Instruction Pipeline Design - Instruction Execution Phases

A G Sreejith

November 23, 2018

Instruction Execution Phases

1. Instruction Fetch
2. Decode instruction
3. Operand Fetch
4. Execute
5. write-back

Instruction Fetch

Instruction Fetch

fetches instructions from a cache memory, presumably one per cycle

Decode instruction

Decode instruction

reveals the instruction function to be performed and identifies the resources needed. Resources include general-purpose registers, buses, and functional units

Operand Fetch

Operand Fetch

Fetch operands from memory if necessary: If any operands are memory addresses, initiate memory read cycles to read them into CPU registers.

Execute

Execute

Perform the function of the instruction. If arithmetic or logic instruction, utilize the ALU circuits to carry out the operation on data in registers.

write-back

write-back

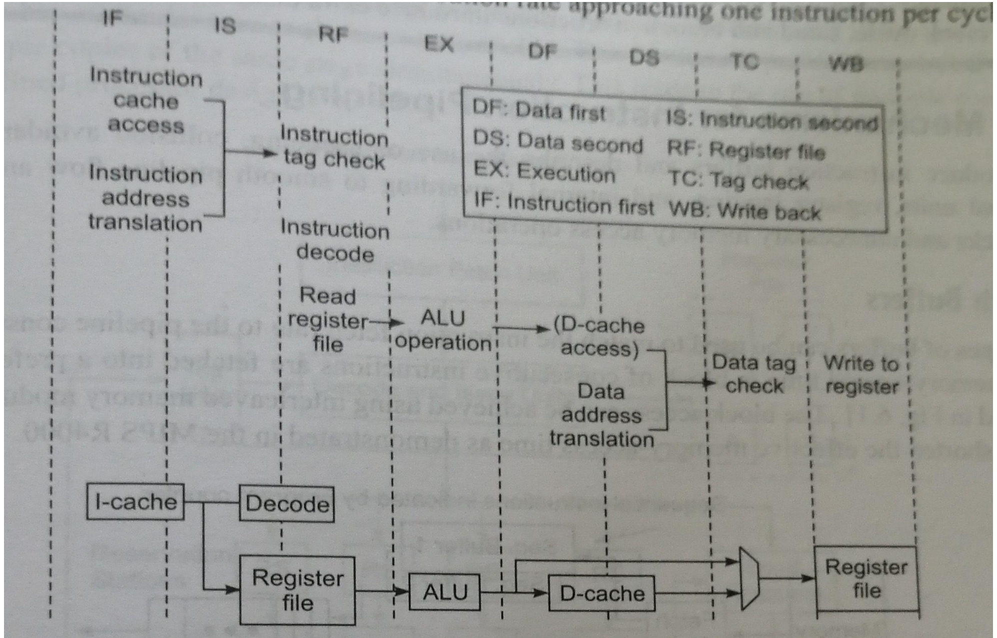
write-back stage is used to write results into the registers. Memory load or store operations are treated as part of execution.

MIPS R4000 Instruction Pipeline

- MIPS R4000 was a pipelined 64-bit processor using seperate instruction and data caches
- Eight-Stage pipeline for executing register-based instructions
- The processor pipeline design was targeted to achieve an execution rate approaching one instruction per cycle

R4000 Pipeline Stages

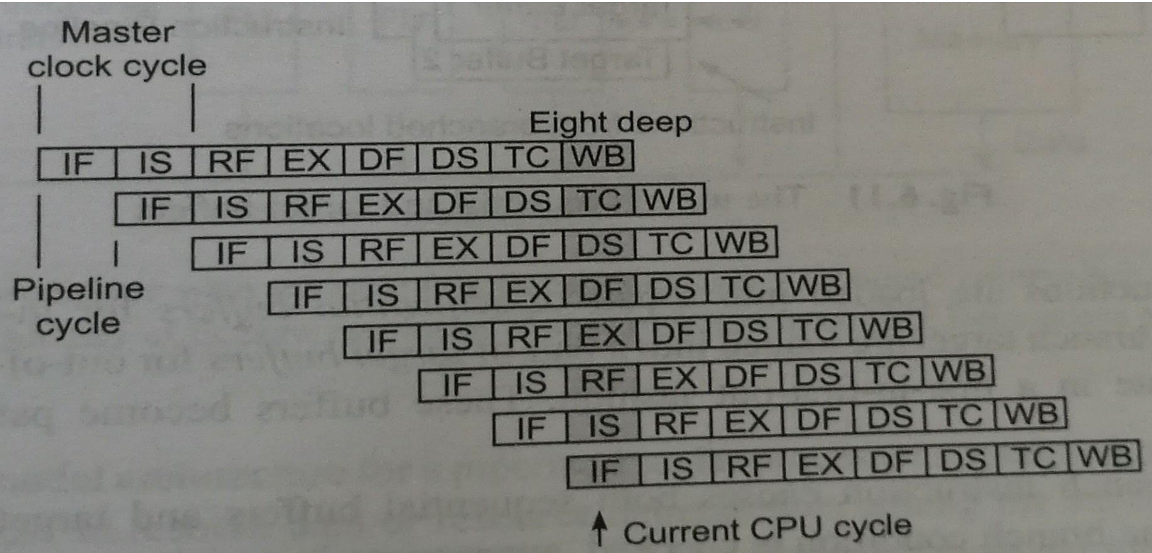
rate approaching one instruction per cycl



(a) R4000 pipeline stages

- The execution of each R4000 instruction consisted of eight major steps
- Each of these steps required approximately one clock cycle
- The instruction and data memory references are split across two stages.
- The single-cycle ALU stage took slightly more time than each of the cache access stages.

R4000 Instruction Overlapping in Pipeline



(b) R4000 instruction overlapping in pipeline

- This pipeline operated efficiently because different CPU resources,such as address and bus access,ALU operations,register access and so on,were utilized simultaneously on a non interfering basis
- The internal pipeline clock rate(100 Mhz) of the R4000 was twice the external input or master clock frequency
- Load and branch instructions introduce extra delays

