CS704 – Advanced Computer Architecture-II

Solution to Assignment 1

Instructions to Solve Assignments

The purpose of assignments is to give you hands on practice. It is expected that students will solve the assignments themselves. Following rules will apply during the evaluation of assignment.

- Cheating from any source will result in zero marks in the assignment.
- Any student found cheating in any two of the assignments submitted will be awarded "F" grade in the course.
- No assignment after due date will be accepted.

Question 1: Total Points (30)

Design a 16-bit ISA for processor containing the following components:

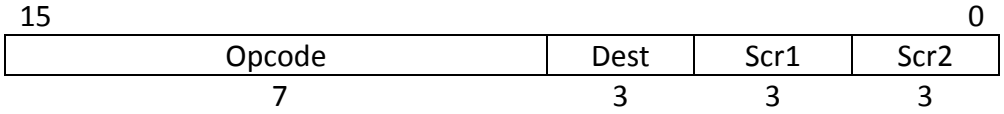
- 8 General Purpose Registers (GPR)
- ALU supporting ADD, SUB, INC, DEC, OR, XOR, AND, NAND, LSHFTR, LSHFTL, ASHFTR, and ASHFTL
- Assume ALU contains saturation and overflow check logic so support ADD and SUB with both options, i.e. overflow and saturate
- LOAD, STORE with two addressing modes Direct and Indirect
- Two control instructions Jump and Call
- NOP instruction

Design means a complete and detailed table containing bit information for every instruction. All 16-bit information must be provided to get full credit.

Solution:

1. Abstract:

In this 16-bit Instruction Set Architecture Design, 9 bits are used for the addresses of three operands including source1, source2 and destination, while rest of the 7 bits are used as fixed numbers of bits for operation code.



Bit Usage:

Bits	Usage
15^{th}	Addressing Mode Specification
14^{th} and 13^{th}	Types of Instructions
12^{th} and 11^{th}	Sub-Types of Instructions
10^{th} and 9^{th}	Operations
8^{th} till 0^{th}	Addresses of three Operands

2. Division of Op-Code:

The 7 bits Op-code is further divided ion to four parts each defined as follows,

2.1. Mode Indication:

As this design support two addressing modes i.e. register-register, direct and indirect modes, and **15^{th} bit** is dedicated to represent the mode, for this purpose '0' indicated the Direct mode while '1' indicates the indirect mode of addressing.

15th bit	
0	Direct Mode
1	In-direct Mode

2.2. Types of Instructions:

The required instructions are divided into four types and two bits 13th and 14th are dedicated to represent the type of instruction.

14th bit	**13th bit**	
0	0	NOP
0	1	Control Inst.
1	0	Load/store Inst.
1	1	ALU Inst.

2.3. Sub-Types of Instructions:

The above indicated four types of instructions are further classified in to sub-types and the two bits **12th and 11th** are dedicated to indicate the sub-type of instructions while the bits **10th and 9th** are dedicated to indicate the particular operation. It must be noted that NOP, Control and Load/store do not need the sub-types because of their simplicity while ALU Operations are classified into subtypes using 12th and 11th bits.

For NOP:

14th	13th	12th	11th	10th	9th	Inst.
0	0	**0**	**0**	**0**	**0**	NOP

For Control Instructions:

14th	13th	**12th**	**11th**	**10th**	**9th**	Inst.
0	1	**0**	**0**	**0**	**0**	Jump
0	1					
0	1					
0	1	**1**	**1**	**1**	**1**	Call

For Load/Store Instructions:

14th	13th	12th	11th	10th	9th	Inst.
1	0	**0**	**0**	**0**	**0**	Load
1	0					
1	0					
1	0	**1**	**1**	**1**	**1**	Store

2.4. ALU Instructions:

These are also classified into subtypes ad follows using 12^{th} and 11^{th} bit and the operations are then indicated using 10^{th} and 9^{th} bits.

