Mine
Date : / /20
Page :

Lecture - 40

Parallel Processing

Simultaneous Data Processing

Types
- Vector Processing } Not in GATE
- Array Processing
- Pipelining Processing

Flynn's Classification

⇒ SISD :
Single instruction stream, single data stream

Fetch single instruction at a time.
Run only one instruction at a time.
Eg : Von Neumann

⇒ SIMD :
Single Instruction stream, multiple data stream

Fetch single instruction at a time.
Execute multiple at a time
Eg, pipeline processor.

Mine
Date : / /20
Page :

⇉ MISD
Multiple instruction stream, Single data stream

Fetch multiple instructions at a time.
Execute single instruction at a time.

Only hypothetical.

⇉ MIMD
Multiple instruction stream, Multiple data stream

Fetch multiple instructions.
Execute " " .

Multiple pipelines

Eg, Super scalar computers.

These computers have instruction level parallelism.

Pipelining

Useful when the same processing is applied over multiple inputs.

We divide the processing into sub operations.

⇉ We decompose a sequential process into sub-operations.

⇒ Sub operations are performed in segments.

i) We decompose a sequential process into sub operations.

ii) Sub operations are performed in segments (seperate hardware).

→ Each segment can perform its respective sub operation over different input in parallel to one another.

Task ⇒ One operation performed in all segments forms a complete task.

Think of this like a car assembly. Each worker represents a segment or stage. Each worker is doing his specialized work which represents doing that suboperation.

A single car represents a task and the complete assembly line represents the pipeline.

Mine
Date : / /20
Page :

Eg of pipelining,

Suppose we want to do

$A_i * B_i + C_i$ for $i = 1, 2, 3 \ldots 5$

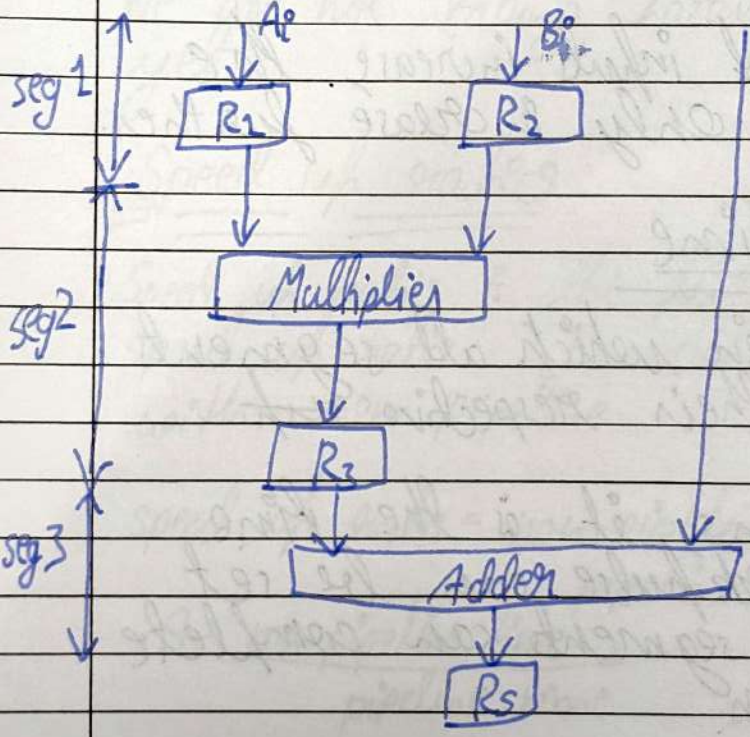
Let's ~~divide~~ decompose into subfunction as,

Segment 1: $R_1 \leftarrow A_i$, $R_2 \leftarrow B_i$

Segment 2: $R_3 \leftarrow R_1 * R_2$, $R_4 \leftarrow C_i$

Segment 3: $R_5 \leftarrow R_3 + R_4$

Rough hardware design →



Clock Pulse	Segment 1		Segment 2		Segment 3
	R_1	R_2	R_3	R_4	R_5
1	A_1	B_1			
2	A_2	B_2	$A_1 * B_1$	C_1	
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6			$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7					$A_5 * B_5 + C_5$
8					

The same process in sequential manner will take 15 cycles.

As the number of inputs increase, the efficiency will only increase further.

Pipeline Cycle Time

Minimum time in which all segments can perform their respective ~~out~~ suboperations.
In simple terms, it is the time to which clock pulse can be set so that all segments can complete their operation.