PREFETCH BUFFERS

MECHANISM FOR INSTRUCTION PIPELINING

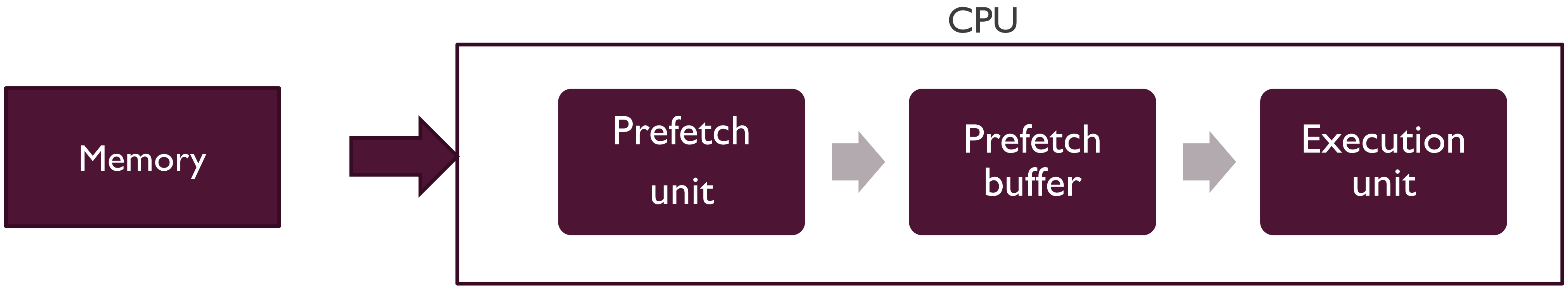
Abhirami P H
CSE S7
MUT15CS003

INSTRUCTION PREFETCH

- Technique which attempts to minimize the time a processor spends waiting for machine instruction to be fetched from memory.
- Instructions following the one being currently executed are loaded into a prefetch queue when the processor's external bus is otherwise idle.
- Instruction prefetch is combined with pipelining in an attempt to keep the pipeline busy.
- By 1995, most processors used prefetching. E.g. Motorola 680x0, intel 80x86.

INSTRUCTION LEVEL PARALLELISM-INSTRUCTION PREFETCH

- Break up the fetch-execute cycle and do the two in parallel.
- This dates to the IBM stretch (1959).



CONTD..

- The prefetch buffer is implemented in the CPU with on-chip registers.
- The prefetch buffer is implemented as a single register or a queue.
- Think of the prefetch buffer as containing the IR; When the execution of one instruction completes, the next one is already in the buffer and does not need to be fetched.
- Naturally, a program branch(loop structure, conditional branch etc.) invalidates the contents of the prefetch buffer, which needs to be reloaded.

Mechanisms for Instruction Pipelining - Multiple Functional Units

AKHIL GOKULDAS
ROLL NO : - 4
CSE - S7

What is instruction pipelining?

Instruction pipelining is a technique used in the design of modern microprocessors, microcontrollers and CPUs to increase their instruction throughput (the number of instructions that can be executed in a unit of time).

Concept & Working

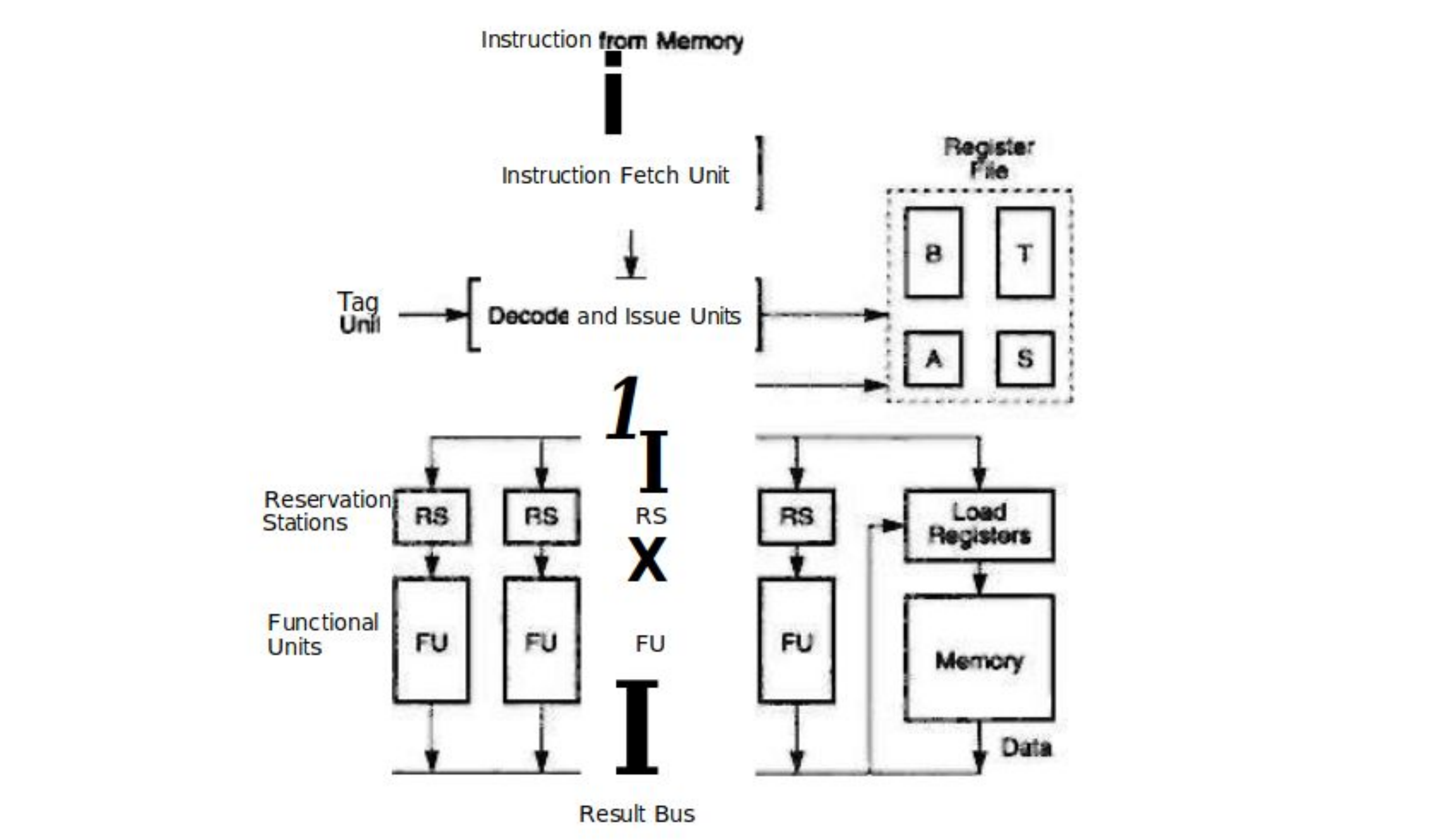
1. The main idea is to divide the processing of a CPU instruction, as defined by the instruction **microcode**, into a series of independent steps of micro-operations (also called **"microinstructions"**, **"micro-op"** or **"µop"**), with storage at the end of each step.
2. This allows the CPUs control logic to handle instructions at the processing rate of the slowest step, which is much faster than the time needed to process the instruction as a single step.

Background

Most modern CPUs are driven by a clock. The CPU consists internally of logic and memory (flip flops). When the clock signal arrives, the flip flops store their new value then the logic requires a period of time to decode the flip flops new values. Then the next clock pulse arrives and the flip flops store another values, and so on. By breaking the logic into smaller pieces and inserting flip flops between pieces of logic, the time required by the logic (to decode values till generating valid outputs depending on these values) is reduced.