14^{th}	13^{th}	12^{th}	11^{th}	Sub-Type
1	1	**0**	**0**	Arithmetic Operations
1	1	**0**	**1**	Shift Operations
1	1	**1**	**0**	Increment / Decrement Operations
1	1	**1**	**1**	Logical Operations

a) Arithmetic Operations:

14^{th}	13^{th}	12^{th}	11^{th}	10^{th}	9^{th}	**Operations**
1	1	0	0	**0**	**0**	ADD Saturate
1	1	0	0	**0**	**1**	ADD Overflow
1	1	0	0	**1**	**0**	Subt Saturate
1	1	0	0	**1**	**1**	Subt Overflow

b) Shift Operations:

14^{th}	13^{th}	12^{th}	11^{th}	10^{th}	9^{th}	**Operations**
1	1	0	1	**0**	**0**	LSHFTR
1	1	0	1	**0**	**1**	LSHFTL
1	1	0	1	**1**	**0**	ASHFTR
1	1	0	1	**1**	**1**	ASHFTL

c) Increment/Decrement Operations:

14^{th}	13^{th}	12^{th}	11^{th}	10^{th}	9^{th}	**Operations**
1	1	1	0	**0**	**0**	Increment
1	1					
1	1					
1	1	1	0	**1**	**1**	Decrement

d) Logical Operations:

14^{th}	13^{th}	12^{th}	11^{th}	10^{th}	9^{th}	**Operations**
1	1	1	1	**0**	**0**	XOR
1	1	1	1	**0**	**1**	OR
1	1	1	1	**1**	**0**	AND
1	1	1	1	**1**	**1**	NAND

3. Instruction Sheets:

Combining all the divisions and concepts discussed above, following is the complete instruction sheet for the assigned 16-bit ISA.

Bit Usage:

Bits	Usage
15th	Addressing Mode Specification (Direct / Indirect)
14th and 13th	Types of Instructions
12th and 11th	Sub-Types of Instructions
10th and 9th	Operations
8th till 0th	Addresses of three Operands

a) Instruction Sheet (Direct Mode – 15th bit '0'):

15th	14th	13th	12th	11th	10th	9th	Instruction
0	0	0	0	0	0	0	NOP
Control Instructions							
0	0	1	0	0	0	0	Jump
0	0	1	1	1	1	1	Call
Load/store Instructions							
0	1	0	0	0	0	0	Load
0	1	0	1	1	1	1	Store
ALU – Arithmetic Instructions							
0	1	1	0	0	0	0	ADD Saturate
0	1	1	0	0	0	1	ADD Overflow
0	1	1	0	0	1	0	Subt Saturate
0	1	1	0	0	1	1	Subt Overflow
ALU – Shift Instructions							
0	1	1	0	1	0	0	LSHFTR
0	1	1	0	1	0	1	LSHFTL
0	1	1	0	1	1	0	ASHFTR
0	1	1	0	1	1	1	ASHFTL
ALU – Increment / Decrement Instructions							
0	1	1	1	0	0	0	Increment
0	1	1	1	0	1	1	Decrement
ALU – Logical Instructions							
0	1	1	1	1	0	0	XOR
0	1	1	1	1	0	1	OR
0	1	1	1	1	1	0	AND
0	1	1	1	1	1	1	NAND

b) Instruction Sheet (In-Direct Mode – 15th bit '1'):

15th	14th	13th	12th	11th	10th	9th	Instruction
1	0	0	0	0	0	0	NOP
Control Instructions							
1	0	1	0	0	0	0	Jump
1	0	1	1	1	1	1	Call
Load/store Instructions							
1	1	0	0	0	0	0	Load
1	1	0	1	1	1	1	Store
ALU – Arithmetic Instructions							
1	1	1	0	0	0	0	ADD Saturate
1	1	1	0	0	0	1	ADD Overflow
1	1	1	0	0	1	0	Subt Saturate
1	1	1	0	0	1	1	Subt Overflow
ALU – Shift Instructions							
1	1	1	0	1	0	0	LSHFTR
1	1	1	0	1	0	1	LSHFTL
1	1	1	0	1	1	0	ASHFTR
1	1	1	0	1	1	1	ASHFTL
ALU – Increment / Decrement Instructions							
1	1	1	1	0	0	0	Increment
1	1	1	1	0	1	1	Decrement
ALU – Logical Instructions							
1	1	1	1	1	0	0	XOR
1	1	1	1	1	0	1	OR
1	1	1	1	1	1	0	AND
1	1	1	1	1	1	1	NAND

4. Addressing Implementations:

Following are few sample implementations with respective addressing modes,

OR (Direct Mode)

15th	14th	13th	12th	11th	10th	9th	8th	7th	6th	5th	4th	3rd	2nd	1st	0th
0	1	1	1	1	0	1	Destination			Source			0	0	0