Now let's look at the above table and make general considerations for pipelines.

Mine
Date : / /20
Page :

Consider k segment pipeline
clock cycle time = t_p
Number of inputs = n

Time for first input = $k \times t_p$
Or if we want in terms of cycle number, we are numbering as cycle no = 1 complete cycle.

Cycles for first input = k
Cycles for all n inputs = $k + (n-1)$

Time for processing all n inputs = $(k + n - 1) * t_p$

We are not taking hazards into consideration yet.

Speed up ratio

Speed up ratio is the performance gain.
Speed up ratio > 1 when comparing with non-pipeline

$$\text{speed up ratio} = \frac{\text{non-pipeline time}}{\text{pipeline time}}$$

$$S = \frac{\text{non-pipeline time}}{\text{pipeline time}}$$

If we consider that both have same clock pulse time,

$$S = \frac{\text{non-pipeline cycles}}{\text{pipeline cycles}}$$

Mine
Date : / /20
Page :

Therefore,

$$S = \frac{k \times n}{k + (n-1)}$$

[k = no. of suboperations
n = no. of inputs]

Suppose $n \gg k$ [We can ignore first k-1 cycles in this case]

This is ideal case for pipeline system,

~~$S = \frac{k \times}{}$~~

$$S_{ideal} = S_{max} = \frac{k \times n}{n} = k$$

So maximum/ideal speed up is k times.

If the question does not tell cycles and wants the maximum speed up use,

$$S_{max} = \frac{\text{Time for one task non-pipeline}}{\text{Time of single clock pulse}}$$

We use time of single clock pulse in denominator since k-1 cycles are ignored. The above formulae is better for numericals.

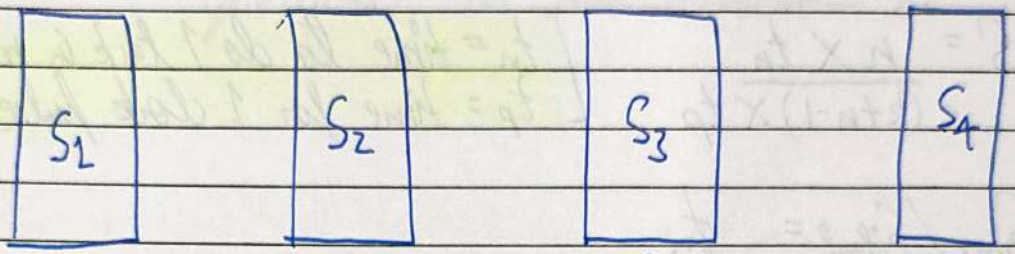
Hdr ; ignore first k-1 cycles for ~~non~~ pipeline if asking for max/ideal speed up.

Mine
Date : / /20
Page :

Lecture-41

Synchronous Pipeline

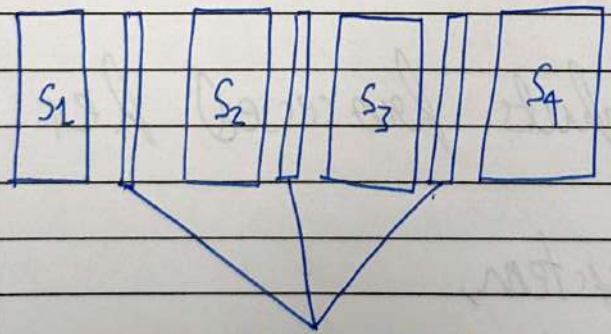
Suppose all segments ~~#~~ take different amount of time to do ~~on~~ their operation.



So to keep it synchronous, we choose the max segment delay as clock pulse.

t_p = max (segment delay)

But ~~the~~ there can be loss of intermediate values if some segments are faster.
So we have intermediate register/buffer.



But since they will also cause some delay

t_p = max (segment delay) + register delay.

Therefore in pipelines there is a delay caused by registers. This is not present in non-pipeline system.

So we modify the formula for speed up ratio as,

$$S = \frac{n \times t_n}{(k+n-1) \times t_p}$$

[t_n = time to do 1 task in non-pipeline
t_p = time for 1 clock pulse]

$$S_{max} / S_{ideal} = \frac{t_n}{t_p}$$

Latency

Time after which machine takes next input.

Latency for non-pipelined system = t_n
Latency for piplined system = t_p

Throughput

Number of inputs processed per unit time.