Multiple Functional Units

- sometimes a certain pipeline stage becomes the bottleneck.
- This stage corresponds to the row with the maximum number of checkmarks in the reservation table.
- This bottleneck problem can be alleviated by using multiple copies of the same stage simultaneously.
- This leads to the use of multiple execution units in a pipelined processor design .



A pipelined processor with multiple functional units and distributed reservation stations supported by tagging.

Continuation

a model architecture for a pipelined scalar processor containing multiple functional units (Fig. 6.12). In order to resolve data or resources dependences among the successive instructions entering the pipeline, the reservation stations (RS) are used with each functional unit. Operands can wait in the RS until its data dependence have been resolved. Each RS is uniquely identified by a tag, which is monitored by a tag unit.

THANKYOU

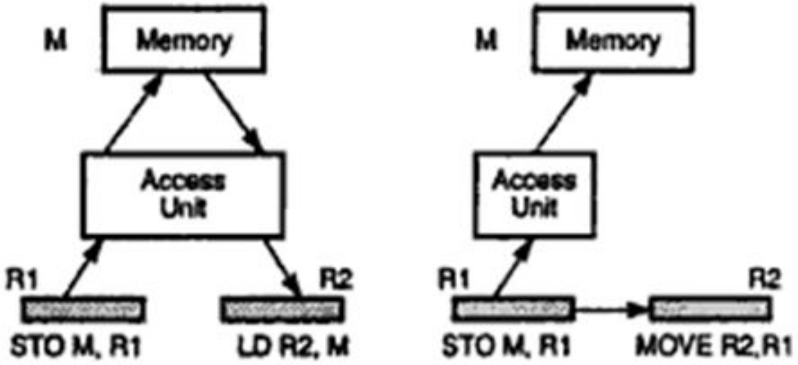
Internal Data Forwarding

CSA Assignment

- An optimization technique to speed up the performance and overcome data hazards.

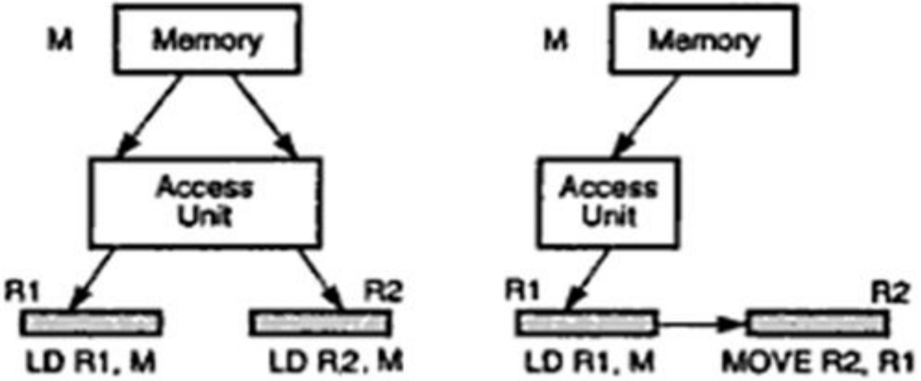
- Replacing unnecessary memory accesses by register-to-register transfers.
- After each stage in pipeline there are buffers to store the intermediate results.
- Take the result from the earliest point that it exists in any of the pipeline state registers and forward it to the functional units (e.g., the ALU) that need it that cycle
- Concept is explained in 3 directions: store-load, load-load & store-store forwarding.

Store – Load Forwarding

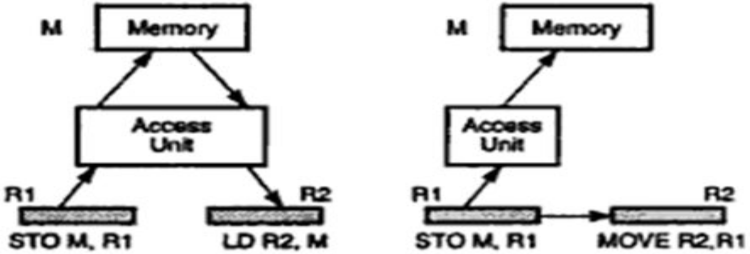


(a) Store-load forwarding

Load – Load Forwarding



Store – Load Forwarding



(a) Store-load forwarding

- Forwarding can achieve a CPI of 1 even in the presence of data dependencies

Mechanisms for Instruction Pipelining

Hazard Avoidance

Generic

Pipeline bubbling

- *Bubbling the pipeline*, also termed a *pipeline break* or *pipeline stall*, is a method to preclude data, structural, and branch hazards.
- As instructions are fetched, control logic determines whether a hazard could/will occur.
- If this is true, then the control logic inserts no operations (NOPs) into the pipeline.
- Thus, before the next instruction (which would cause the hazard) executes, the prior one will have had sufficient time to finish and prevent the hazard.

- If the number of NOPs equals the number of stages in the pipeline, the processor has been cleared of all instructions and can proceed free from hazards.
- All forms of stalling introduce a delay before the processor can resume execution.
- *Flushing the pipeline* occurs when a branch instruction jumps to a new memory location, invalidating all prior stages in the pipeline.
- These prior stages are cleared, allowing the pipeline to continue at the new instruction indicated by the branch.

Data hazards

There are several main solutions and algorithms used to resolve data hazards:

- insert a *pipeline bubble* whenever a read after write (RAW) dependency is encountered, guaranteed to increase latency, or
- use out-of-order execution to potentially prevent the need for pipeline bubbles
- use *operand forwarding* to use data from later stages in the pipeline

In the case of out-of-order execution, the algorithm used can be:

- scoreboarding, in which case a *pipeline bubble* is needed only when there is no functional unit available
- the Tomasulo algorithm, which uses register renaming, allowing continual issuing of instructions.

The task of removing data dependencies can be delegated to the compiler, which can fill in an appropriate number of NOP instructions between dependent instructions to ensure correct operation, or re-order instructions where possible.

Control hazards (branch hazards)

To avoid control hazards microarchitectures can:

- insert a *pipeline bubble* (discussed above), guaranteed to increase latency, or
- use branch prediction and essentially make educated guesses about which instructions to insert, in which case a *pipeline bubble* will only be needed in the case of an incorrect prediction

In the event that a branch causes a pipeline bubble after incorrect instructions have entered the pipeline, care must be taken to prevent any of the wrongly-loaded instructions from having any effect on the processor state, excluding energy wasted processing them before they were discovered to be loaded incorrectly.

Other techniques

- Memory latency is another factor that designers must attend to, because the delay could reduce performance.
- Different types of memory have different accessing time to the memory.
- Thus, by choosing a suitable type of memory, designers can improve the performance of the pipelined data path.

Tomasulo algorithm

Introduction

- Tomasulo’s algorithm is a computer architecture hardware algorithm for dynamic scheduling of instructions that allows out-of-order execution and enables more efficient use of multiple execution units.
- It was developed by Robert Tomasulo at IBM in 1967 and was first implemented in the IBM System/360 Model 91’s floating point unit.