ADD Saturate (Direct Mode)

15th	14th	13th	12th	11th	10th	9th	8th	7th	6th	5th	4th	3rd	2nd	1st	0th
0	1	1	0	0	0	0	Destination			Source-1			Source-2		

LOAD (In-Direct Mode)

15^{th}	14^{th}	13^{th}	12^{th}	11^{th}	10^{th}	9^{th}	8^{th}	7^{th}	6^{th}	5^{th}	4^{th}	3^{rd}	2^{nd}	1^{st}	0^{th}
1	1	0	0	0	0	0	Source			Offset			0	0	0

STORE (In-Direct Mode)

15^{th}	14^{th}	13^{th}	12^{th}	11^{th}	10^{th}	9^{th}	8^{th}	7^{th}	6^{th}	5^{th}	4^{th}	3^{rd}	2^{nd}	1^{st}	0^{th}
1	1	0	1	1	1	1	Destination			Offset			0	0	0

JUMP (Direct)

15^{th}	14^{th}	13^{th}	12^{th}	11^{th}	10^{th}	9^{th}	8^{th}	7^{th}	6^{th}	5^{th}	4^{th}	3^{rd}	2^{nd}	1^{st}	0^{th}
0	0	1	0	0	0	0	Base			0	0	0	0	0	0

Question 2: Total Points (10)

In reg-mem architecture, clock cycle is 10 ns wide. It is proposed that reg-reg architecture be used instead, that reduces the clock cycle by 2 ns. However, it requires an additional load instruction, in some cases! Will the new processor be more efficient, if so under what circumstances? Quantify your answer.

Solution:

Efficiency of reg-reg will depend upon the resulting increase in instruction count. If it is below a certain threshold level, reg-reg architecture will be more efficient. Assume that we have x instructions originally in reg-mem architecture. Execution time of this architecture is given by (all calculations assume CPI=1)

Execution $time_{old}$ = IC * CPI * CCT = x*1*10 = 10x ns

Now for proposed reg-reg architecture, we need additional load/store instructions. Let s assume that increase in instruction count is y%. Then new execution time will be:

Execution $time_{new}$ = IC * CPI * CCT = (x + y% of x)*1*8

Overall speedup is given by Amdhal's law:

$Speedup_{overall}$ = Execution $time_{old}$ / Execution $time_{new}$

If increase in instruction count (y%) is such that Execution $time_{old}$ = Execution $time_{new}$, then value of y will reflect the limit below which new reg-reg architecture will be more efficient than reg-mem architecture. A simple calculation illustrates this point. Assume that increase in the instruction count is 25% i.e. y= 25 %, then both new and old execution times will be equal. If the value of y is less than 25%, as usually in the case, then new architecture will be more efficient. In fact, this is really the case because in typical reg-reg architecture, load/store instruction are about 20 to 22%.

Question 3: Total Points (10)

A computer architect is designing a hardware datapath implementation and the architect has determined following circuit element delays.

Instruction Memory	150 ps
Decode	70 ps
Register Fetch	60 ps
ALU	150 ps
Data Memory	200 ps
Register Write Back	60 ps

(a) What is the length of a clock cycle for a single cycle datapath implementation? (2)

(b) What would be the frequency of a processor, corresponding to single datapath implementation? (3)

(c) What would be the length of fastest clock cycle for a 5-stage pipeline datapath? What would be the corresponding processor frequency? (3)

(d) How much faster is the 5-stage pipelined datapath compared to the single cycle datapath implementation? (2)

Solution:

(a) What is the length of a clock cycle for a single cycle datapath implementation?

The cycle-time has to allow an instruction to go through all the stages each cycle, so:

Clock Cycle Time = Instruction Memory + Decode + Register Fetch + ALU + Data Memory + Register Write Back

Clock Cycle Time = 150 + 70 + 60 + 150 + 200 + 60 = 690 ps
Length of Clock Cycle = 690 ps

(b) What would be the frequency of a processor, corresponding to single datapath implementation?

Processor Frequency is the reciprocal of Clock Cycle Length, so:

$$\text{Processor Frequency} = \frac{1}{clock\ cycle\ length} = \frac{1}{690\times10^{-12}} = \frac{1}{6.9\times10^{-10}} = 0.145\times10^{10} = 1.45GHz$$

(c) What would be the length of fastest clock cycle for a 5-stage pipeline datapath? What would be the corresponding processor frequency?