In pipelined system,

Time taken for n inputs = $(k+n-1) \times t_p$
Inputs processed per unit time (Throughput) = $\frac{n}{(k+n-1) \times t_p}$

Mine
Date : / /20
Page :

For ideal case, ignore k-1

$$\text{Throughput}_{ideal} = \frac{1}{t_p}$$

Note : This lecture also has many questions in ~~the lecture~~. It.

Mine
Date : / /20
Page :

Lecture - 42

Instruction Pipeline

We will apply pipeline processing on instruction cycle, it is called an instruction pipeline.

Assume such pipeline has 5 phases,

IF : Instruction Fetch
ID : Instruction Decode and Address calculation
OF : Operand Fetch
EX : Execution
WB : Write Back

Eg

Instructions	Clock Pulses 1	2	3	4	5	6	7	8
I_1	IF	ID	OF	EX	WB			
I_2		IF	ID	OF	EX	WB		
I_3			IF	ID	OF	EX	WB	
I_4				IF	ID	OF	EX	WB

This works well when we don't have branching instruction. We follow the certain rule if there is a branching instruction.

	1	2	3	4	5	6	7	8	9	10	11	12
I_1	IF	ID	OF	Ex	WB							
I_2		IF	ID	OF	Ex	WB						
I_3			IF	ID	OF	Ex	WB					
I_4				IF	-	-	IF	ID	OF	Ex	WB	
I_5								IF	ID	OF	Ex	WB

In cycle 4, I_3 is decoded as a branch type instruction. So all further Instructions are removed from the pipeline. So the I_4 fetch is discarded.

No further instructions enter pipeline until branch decision is taken.

By standard, branch decision is made in EXECUTION Phase.

So in our example at the end of cycle 6. So at start of 7th cycle, Instructions can be entered in pipeline.

Let's consider branch is not taken, the I_4 is fetched again because fetch was discarded.

Then pipeline continues it's usual working.

Cycles 4, 5 and 6 above had the pipeline stalled, so they are called stalled cycles (also called pipeline bubbles).

Mine
Date : / /20
Page :

Example when jump is taken.

	1	2	3	4	5	6	7	8	9	10	11	12
I_1	IF	ID	OF	EX	WB							
I_2		IF	ID	OF	EX	WB						
I_3			IF	ID	OF	EX	WB					
I_4				IF	-	-						
I_5												
I_6												
I_7							IF	ID	OF	EX	WB	
I_8								IF	ID	OF	EX	WB

Here I_4 is the instruction ~~where~~ which is branch instruction.
So when it is decoded at ~~the~~ (cycle 4), the further pipeline is stalled till cycle 6 completes.

And since branch is taken we jump to I_7 at start of cycle 7.

Number of instructions processed = 5(n)
Number of segments (k) = 5

Total cycles required = n+k-1
= 5+5-1
= 9

But in our example, we took 12 cycles because 3 were stalled.

Mine
Date: / /20
Page:

To calculate number of stalled cycles.

No. of stalled cycles = i - 1

where, i = the cycle at the end of which branch decision is taken.

In our example we have 5 segments

IF ID OF EX WB

We taken decision at end of 4th segment. So
stalled cycles = 4 - 1

i = ith segment at the end of which branch decision is taken.

Pipeline Hazards

Situations that prevent next instruction from being executed during its designated clock cycle.

They cause stalled cycles.

Pipeline Hazards

- Structural Hazards / Resource Conflict
- Data Hazard / Data Dependency
- Control Hazard / Branch Difficulty

Mine
Date : / /20
Page :

Structural Hazard

2 different segments try to use same resource at same time.

Suppose we have two instructions I_1 and I_2, but I_1 takes 3 cycles for execution.

So in pipeline,

I_1	IF	ID	OF	EX	EX	EX	WB	
I_2		IF	ID	OF	-	-	EX	WB

stalled cycles.

Since EX segment is already in use, this causes resource conflict.

Another conflict of this type is if we have operand fetch and instruction fetch in the same cycle in a pipeline.

Since only a single access can be done, it causes resource conflict. To solve this we can use two seperate caches, a data cache and an instruction cache.

Mine
Date : / /20
Page :

Lecture - 43

Data Dependency.

Result of an instruction is used as input in next.

If I_n : $R_1 \leftarrow R_2 \times R_3$
I_{n+1} : $R_4 \leftarrow R_1 + R_5$

So in pipeline

I_n	IF	ID	OF	EX	WB				
I_{n+1}		IF	ID	-	-	OF	EX	WB	

stall cycles.