Implementation concepts

- Common data bus
- Instruction order
- Register renaming
- Exceptions

Instruction lifecycle

- Stage 1: issue

    In the issue stage, instructions are issued for execution if all operands and reservation stations are ready or else they are stalled. Registers are renamed in this step, eliminating WAR and WAW hazards.

Stage 2: execute

In the execute stage, the instruction operations are carried out. Instructions are delayed in this step until all of their operands are available, eliminating RAW hazards. Program correctness is maintained through effective address calculation to prevent hazards through memory.

Stage 3: write result

In the write Result stage, ALU operations results are written back to registers and store operations are written back to memory

Tomasulo algorithm

Introduction

- Tomasulo’s algorithm is a computer architecture hardware algorithm for dynamic scheduling of instructions that allows out-of-order execution and enables more efficient use of multiple execution units.
- It was developed by Robert Tomasulo at IBM in 1967 and was first implemented in the IBM System/360 Model 91’s floating point unit.

Implementation concepts

- Common data bus
- Instruction order
- Register renaming
- Exceptions

Instruction lifecycle

- Stage 1: issue

  In the issue stage, instructions are issued for execution if all operands and reservation stations are ready or else they are stalled. Registers are renamed in this step, eliminating WAR and WAW hazards.

Stage 2: execute

In the execute stage, the instruction operations are carried out. Instructions are delayed in this step until all of their operands are available, eliminating RAW hazards. Program correctness is maintained through effective address calculation to prevent hazards through memory.

Stage 3: write result

In the write Result stage, ALU operations results are written back to registers and store operations are written back to memory

Branching Techniques And It’s Effects

Amal Binoy
S7 CSE
10

Branching Techniques

- Branching is a basic concept in computer science. It means an instruction that tells a computer to begin executing a different part of a program rather than executing statements one-by-one.
- Branching is implemented as a series of control flow statements in high-level programming languages. These can include:
- If statements
- For loops
- While loops
- Goto statements .

Branching instructions are also implemented at the CPU level, though they are much less sophisticated than the kinds of instructions found in high-level languages. These instructions are accessed through assembly programming and are also referred to as "jump" instructions.

Effect Of Branching

One of the major problems in designing an instruction pipe line is assuming a steady flow of instructions to the initial stages of the pipeline.

The primary problem is the conditional branches instruction until the instruction is actually executed, it is impossible to determine whether the branch will be taken or not.

A variety of approaches have been taken for dealing with conditional branches:

- Multiple streams
- Prefetch branch target
- Loop buffer
- Branch prediction
- Delayed branch

- **Multiple streams**

  A single pipeline suffers a penalty for a branch instruction because it must choose one of two instructions to fetch next and sometimes it may make the wrong choice. A brute-force approach is to replicate the initial portions of the pipeline and allow the pipeline to fetch both instructions, making use of two streams.

- **Prefetch Branch target**

  When a conditional branch is recognized, the target of the branch is prefetced, in addition to the instruction following the branch. This target is then saved until the branch instruction is executed. If the branch is taken, the target has already been prefetched.

- **Loop Buffer:**

  A top buffer is a small, very high speed memory maintained by the instruction fetch stage of the pipeline and containing the most recently fetched instructions, in sequence. The loop buffer is similar in principle to a cache dedicated to instructions. The differences are that the loop buffer only retains instructions in sequence and is much smaller in size and hence lower in cost.

- **Branch Prediction:**

Various techniques can be used to predict whether a branch will be taken or not. The most common techniques are:

- Predict never taken
- Predict always taken
- Predict by opcode
- Taken/not taken switch
- Branch history table.

- **Delayed branch**

A conditional branch instruction found in some RISC architectures that include pipelining. The effect is to execute one or more instructions following the conditional branch before the branch is taken. This avoids stalling the pipeline while the branch condition is evaluated, thus keeping the pipeline full and minimizing the effect of conditional branches on processor performance.

THANK YOU

Branch Handling Techniques - Branch Prediction

ANAKHA SADANANDAN
S7 CSE
ROLL NO: 11

Branch Handling Techniques - Branch Prediction

- Branch prediction is one of the ancient performance improving techniques which still finds relevance into modern architectures.
- While the simple prediction techniques provide fast lookup and power efficiency they suffer from high misprediction rate.
- On the other hand, complex branch predictions – either neural based or variants of two level branch prediction – provide better prediction accuracy but consume more power and complexity increases exponentially.
- In addition to this, in complex prediction techniques the time taken to predict the branches is itself very high – ranging from 2 to 5 cycles – which is comparable to the execution time of actual branches.
- Branch prediction is essentially an optimization (minimization) problem where the emphasis is on to achieve lowest possible miss rate, low power consumption and low complexity with minimum resources.

Different types of branches

- **Forward conditional branches** - based on a run-time condition, the PC (Program Counter) is changed to point to an address forward in the instruction stream.
- **Backward conditional branches** - the PC is changed to point backward in the instruction stream. The branch is based on some condition, such as branching backwards to the beginning of a program loop when a test at the end of the loop states the loop should be executed again.
- **Unconditional branches** - this includes jumps, procedure calls and returns that have no specific condition.

Static Branch Prediction

Static Branch Prediction predicts always the same direction for the same branch during the whole program execution. It comprises hardware-fixed prediction and compiler-directed prediction. Simple hardware-fixed direction mechanisms can be:

- Predict always not taken
- Predict always taken
- Backward branch predict taken, forward branch predict not taken.

Sometimes a bit in the branch opcode allows the compiler to decide the prediction direction.

Dynamic Branch Prediction

- Dynamic Branch Prediction: the hardware influences the prediction while execution proceeds.
- Prediction is decided on the computation history of the program.
- During the start-up phase of the program execution, where a static branch prediction might be effective, the history information is gathered and dynamic branch prediction gets effective.
- In general, dynamic branch prediction gives better results than static branch prediction, but at the cost of increased hardware complexity.

Branch Handling Techniques - Delayed Branches

Ananth Krishna KS
S7 CSE
12

What is branch ?

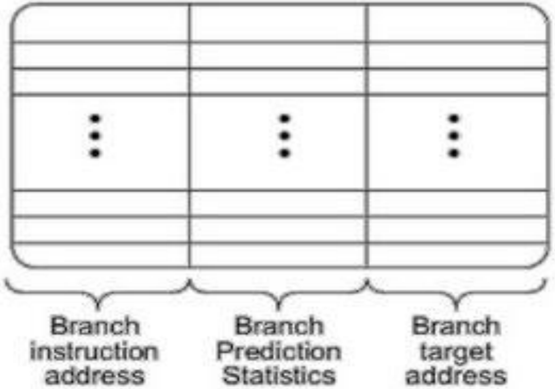
- A **branch** is an instruction in a computer program that can cause a computer to begin executing a different instruction sequence and thus deviate from its default behavior of executing instructions in order.
- **Branch** may also refer to the act of switching execution to a different instruction sequence as a result of executing a branch instruction. Branch instructions are used to implement control flow in program loops and conditionals.