Pipelining to 5 stages reduces the cycle time to the length of the longest stage which in this case is data memory. So, the length of the clock cycle is 200 ps.

Processor Frequency is the reciprocal of Clock Cycle Length, so:

$$\text{Processor Frequency} = \frac{1}{clock\ cycle\ length} = \frac{1}{200\times10^{-12}} = \frac{1}{2\times10^{-10}} = 0.5\times10^{10} = 5GHz$$

(d) How much faster is the 5-stage pipelined datapath compared to the single cycle datapath implementation?

The performance ratio can be calculated as:

$$\frac{CPU\ Clock\ Cycle_{\text{single-cycle}}}{CPU\ Clock\ Cycle_{\text{pipelined}}}$$

= 690/200 = 3.45

So, the 5-stage pipelined datapath is 3.45 times faster than single cycle datapath implementation.

Question 4: Total Points (20)

For the following mathematical expressions write an assembly language code:

(a) For Reg-Reg architecture (10)

(b) For Reg-Mem architecture (10)

U = A + B + D
V = C + D
W= B << 3
X = 7B + B + C+ D
Y = X + V

Solution:

For simplicity, I am considering these expressions as independent.

(a) For Reg-Reg architecture

U = A + B + D

```
LW R1, 0(R0)        ;load A from memory to R1
LW R2, 4(R0)        ;load B from memory to R2
ADD R3, R1, R2
LW R4, 8(R0)        ;load D from memory to R4
ADD R5, R3, R4
SW R5, 12(R0)       ;store R5 to memory
```

V = C + D

```
LW R1, 0(R0)        ;load C from memory to R1
LW R2, 4(R0)        ;load D from memory to R2
ADD R3, R1, R2
SW R3, 8(R0)        ;store R3 to memory
```

W= B << 3

```
LW R1, 0(R0)        ;load B from memory to R1
```

```
SLL R1, R1, 3          ;shift logical left 3 bits
SW R1, 4(R0)           ;store R1 to memory
```

X = 7B + B + C + D

```
LW R1, 0(R0)           ;load B from memory to R1
MULTI R2, R1, 7        ;multiply immediate with 7
ADD R3, R2, R1
LW R4, 4(R0)           ;load C from memory to R4
ADD R5, R3, R4
LW R6, 8(R0)           ;load D from memory to R6
ADD R7, R5, R6
SW R7, 12(R0)          ;store R7 to memory
```

Y = X + V

```
LW R1, 0(R0)           ;load X from memory to R1
LW R2, 4(R0)           ;load Y from memory to R2
ADD R3, R1, R2
SW R3, 8(R0)           ;store R3 to memory
```

(b) For Reg-Mem architecture

U = A + B + D

```
LW R1, 0(R0)           ;load A from memory to R1
ADD R3, R1, 4(R2)
ADD R5, R3, 8(R0)
SW R5, 12(R0)          ;store R5 to memory
```

V = C + D

```
LW R1, 0(R0)           ;load C from memory to R1
ADD R3, R1, 4(R0)
SW R3, 8(R0)           ;store R3 to memory
```

W= B << 3

```
LW R1, 0(R0)           ;load B from memory to R1
SLL R1, R1, 3          ;shift logical left 3 bits
SW R1, 4(R0)           ;store R1 to memory
```

X = 7B + B + C + D

```
LW R1, 0(R0)           ;load B from memory to R1
MULTI R2, R1, 7        ;multiply immediate with 7
ADD R3, R2, R1
ADD R5, R3, 4(R0)
ADD R7, R5, 8(R0)
SW R7, 12(R0)          ;store R7 to memory
```

Y = X + V

```
LW R1, 0(R0)        ;load X from memory to R1
ADD R3, R1, 4(R0)
SW R3, 8(R0)        ;store R3 to memory
```

Question 5: Total Points (20)

Identify data hazards from the below code and show the execution of the code on a pipelined architecture on per cycle basis. You are required to highlight data hazard(s) and technique used to avoid it.