A general pipeline hardware cannot detect data dependency.

Solution for data dependency

- Software (By compiler)
- Hardware

Software solution is called delayed load and is used when general pipeline is used.

Compiler generates code in a way that avoids data dependency.

Mine
Date : / /20
Page :

Delayed load

We either

- Move independent instructions between dependent instructions
- Insert no operation instruction.

I_1
I_2 } Data Dependency
I_3

I_4 } Independent instructions
I_5

↓ compiler converts to.

I_1
I_2 ← Data Dependency
I_4
I_5
I_3

or compiler adds no operation instructions.

I_1
I_2 ← Converted to ⟹
I_3
I_4

I_1
I_2 ← Data Dependency
No operation instruction
" " "
I_3
I_4

Mine
Date : / /20
Page :

Hardware solutions

↳ Hardware interlock
↳ Operand forwarding

Hardware interlock

This method will make stall cycles in pipeline to solve this problem.

This is what we saw in the very first example.

I_n : $R_1 \leftarrow R_2 \times R_3$
I_{n+1} : $R_4 \leftarrow R_1 + R_5$

IF ID OF EX WB
IF ID - - OF EX WB

Hardware locked to create stall cycles.

Operand Forwarding

Rather than stalling cycles, we take operand value and ALU data from the intermediate registers within the pipeline rather than waiting for write back (WB).

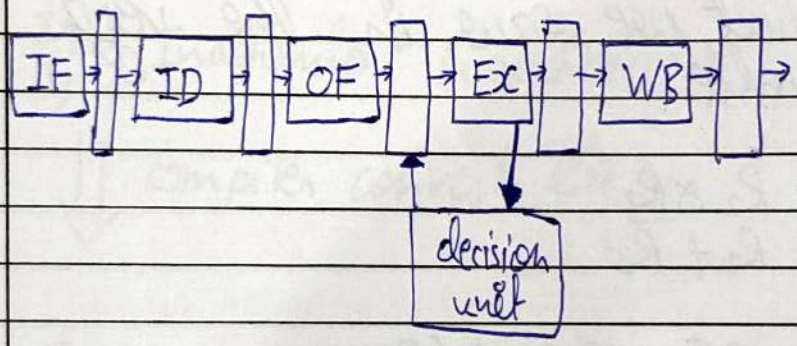
So if we have data dependency as

$I_n : R_1 \leftarrow R_2 \times R_3$
$I_{n+1} : R_4 \leftarrow R_1 + R_5$

IF ID OF EX WB

IF ID OF EX WB

In hardware implementation,



The decision unit detects data dependency. This is also called bypassing.

Because a specialized decision unit is need this method is expensive.

Operand forwarding can't make it so that ALL ALL data dependency causes no stall cycles.

This cannot prevent stall cycles in memory data dependency i.e load and store operation data dependency.

Mine
Date : / /20
Page :

Load Data Dependency

$R_1 \leftarrow M[add]$
$R_4 \leftarrow R_1 + R_2$

By standard many CPUs do LOAD operation in EX phase. So we consider a single stall cycle.

IF	ID	OF	EX	WB	
	IF	ID	-	OF	EX	WB

~~Read~~ Stop Data Dependency

$R_4 \leftarrow R_1 \times R_5$
$M[add] \leftarrow R_4$

There are always 2 stalled cycles.

IF	ID	OF	EX		WB	
	IF	ID	-	-	OF	EX	WB

Control Hazard or Branch Difficulty

Hazards because of branch instruction. General pipeline hardware cannot detect branch difficulty.

Solution
- Software solution (Delayed Branch)
- Hardware solution

Date : / /20
Page :

Software Solution (Delayed Branch)

The compiler will add no operation instructions equal to the number of cycles to stall.
Similar to ~~software~~ delayed load.

Hardware solutions

1. **Prefetch Target Instruction**

Two pipelines will work together and the wrong pipeline is discarded.

2. **Branch Prediction**

The pipeline will ~~does~~ try to predict the branch decision and if wrong decision taken, it will rollback.

Types

- Static (Either always take branch or not take branch)
- Dynamic (Will make guess according to instruction)

Mine
Date : / /20
Page :

3. Loop Buffer

In pipeline we have a buffer which will store instructions of a loop and continuosly run them without fetching or decoding over and over.