Branch Handling Techniques

- Branch taken,Branch target,Delayed slot.
- Effect of branching:
    - Parameters:
        - k :Number of stages in pipeline.
        - n :Total no of instructions
        - p :Percentage of branch instructions over n.
        - q :Percentage of successful branch instructions over p.
        - b :Delay slot.
        - t :Pipeline cycle.
    - Branch penalty :q of (p of n) * bT
    - Execution time:[k+(n-1)+pqnb]*T

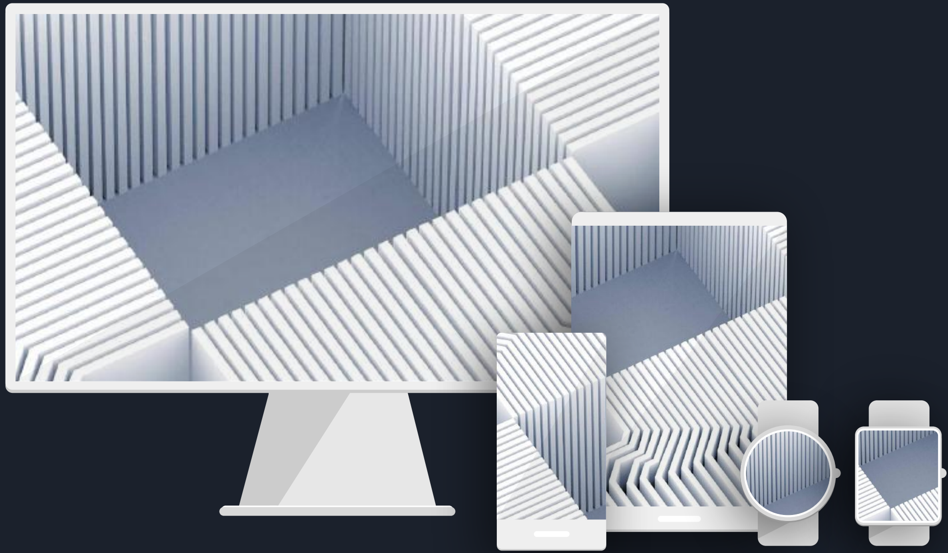
Delayed Branches

- A delayed branch of d cycles allow the at most d-1 useful instruction to be executed following the broken branch.
- Execution of these instruction should be independent of branch instruction to achieve a zero branch penalty.



(a) Branch target buffer organization

Thank you!



Fixed Point Operations

o To convert a number from a fixed point type with scaling factor R to another type with scaling factor S, the underlying integer must be multiplied by R and divided by S; that is, multiplied by the ratio R/S. Thus, for example, to convert the value 1.23 = 123/100 from a type with scaling factor R=1/100 to one with scaling factor S=1/1000, the underlying integer 123 must be multiplied by (1/100)/(1/1000) = 10, yielding the representation 1230/1000.

- To add or subtract two values of the same fixed-point type, it is sufficient to add or subtract the underlying integers, and keep their common scaling factor. The result can be exactly represented in the same type, as long as no overflow occurs (i.e. provided that the sum of the two integers fits in the underlying integer type). If the numbers have different fixed-point types, with different scaling factors, then one of them must be converted to the other before the sum.

- o multiply two fixed-point numbers, it suffices to multiply the two underlying integers, and assume that the scaling factor of the result is the product of their scaling factors. This operation involves no rounding. For example, multiplying the numbers 123 scaled by 1/1000 (0.123) and 25 scaled by 1/10 (2.5) yields the integer 123×25 = 3075 scaled by (1/1000)×(1/10) = 1/10000, that is 3075/10000 = 0.3075. If the two operands belong to the same fixed-point type, and the result is also to be represented in that type, then the product of the two integers must be explicitly multiplied by the common scaling factor; in this case the result may have to be rounded, and overflow may occur. For example, if the common scaling factor is 1/100, multiplying 1.23 by 0.25 entails multiplying 123 by 25 to yield 3075 with an intermediate scaling factor of 1/10000. This then must be multiplied by 1/100 to yield either 31 (0.31) or 30 (0.30), depending on the rounding method used, to result in a final scale factor of 1/100.

- divide two fixed-point numbers, one takes the integer quotient of their underlying integers, and assumes that the scaling factor is the quotient of their scaling factors. The first division involves rounding in general. For example, division of 3456 scaled by 1/100 (34.56) and 1234 scaled by 1/1000 (1.234) yields the integer 3456÷1234 = 3 (rounded) with scale factor (1/100)/(1/1000) = 10, that is, 30. One can obtain a more accurate result by first converting the dividend to a more precise type: in the same example, converting 3456 scaled by 1/100 (34.56) to 3,456,000 scaled by 1/100000, before dividing by 1234 scaled by 1/1000 (1.234), would yield 3456000÷1234 = 2801 (rounded) with scaling factor (1/100000)/(1/1000) = 1/100, that is 28.01 (instead of 30). If both operands and the desired result are represented in the same fixed-point type, then the quotient of the two integers must be explicitly divided by the common scaling factor.

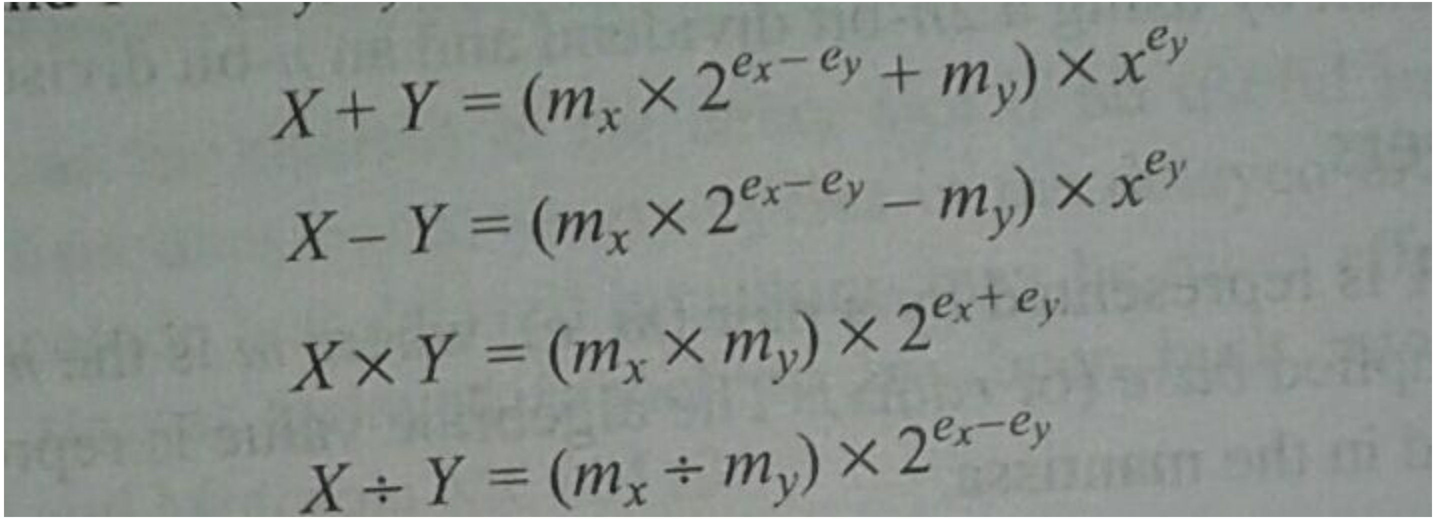
Floating Point Operation and Floating Point Number

Floating Point Number

- A floating point number X is represented by a pair (m,e)
  - m - mantissa
  - e - exponent with an implied base
- The algebraic value is represented as X= m x (r^e)
  - The sign of X can be embedded in mantissa.