Opcode	Target	Source 1	Source 2
ADD	R1	R2	R3
SUB	R4	R1	R5
AND	R6	R1	R7
OR	R8	R1	R9
XOR	R10	R1	R11

Solution:

All the instructions after the ADD use the result of the ADD instruction. As shown in Figure 1.1, the ADD instruction writes the value of R1 in the WB pipe stage, but the SUB instruction reads the value during its ID stage, which is called data hazard. Unless precautions are taken to prevent it, the SUB instruction will read the wrong value and try to use it. In fact, the value used by the SUB instruction is not even deterministic: Though it is logical to assume that SUB would always use the value of R1 that was assigned by an instruction prior to ADD, this is not always the case. If an interrupt should occur between the ADD and SUB instructions, the WB stage of the ADD will complete, and the value of R1 at that point will be the result of the ADD. This unpredictable behavior is obviously unacceptable.

The AND instruction is also affected by this hazard. As shown in Figure 1.1, the write of R1 does not complete until the end of clock cycle 5. Thus, the AND instruction that reads the registers during clock cycle 4 will receive the wrong results.

The XOR instruction operates properly because its register read occurs in clock cycle 6, after the register write.

The OR instruction also operates without incurring a hazard because we perform the register file reads in the second half of the cycle and the writes in the first half.

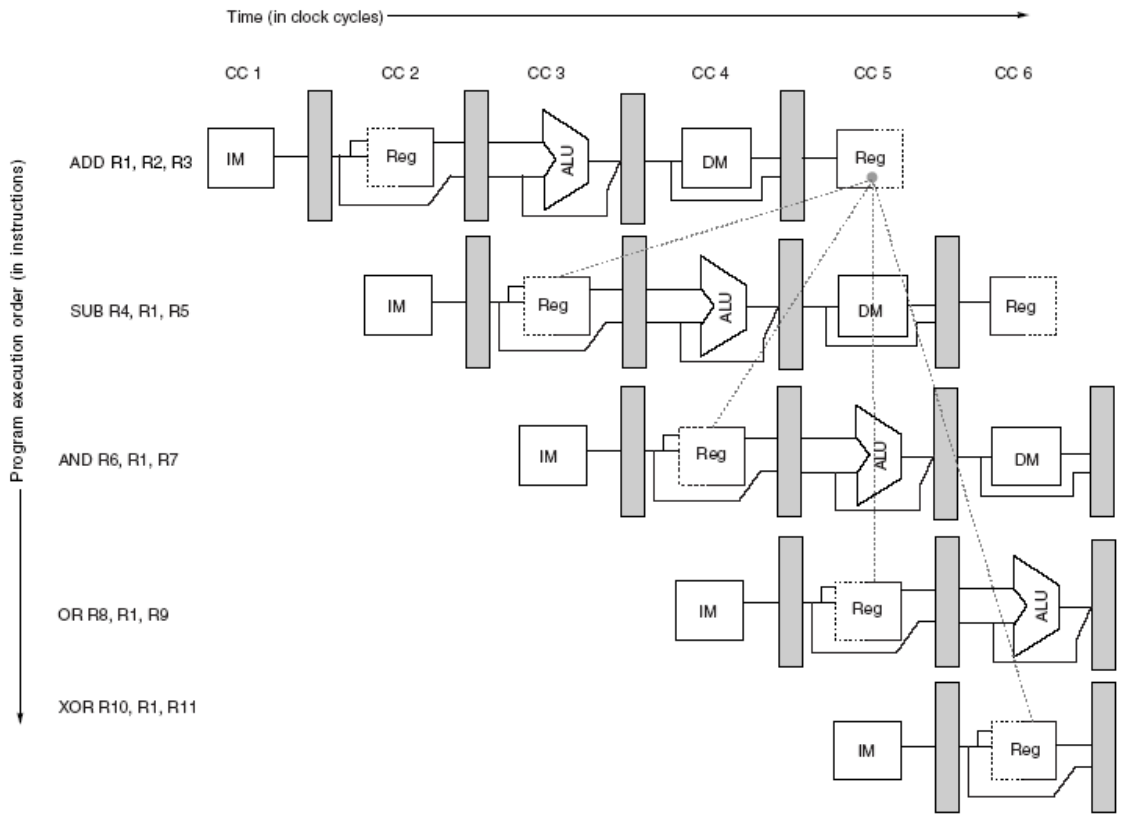


Figure 1.1 The use of the result of the DADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