4. Branch Target Buffer

We use this frequently with branch prediction.
If a certain branch instruction previously taken decision, we will take same decision next time.
The previous decisions result are stored in a buffer.

Mine
Date : / /20
Page :

Lecture - 44

Data Hazard Classification

Assume there are two instructions i and j and i is executed before j.

Read After Write (RAW)

j tries to read a source before i writes it. So j incorrectly gets the old value.

i: $R_1 \leftarrow R_2 + R_3$ } called data dependency
j: $R_4 \leftarrow R_1 * R_5$ }

Solution is operand forwarding.

Write After Write (WAW)

i: $R_1 \leftarrow R_2 + R_3$ } called write dependency
j: $R_1 \leftarrow R_4 \times R_5$ } (it is a false dependency)

j tries to write an operand before it is written by i.

Write After Read (WAR)

j tries to write a destination before i reads it. So i gets too new value

i: $R_1 \leftarrow R_2 + R_3$ } anti-dependency
j: $R_2 \leftarrow R_4 \times R_5$ } (it is a false dependency)

Mine
Date : / /20
Page :

Solution for false dependencies (Register Renaming)

For j instruction, replace the register causing false dependency with some other register and do the same for all subsequent instructions that depend on the j instruction's operation on that register.

This is a hardware solution.

i: $R_1 \leftarrow R_2 + R_3$ —reg. renaming→ i: $R_1 \leftarrow R_2 + R_3$
j: $R_1 \leftarrow R_4 \times R_5$ j: $R_6 \leftarrow R_4 \times R_5$

Pipeline Efficiency

Because of hazards (stall cycles), efficiency is reduced.

Speedup without hazard = S
Speedup with hazard = S'

$\text{Efficiency} = \frac{S'}{S}$

$0 \leq \text{Efficiency} \leq 1$

Mine
Date : / /20
Page :

CPI (Cycles Per Instruction)

CPI = $\frac{\text{Number of cycles for instructions}}{\text{Number of instructions.}}$

No. of instructions = n
No. of segments = k

CPI = $\frac{n + k - 1 + (\text{stalled cycles})}{n}$

CPI_{ideal} (ignore k-1) = $\frac{n + \text{stalled cycles}}{n}$

$CPI_{ideal} = 1 + \frac{\text{stalled cycles}}{n}$

To calculated stalled cycles,

stalled cycles = stalling instructions x number of cycles it stalls.

$CPI_{ideal} = 1 + \frac{\text{stalling instructions} \times \text{no. of cycle it stalls}}{n}$

$CPI_{ideal} = 1 + (\text{ratio of stalling instructions}) \times \text{no. of cycle it stall}$

So if ques says, 25% of instructions cause 2 stalled cycles then,

rating of stalling instructions = 0.25
no. of cycles it stalls = 2
$CPI_{ideal} = 1 + 0.25 \times 2$

Note : Questions for pipeline in lecture

Mine
Date : / /20
Page :

Lecture - 45

Non-Linear Pipeline (Not frequently in Gate)

Input is not passed in sequential manner and a single segment can be used multiple time, to represent this we use a reservation table.

Segments \ Clock →	1	2	3	4	5	6	7	8
S_1	X					X		X
S_2		X			X			
S_3			X			X		

Latency

Number of time units (cycles) after which next input can be given to pipeline after a input enters pipeline.

~~i.e. no of cycles between two input.~~

→ permissible latency : latency which does not cause collision.

→ forbidden latency : latency which cause collision

~~Collision~~ : when a segment in pipeline is tried to be used for 2 inputs at same time. This is a structural hazard.

Mine
Date : / /20
Page :

Forbidden latency calculation

For example,

	1	2	3	4	5	6	7	8
S_1	X					X		X
S_2		X		X				
S_3			X		X		X	

All latencies ⇒ 1, 2, 3 8

For S_1,
since it is used in cycle 8 and then 6
Forbidden latency = 8 - 6 = 2

Similarly, it is used in 8 and 1
Forbidden latency = 8 - 1 = 7

Similarly, it is used in 6 and 1
Forbidden latency = 6 - 1 = 5

Forbidden latencies = 2, 5, 7

For S_2,
Similarly calculate as in S_1
Forbidden latencies = 2

For S_3,
Forbidden latencies = 4, 2, 2

So all forbidden frequencies are,
2, 4, 5, 7

Remaining are permissible latencies
1, 3, 6, 8

Collison Vector

It is a n-bit vector (number) where n is the number of clocks to process 1 input.

