35     ......
```

3.2.1 Explanation of Code

- **Line 2:-** Z is initialised to Zero.
- **Line 3:-** c is initialised to Zero.
- **Line 4:-** If a is non-negative jump to L2.
- **Line 8-11:-** Negate a.
- **Line 12:-17-** Negate b.
- **Line 21-22:-** c is set to a.
- **Line 24-26:-** d is set to b.
- **Line 32-34:-** If a<=1 terminate the multiplication loop.
- **Line 34-36:-** Incrementally add d which adds d(b) to b, until a is > 1 and then to the start of the loop.

4 CPU Data Path

4.1 Circuit Diagram



	addrSel	EN	WEN	TASel	dataSel	Branch	addSel	MemSel	wTA	wTB
1	0	1	0	x	0	0	0	0	0	0
2	1	1	0	0	1	0	1	0	1	0
3	0	1	0	x	0	0	0	0	0	1
4	1	1	0	1	2	0	0	0	1	0
5	2	1	1	x	x	0	1	1	0	0
6	0	1	0	x	3	1	1	0	0	0

5 Controller Requirements for Orchestrating the Data Path Components

Each **subleq** operation takes 6 cycles to complete:-

- The instruction address from PC will be sent to **MAB** in the first clock cycle, and we will take address a from **MDB** and be ready to load it back into **MAB** for the next clock cycle.

  - MAB ← PC [xMAB = 0, wMAB = 1]
  - MDB ← a (address of a) [memSel = 1, wMDB = 1]

- The value at address a will be obtained in the second clock cycle, and we will store it in **TA**. We will also raise the Program counter by 4 in order to obtain the value at address b in the third clock cycle.

  - MAB ← a [xMAB = 1, wMAB = 1]
  - MDB ← mem[a] (value at a) [memSel = 1, wMDB = 1]
  - TA ← MDB [wTA = 1, TASel = 0]
  - PC ← PC + 4 [xPC = 1]

- The value at address a will be obtained in the second clock cycle, and we will store it in **TA**. We will also raise the Program counter by 4 in order to obtain the value at address b in the third clock cycle.

  - MAB ← PC [xMAB = 0, wMAB = 1]
  - MDB ← b (address of b) [memSel = 1, wMDB = 1]
  - TB ← b [wTB = 1]

- Because the **ALU** operates asynchronously. At the end of the fourth cycle, we will have the value of [b], the computed value of [b] - [a], and this computed data will be stored in **TA**. We will also give the **MAB** with the address b so that it can operate at the next posedge.

  - MAB ← b [xMAB = 1, wMAB = 1]
  - MDB ← mem[b] (value at b) [memSel = 1, wMDB = 1]
  - TA ← ALU result (mem[b] - mem[a]) [dataSel = 2, TASel = 1, wTA = 1]

- In the fifth cycle, we will write the value held at **TA** to memory address **b**. In order to get the address c in the following clock cycle, we will also increment the Program Counter by 4.

  - MAB ← TB [xMAB = 2, wMAB = 1]
  - MDB ← ALU result (mem[b] - mem[a]) [memSel = 1, wMDB = 1]
  - MAB ← MDB [wMAB = 1]
  - PC ← PC + 4 [xPC = 1]

- In the sixth cycle, we receive the address c from **MDB** and then compare the value saved at address b to decide whether to jump on address c or increase the counter by 4.

  - MAB ← PC [xMAB = 0, wMAB = 1]
  - MDB ← c (address of c) [memSel = 1, wMDB = 1]
  - PC ← c [if Branch = 1 and sign = 1]
  - PC ← PC + 4 [Either Branch = 0 or sign = 0]

clk	TA	TB	MAB	MDB
1	x	x	instruction(a)	a
2	Mem[a]	x	a	Mem[a]
3	Mem[a]	b	instruction(b)	b
4	Mem[b]-Mem[a]	b	b	Mem[b]
5	Mem[b]-Mem[a]	b	b	Mem[b]
6	Mem[b]-Mem[a]	b	instruction(c)	c

6 Conclusion

1. Each instruction pointed to by pc is believed to be 32 bits long, which means that for the instruction subleq A, B, C, each of A, B, and C is represented by 32 bits.
2. The Processor is designed in such a way that it will need 6 cycles of instruction for execution.
3. In the instruction **subleq A, B, C**:
   (a) The Value of A is taken into the Address Bus and then referenced to a address in the First Clock Cycle and will also load it back into Data Bus.
   (b) The Value of **A** will be loaded from the Data Bus and stored in **TA**.
   (c) The Value of **B** is taken into the Address Bus and then referenced to a address in the Third Clock Cycle and will also load it back into Data Bus.
   (d) The value of **B** is fetched, the subtraction is calculated, the flag is set as the case may be and the result is stored in **TA**.
   (e) In this cycle, the result which is stored in **TA** is put in the address of**B** which is stored and the flag is set accordingly. (0 if B <= 0 else 1)
   (f) In this cycle, the address of **C** is fetched from the instruction and branched to that address if flag is set to 1 else to the next instruction(If C is not present then accumulator will make sure that it jump to the next instruction).
4. After the execution of the instruction, again, the next instruction will be run.