We can solve this problem by using a simple hardware technique called forwarding (also called bypassing and sometimes short-circuiting) as shown in Figure 1.2 below. The key insight in forwarding is that the result is not really needed by the SUB until after the ADD actually produces it. If the result can be moved from the pipeline register where the ADD stores it to where the SUB needs it, then the need for
a stall can be avoided.
With forwarding, if the SUB is stalled, the ADD will be completed and the bypass will not be activated. This relationship is also true for the case of an interrupt between the two instructions.
We need to forward results not only from the immediately previous instruction, but possibly from an instruction that started 2 cycles earlier. Figure 1.2 shows the code sequence with the bypass paths in place and highlighting the timing of the register read and writes. This code sequence can be executed without stalls.

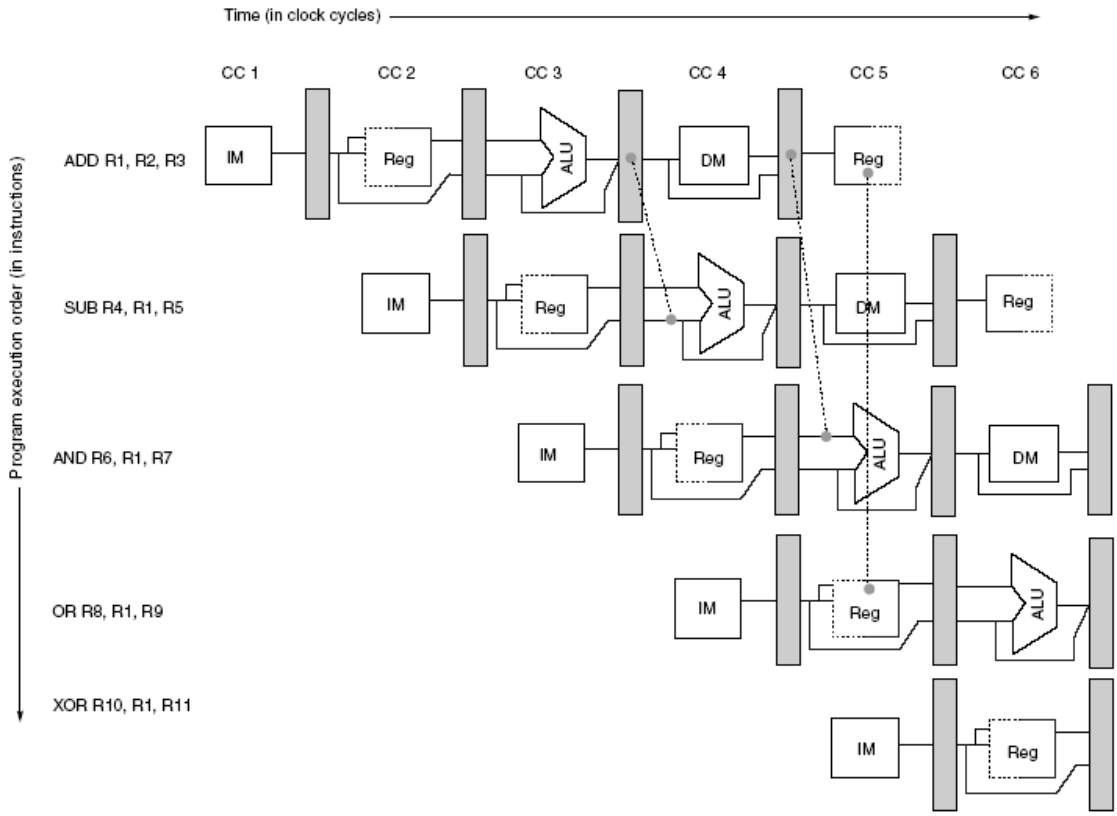


Figure 1.2 A set of instructions that depends on the ADD result uses forwarding paths to avoid the data hazard. The inputs for the SUB and AND instructions forward from the pipeline registers to the first ALU input. The OR receives its result by forwarding through the register file, which is easily accomplished by reading the registers in the second half of the cycle and writing in the first half, as the dashed lines on the registers indicate.

Question 6: Total Points (10)

Read the paper title "Reducing Data Hazards on Multi-Pipelined DSP Architecture with Loop Scheduling", and answer the following question.

Describe the loop scheduling algorithm and explain how loop scheduling algorithm is better than other algorithms?

Solution:

Research-paper based question.