$C_n C_{n-1} \cdots C_2 C_1$

If n^{th} cycle number is Forbidden then,
$C_n = 1$
If n^{th} cycle number is permissible then,
$C_n = 0$

From our example in previous section

C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1
0	1	0	1	1	0	1	0
P	F	P	F	F	P	F	P

(P = Permissible; F = Forbidden)

So collision vector for our example,
01011010

Mine
Date : / /20
Page :

State diagram using collision vector

⇒ Take a state S
⇒ For every 0 (zero) at position i, shift ~~collision vector for state~~ S in right by i and the take bitwise OR for new state with collision vector.
⇒ Draw arrow from S to new state and write i on the arrow.
⇒ Do same process for new states acquired.
Note: Shift state S and dig OR with orignal collision vector

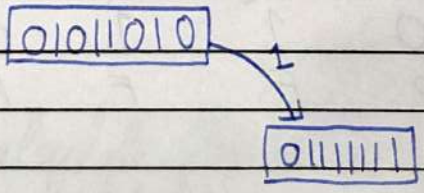
* Continuing old example

For state 01011010,

★ 0 at i = 1

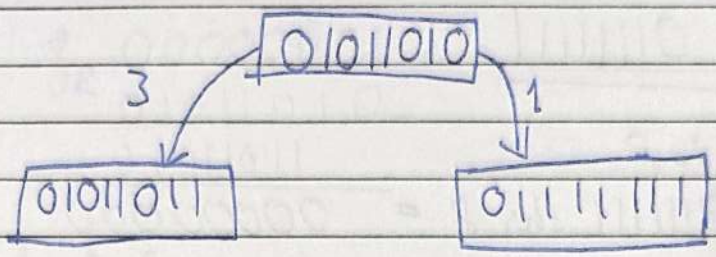
⇒ 01011010 shr 1 = 00101101

⇒ OR 01011010
00101101
01111111



★ 0 at i = 3

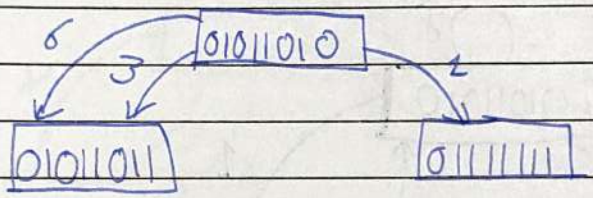
⇒ 01011010 shr 3 = 00001011
⇒ ~~01011010~~ OR 01011010
00001011
01011011



★ 0 at i = 6

⇒ 01011010 shr 6 = 00000001

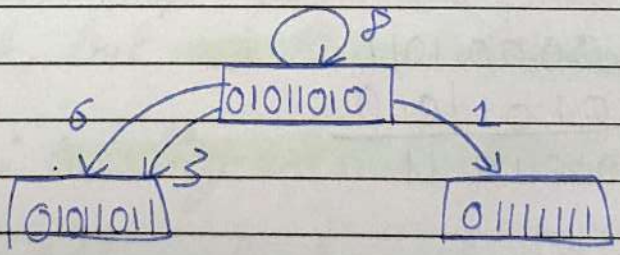
⇒ OR 01011010
00000001
01011011



★ 0 at i = 8

⇒ 01011010 shr 8 = 00000000

⇒ OR 01011010
00000000
01011010



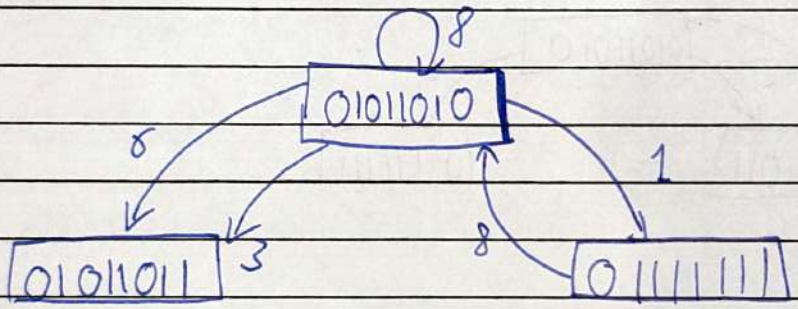
Mine
Date : / /20
Page :