Floating-point operations

Four primitive operations are defined below for a pair of floating point number represented by $X = (m_x, e_x)$ and $Y = (m_y, e_y)$.. For clarity, we assume $e_x \leq e_y$ and base r=2.



$$X + Y = (m_x \times 2^{e_x - e_y} + m_y) \times x^{e_y}$$

$$X - Y = (m_x \times 2^{e_x - e_y} - m_y) \times x^{e_y}$$

$$X \times Y = (m_x \times m_y) \times 2^{e_x + e_y}$$

$$X \div Y = (m_x \div m_y) \times 2^{e_x - e_y}$$

- The above equations clearly identifies the number of arithmetic operations involved in each floating point function.
- These operations can be divided into 2.
    - For exponent operations such as comparing their relative magnitude or adding or subtracting them.
    - For mantissa operations, including 4 type of fixed operations.
- Floating-point units are ideal for pipelined implementation.

- Two halves of operation demand almost twice as much hardware as that required in fixed point unit.
- Arithmetic shifting operations are needed for equalizing the 2 exponent before their mantissa can be added or subtracted.
- In addition, normalization of a floating point number also require left shift to be performed.

Static Arithmetic Pipelines - Arithmetic Pipeline Stages

Anu S Alunkal
15-CSE

Introduction

- Arithmetic pipelines are designed to perform fixed functions.
- These arithmetic/logic units performs fixed and floating points operations separately.
- These arithmetic unit performs scalar operations involving one pair of operands at a time.
- The pipelining in scalar arithmetic pipelines are controlled by software loops.
- The pipelining in vector arithmetic pipelines are hardware controlled.

- Both scalar and vector pipelines differ in register files and control mechanism.

Arithmetic pipeline stages

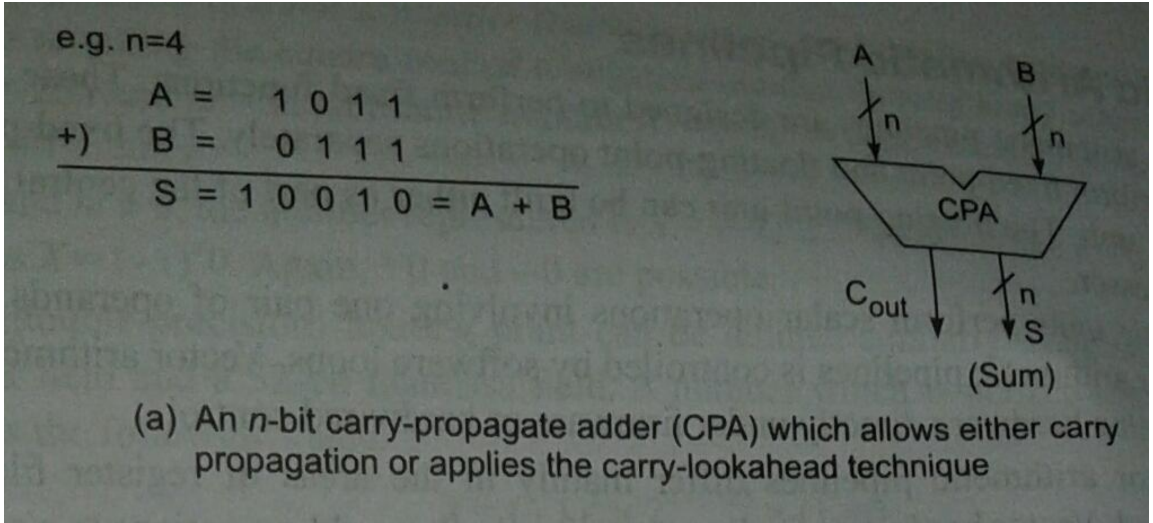
- Depending on the functions to be implemented, different pipeline stages in an arithmetic unit require different hardware logic.
- Since all arithmetic operations can be implemented with the basic add and shifting operations, the core arithmetic stages require some form of hardware to add and to shift.
- For eg: a typical 3 stage floating point adder includes

1.Exponent comparison and equalization

2.Fraction Addition

3.Fraction normalization and exponent readjustment

- (1)can be easily implemented with shift register.
- High speed addition requirement into use carry propagation adder which adds 2 numbers and produces an arithmetic sum (in figure.)
- (3)is implemented using another shift register and addition logic.



e.g. n=4

```
     A =   1 0 1 1
+)   B =   0 1 1 1
-------------------------
     S = 1 0 0 1 0 = A + B
```

(a) An *n*-bit carry-propagate adder (CPA) which allows either carry propagation or applies the carry-lookahead technique

Thank you!

Static Arithmetic Pipelines

ARAVIND M MENON
R.NO 16
CSE S7

INRODUCTION

Pipelining is one way of improving the overall processing performance of a processor. This architectural approach allows the simultaneous execution of several instructions. Pipelining is transparent to the programmer; it exploits parallelism at the instruction level by overlapping the execution process of instructions. It is analogous to an assembly line where workers perform a specific task and pass the partially completed product to the next worker.

ARITHMATIC PIPELINING

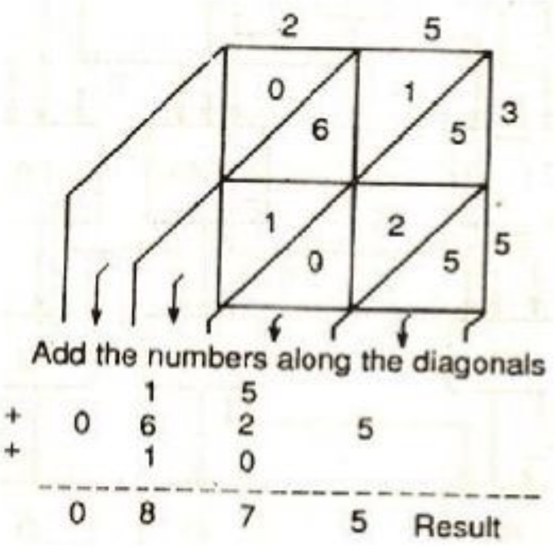
Some functions of the arithmetic logic unit of a processor can be pipelined to maximize performance. An arithmetic pipeline is used for implementing complex arithmetic functions like floating-point addition, multiplication, and division. These functions can be decomposed into consecutive subfunctions. For example Figure 3.13 presents a pipeline architecture for floating-point addition of two numbers. (A nonpipelined architecture of such an adder is described in Chapter 2.) The floating-point addition can be divided into three stages: mantissas alignment, mantissas addition, and result normalization [MAN 82,HAY 78].

