12 2016 MARCH

SATURDAY DAYS 072-294 WK 11

MARCH

Wk	M	T	W	T	F	S	S
10		1	2	3	4	5	6
11	7	8	9	10	11	12	13
12	14	15	16	17	18	19	20
13	21	22	23	24	25	26	27
14	28	29	30	31			

Week - 15

Computer Organisation And Architecture : Instruction pipelining

Theory Of computation : Regular and CFLs, Pumping Lemma

Computer Architecture

Instruction Pipelining

⇒ Pipeline : Overlapping the execution of successive instructions to achieve high performance. Here new input enters into the pipeline before completion of an old input so that new input instruction is executed along with old one.

⇒ 3 stages of pipeline : a) Filling : So new instructions acquire the functional units which are utilised by older instructions But still some of the functional units (stages) are unused.

b) Full : This corresponds to the case when maximum no. of instruc are present at a time. (Or) we can say all pipeline stages are being used.

13 SUNDAY

c) Empty : The subsequent pipeline cycles are empty stage cycles, as the pipeline is being emptied.

⇒ The control signals are propogated through the pipeline till they reach the appropriate stage.

⇒ The pipelined organisation uses interface registers between

9 The stages of pipeline.

10 ⇒ Hazard: Any condition that leads to the pipeline to stall is called a hazard.

11

⇒ Dependency: Dependency means simply operation (read or write) by an instruction on a data item which has been operated by an earlier instruction.

1

⇒ Dependency does not necessarily mean hazard but hazard definitely implies being a dependency.

⇒ Types of hazards:

4 a) Data Dependency: Dependency created to data between the different instructions. Also known as RAW hazard.

5

⇒ RAW hazard causes stalls in pipeline.

6 ⇒ Solutions:

a) Dividing WB stage into 2 parts: first part for writing followed by reading. But this also prevents only one stall.

7

b) Operand forwarding: Forwarding the data through interface registers generally used between execute phase and instruction decode / operand fetch phase as applicable.

c) Compiler optimisations can also be done like insertion of Notes NOP instructions.

b) WAR and WAW hazard: ⇒ WAR also known as anti-dependency.

⇒ WAW also known as output dependency.

15 2016 MARCH

TUESDAY DAYS 075-291 WK 12

⇒ Bernstein's conditions for finding dependencies :

a) $R(S_i) \cap W(S_j)$: WAR dependency
b) $W(S_i) \cap W(S_j)$: WAW dependency
c) $W(S_i) \cap R(S_j)$: RAW dependency

} $\neq \phi$

⇒ Branch Delays: ⇒ To support pipelining which involves branch instruction, we need stalls so as to maintain the functionality of program.

⇒ For that, branch penalty = Stage at which target address is available - 1

⇒ Total no of stall due to branch = Branch frequency * Branch Penalty [No of stales for each branch]

⇒ Methods to minimise branch delays:

a) Delayed branching: Insertion of NOP or rearrangement of instructions. But its effectiveness depends on how often a compiler can reorder instructions usefully to fill the delay slot.

⇒ In this rearrangement process only, there is a possibility of WAR and WAW hazards as instructions may become out of order.

⇒ To remove such hazards, we use the technique known as register renaming.

Notes

b) Branch Prediction Technique: This is better approach than

delayed branching to minimise stalls due to branching.

⇒ The hazard associated with branch instruction is also known as control hazard.

⇒ Structural hazard: This occurs due to conflict in use of functional units. e.g. conflict in instruction fetch and memory access phase of RISC pipeline.

⇒ Performance Evaluation of Processor and Pipelined System:

⇒ Execution time of a program = $(\sum_i IC_i * CPI_i) *$ cycle time

where IC_i: dynamic ins^n count, CPI_i: cycles / ins^n (i^{th} one)

⇒ Throughput = $\frac{R}{S}$ for non pipelined system where

R: Clock rate S: CPI for instruction (avg CPI)

⇒ avg. CPI = $\frac{\sum_i IC_i * CPI_i}{\sum_i IC_i}$

⇒ For pipelined system, ideal throughput = R where R is clock rate as CPI = 1 for ideal case having no stalls.

Notes: given k stages, n instructions,

Execution time = $(k+n-1) *$ cycle time.

17 2016 MARCH

THURSDAY DAYS 077-289 WK 12

⇒ The cycle time of a pipeline is chosen keeping in mind the max stage delay out of the given stage delays along with interface delays if any. Thus.

t_p (Cycle time) = max(stage delay + interface delay)

⇒ For single cycle based CPU, instruction is performed in 1 cycle only and hence no concept of CPI etc. CPI will be considered as 1 even for non pipelined system.

⇒ Effect of stall on CPI:

a) $\delta_{branch\ penalty} = p \times$ [illegible] $\times$ [illegible] ; p: frequency of branch instruction, [illegible]: branch penalty

∴ Effective CPI = $1 + \delta_{branch\ penalty}$

b) $\delta_{miss} = (m_i + d \times m_d) \times p_m$ where

m_i: fraction of instructions incurred a cache miss during instruction fetch

m_d: fraction of instructions incurred a cache miss during data fetch

d: fraction of Load/Store instructions

p_m: Penalty to access main memory.

⇒ a) and b) cases may be combined together.

⇒ Pipelining does not improve the execution time of a

single instruction. However it increases the throughput of the entire system.

⇒ Let T_1 is execution time of 1 instruction in pipeline and T_2 in non pipeline system, then $T_1 \geq T_2$

⇒ Speedup: Performance gain by pipeline over non pipeline. given by,

a) $$\text{Speedup} = \frac{CPI_{\text{non pipeline}} * \text{Cycle time}_{\text{non pipeline}}}{CPI_{\text{pipeline}} * \text{Cycle time}_{\text{pipeline}}}$$

[General Formula]

b) Say we have uniform pipeline, then

$$\text{speedup} = \frac{CPI \text{ non pipeline}}{1 + \text{no. of stalls in pipeline / ins}^n}$$

c) If no stall, then speedup = $\frac{t_n}{t_p}$; t_n: execution time of on ins^n in non pipeline system and t_p: clock cycle time (max) in pipelined system.

⇒ In this, we do not add interface delays of pipeline in t_n as interface registers are meant for pipeline only.

⇒ Amdahl's law: Achieving optimal performance under constraints

$$\text{Speedup} = \frac{1}{1 - (P_1 + P_2) + \frac{P_2}{S_1} + \frac{P_2}{S_2}}$$

where β_1 : fraction of enhancement using speedup S_1

β_2 : fraction of enhancement using speedup S_2

$1 - \beta_1 - \beta_2$: sequential part.

⇒ Efficiency of Pipeline : defined mathematically as:

$$\eta = \frac{\text{Speedup}}{\text{Pipeline Depth (No of stages in pipeline)}}$$

⇒ B. maximum achievable speedup = no of stages

maximum throughput = clock rate of processor