For state 01111111

★ 0 at $i=8$

⇒ 01111111 shr 8 = 00000000

⇒ OR 01011010
00000000
01011010



For state 01011011

★ 0 at $i=3$

⇒ 01011011 shr 3 = 00001011

⇒ OR 00001011
01011010
01011011

★ 0 at $i=6$

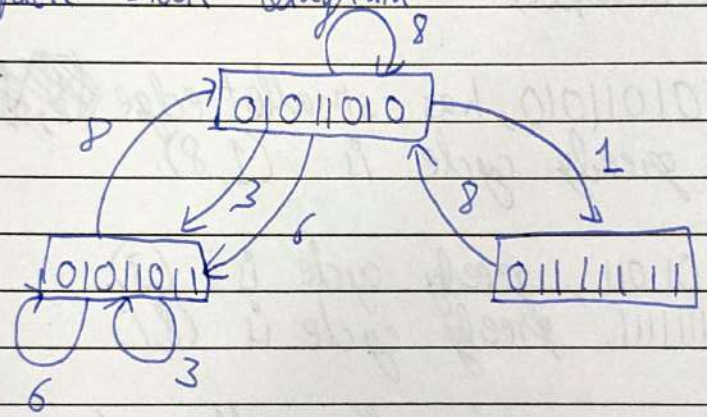
⇒ 01011011 shr 6 = 00000001

$\Rightarrow$ 00000001
OR 01011010
01011011

* 0 at i=8
01011011 shr 8 = 00000000

00000000
OR 01011010
01011010

Complete state diagram



Simple Cycles

A cycle in the state diagram is a complete loop from one state back to itself, but no other state is repeated

Note: including all states is not required.

Eg, simple cycles from state 01011010 $\Rightarrow$ (8), (1,8), (6,8) etc.
cycles from 01011011 = (3), (6), (8,3), (8,6)
" " 01111111 = (8,1)

Mine
Date : / /20
Page :

Greedy Cycles

~~Each state will have~~
A simple cycle whose first edge taken is smallest for the starting state is known as a greedy cycle.

Each state will have a greedy cycle if it has atleast a single simple cycle. The greedy cycle will also have smallest average cycles of all simple cycles.
Eg. In our example,

for state 01011010, has smallest edge 1
Therefore, greedy cycle is (1,8).

for state 01011011, greedy cycle is (3).
for state 01111111, greedy cycle is (8,1).

Hence for a given state, the simple cycle with lowest average cycle is the greedy cycle.

Minimum Average Latency

The average latency of a cycle can be calculated by taking average of all the edges.

Minimum Average Latency (MAL) is the smallest average latency of all greedy cycle.

Mine
Date : / /20
Page :

So in our example,

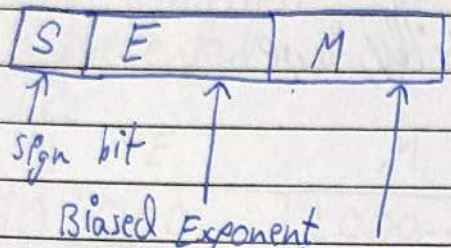
Greedy Cycle	Average latency
(1,8)	$\frac{8+1}{2} = 4.5$
(3)	$\frac{3}{1} = 3$
(8,1)	$\frac{8+1}{2} = 4.5$

MAL = 3

Mine
Date: / /20
Page:

Lecture 46

Floating Point Representation



Biased Exponent

Suppose we have an exponent field of 4 bits.

We know that when using 2's complement for negetive numbers, the range for n bit negetive number is from -2^{n-1} to $(2^{n-1}-1)$ in decimal.

So 4 bits can store from -2^3 to 2^3-1 i.e. -8 to 7 which is from $(1000)_2$ to $(0111)_2$

We do not want to use a negetive number for storage. So to transform it we can just add a bias to make number start from 0 in storage.

In our 4 bit example bias is 8
In n bit exponent, bias is 2^{n-1}
And exponent + bias = E (biased exponent)

Mine
Date : / /20
Page :

Normalized Mantissa

i) Explicit Normalization (default for GATE ques)

When we normalize explicitly, we convert to form 0.10....

So
$101.11 \xrightarrow{\text{converts to}} 0.10111 \times 2^3$

And mantissa (M) = 10111
and biased exponent = $3 + 2^{e-1}$ (e is no. of exponent bits)

ii) Implicit Normalization (used for IEEE)

When we do implicit,
$101.11 \xrightarrow{\text{converts to}} 1.0111 \times 2^{*2}$

Mantissa (M) = 0111

Note: Sometimes bias = n is written as: excess - n form of exponent

So if exponent is excess-n, the bias is n.