FIXED POINT MULTIPLICATION PIPELINE

- A pipelined multiplier based on the digit products can be designed using digit product generation logic and the digit adders.

Example:

25 * 35 = 875



Now for binary multiplication:

A = a1 a0

B = b1 b0

	a1	a0
	b1	b0
	a1b0	a0b0
a1b1	aob1	
a1b1	a1bo + a0b1	a0b0

STATIC ARITHMETIC PIPELINES

CONVERGENCE DIVISION

Archana Venugopal
CSE S7 17

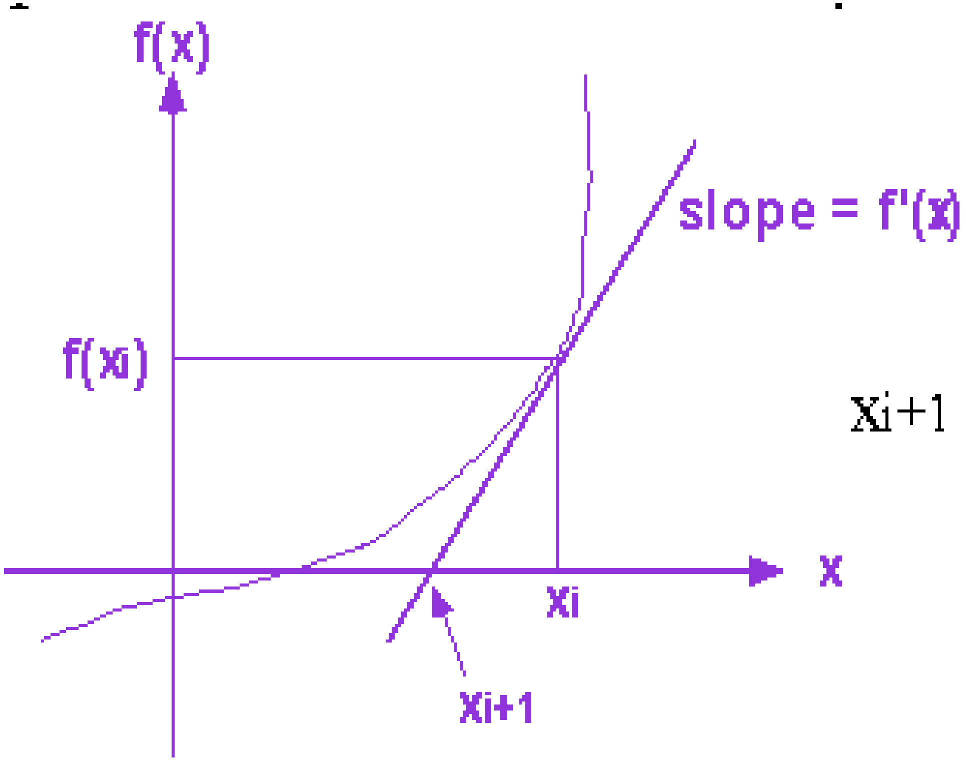
STATIC PIPELINES

- Pipelines are usually divided into two classes: instruction pipelines and arithmetic pipelines.
- There are two types of pipelines: static and dynamic.
- A static pipeline can perform only one function (such as addition or multiplication) at a time.
- Static pipelines are linear - A linear pipeline processor is a series of processing stages and memory access.
- The operation of a static pipeline can only be changed after the pipeline has been drained. (A pipeline is said to be drained when the last input data leave the pipeline.)

- For example,

  Consider a static pipeline that is able to perform addition and multiplication. Each time that the pipeline switches from a multiplication operation to an addition operation, it must be drained and set for the new operation. The performance of static pipelines is severely degraded when the operations change often, since this requires the pipeline to be drained and refilled each time.

CONVERGENCE DIVISION



$$x_{i+1} = x_i - f(x_i)/f'(x_i)$$

CONVERGENCE DIVISION

- Generate the reciprocal of the divisor by an iterative process and then use:

  A/B = A* (1/B)

- Use (great, great, great grand-uncle) Newton Raphson method to solve for (1/B).

  Xi+1 = Xi-f(Xi)/f'(Xi)