For floats it is a tradeoff of ~~exponent~~ range and accuracy.

More bits in mantissa (M) = more precision
More bits in exponent (E) = larger range of numbers

Mine
Date : / /20
Page :

Disadvantages of Floating Point Representation

~~The largest~~

⇒ The smallest +ve number that can be represented in a explicit normalized mantissa is

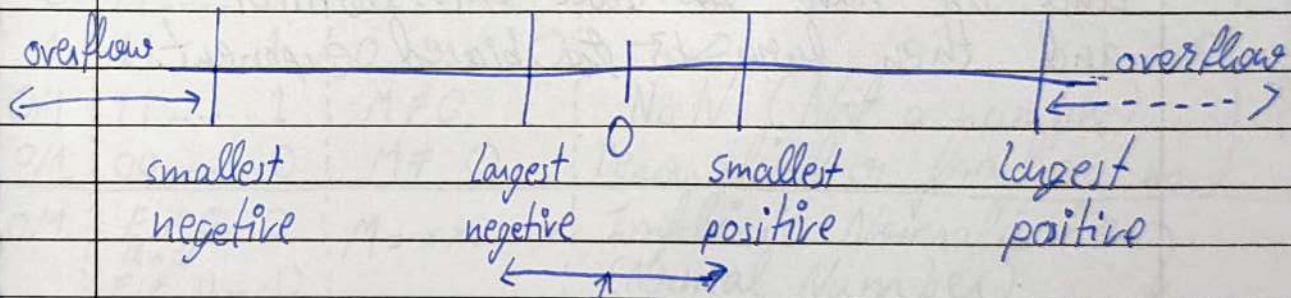
S	E	M
0	000...	1000...

This is because we always have a 1 after the point in explicit normalized form.

⇒ The largest -ve number is similarly

1	00...	100...
S	E	M

So on the number line



This range of numbers cannot be stored in conventional floating point representation (This is called underflow)

Mine
Date : / /20
Page :

So conventional floating point cannot :

- store zero
- store infinity
- store a number which cannot be normalized

To fix this issue, we use standard IEEE-754 format.

In some questions, mantissa is said to not be normalized and said to be stored in pure fraction.

In these questions, don't do normalization

Eg, if 0.239×2^{13} will be directly converted to 0.0011101×2^{13} and we have to take $M = 0011101$ and then burn 13 to biased exponent.

Mine
Date : / /20
Page :

Lecture - 47

IEEE-754 Floating Point Representation

- Single Precision
  (S = 1 bit, E = 8 bits, M = 23 bits) (bias = 127)
- Double Precision
  (S = 1 bit, E = 11 bits, M = 52 bits) (bias = 1023)

We don't use the same bias calculating method as we do in conventional method because
E = 00....0 and E = 11...1 are used for special numbers. [if (e) exponent bits, bias is $2^{e-1} - 1$]
Note ⇒ We do implicit normalization in IEEE-754

S	E	M	Number	
0	00...0	0...0	+0	special numbers
1	00...0	0...0	−0	
0	11...1	0...0	+∞	
1	11...1	0...0	−∞	
0/1	11...1	M ≠ 0	NaN (Not a number)	
0/1	00...0	M ≠ 0	Denormalized or fraction no.	
0/1	E ≠ 0..0 and E ≠ 11...0	M = xx...x	Implicit Normalization (Normal Number)	

(Implicit Normalization → means normalize to 1.0111... form)

Not A Number

Since we ~~normalized~~ chose a bias such that E = 1...1 and 00...0 is reserved, if we try to store a number such that E = 11...1 and M ≠ 0, it gives NaN error.

Mine
Date : / /20
Page :

Denormalized Numbers

Very small number, which cannot be normalized.

In this case, in order to store this in IEEE standard,
we will only normalize till exponent is -(bias-1) and storing the Mantissa we get and E = 00...0.

Eg, suppose we are using single precision float and we have a number 0.0.....0011
and we cannot completely normalize, then we treat it as special numbers,

$0.0\ldots0011 \longrightarrow 0.000011 \times 2^{-126}$ ← bias-1

And store M = 000011
E = 00...0

So for a denormalize number,

$|Value| = 0.M \times 2^{-(bias-1)}$

Note : A question came in GATE 2020 on single precision number division.
The question asked for answer of
0x42200000 ÷ 0xC1200000