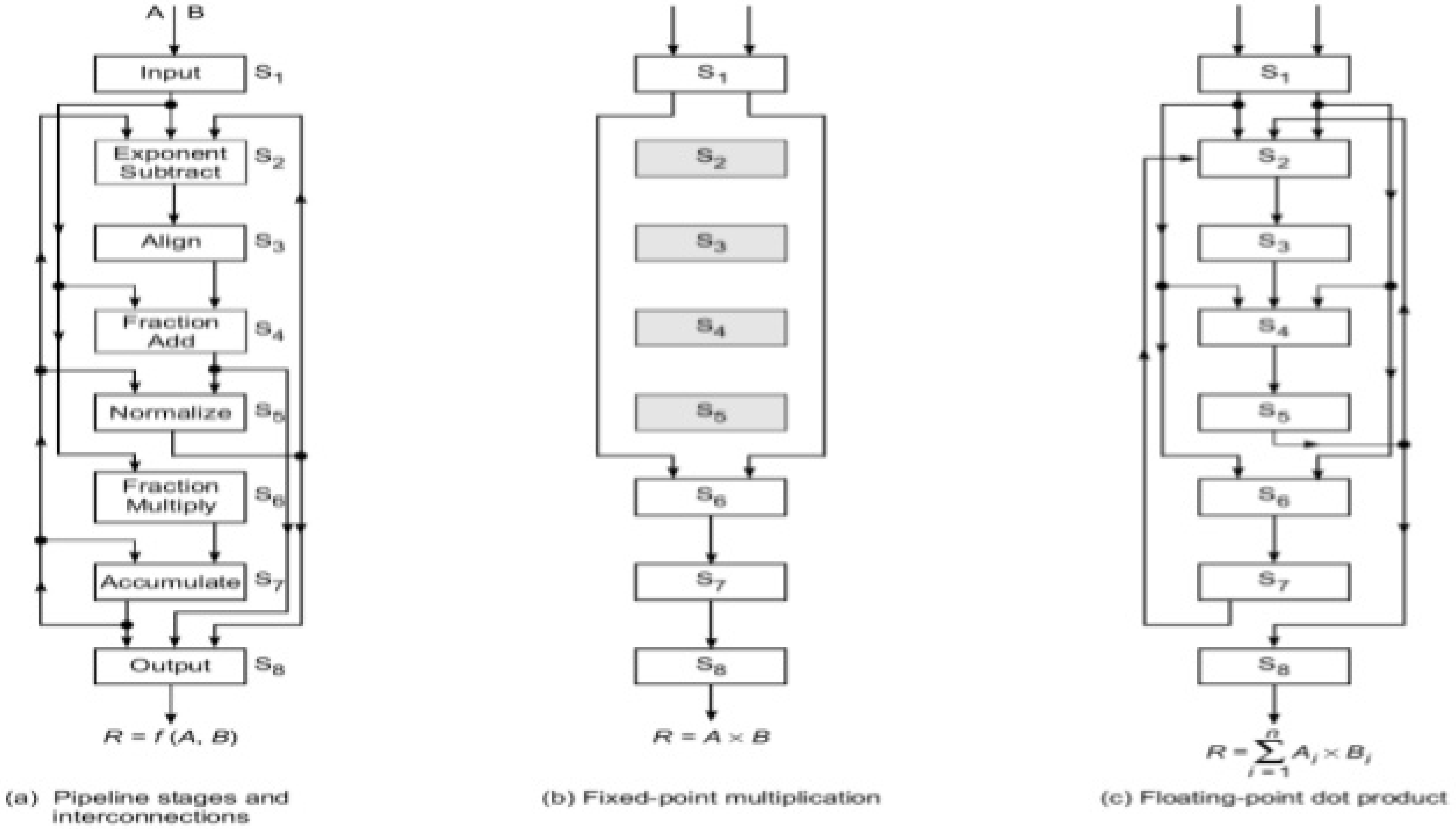
MULTIFUNCTIONAL ARITHMETIC PIPELINES

Basil Reji
CSE S7 18

DEFINITION

- When different functions at different times are performed through the pipeline,this is called Multifunctional Pipelines.
- Multifunctional pipelines reconfigurable at different times according to the operations being performed.
- It is interconnecting different subsets of stages in the pipeline.

- It has
  - One instruction processing unit.
  - Four memory buffer units.
  - Fout arithmetic units.
- It provides four parallel execution pipelines below the IPU.
- Any mixture of scalar and vector instructions can be executed simultaneously in four pipes.



(a) Pipeline stages and interconnections

(b) Fixed-point multiplication

(c) Floating-point dot product

Fig. 6.27 The multiplication arithmetic pipeline of the TI Advanced Scientific Computer and the interstage connections of two representative functions (Shaded stages are unutilized)

TYPES

- Static
  - Initially configured for one functional evaluation.
  - For another function,it need to be drained and reconfigured.
  - You cannot have two inputs of different function at the same time.

- Dynamic
  - Can do different functional evaluation at a time.
  - It is difficult to control as we need to be sure that there is no conflict in usage of stages.

Superscalar Pipeline Design - Pipeline Design Parameters

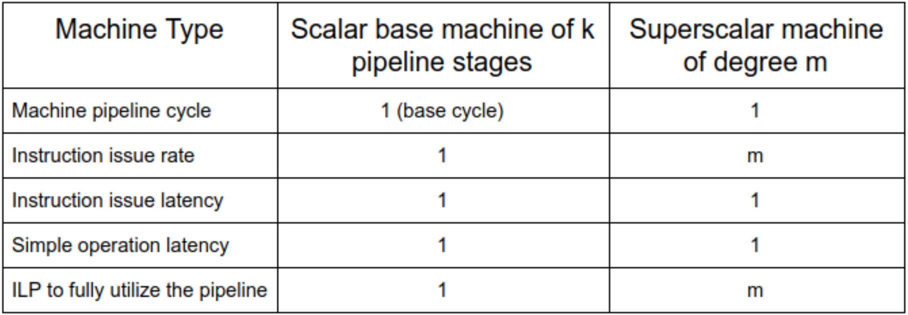
Basil K Y

November 21, 2018

Pipeline design parameters

- Instruction issue rate : The number of instructions issued per cycle.
- Instruction issue latency : Time required between issuing of two adjascent instructions.
- Simple operation latency : Latency of simple operations like integer adds, loads, stores, moves etc.
- Instruction level parallelism(ILP) is the maximum number of instructions that can be simultaneously executed in the pipeline.
- Pipeline cycle for the scalar base processor is assumed to be 1 time unit, called the base cycle.

Design parameters for pipeline design



Machine Type	Scalar base machine of k pipeline stages	Superscalar machine of degree m
Machine pipeline cycle	1 (base cycle)	1
Instruction issue rate	1	m
Instruction issue latency	1	1
Simple operation latency	1	1
ILP to fully utilize the pipeline	1	m

SuperScalar Pipeline Scheduling

Elizabath Saba
S7,CSE(23)

- Instruction issue and completion policies are critical to superscalar processor performance.
- Based on the way a dispatch unit of superscalar processor issue the instructions,superscalar architectures can be classified into
  - In-Order issue processor
  - Out-Of -Order issue processor

In-Order Issue processor

- Issue instructions in the order in which they appear in the program .
- Consider the below figure in which superscalar processor of degree 2 is shown

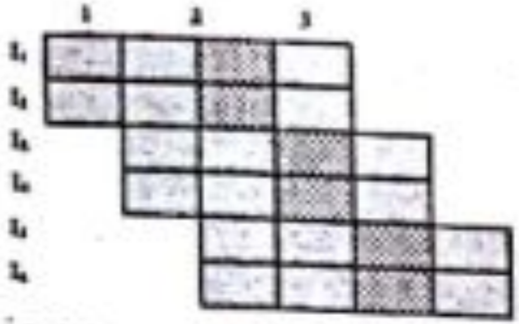
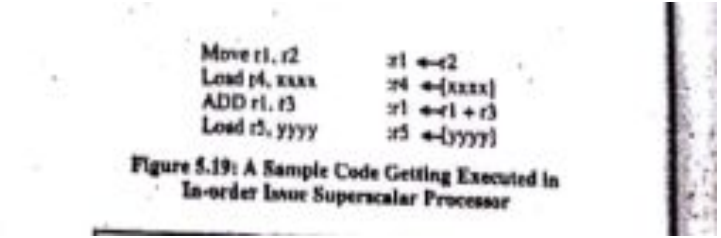


Figure 5.18: Superscalar Processor of Degree 2 with 4-Staged Pipeline

- The pattern in which piece of code get executed is



```
Move r1, r2         r1 ← r2
Load r4, xxxx       r4 ← [xxxx]
ADD r1, r3          r1 ← r1 + r3
Load r5, yyyy       r5 ← [yyyy]
```

Figure 5.19: A Sample Code Getting Executed in In-order Issue Superscalar Processor

- The first two instructions are fetched simultaneously and decoded and scheduled for execution in parallel.
- Since add instruction has a raw dependency with the move instruction,it is stalled for one cycle.

Out of order issue processor

- In out-of-order issue processor it would have not stalled the last load instruction.
- It provides much better performance than in order issue processor.
- It requires much more complex hardware to implement.