

## CS704 – Advanced Computer Architecture

### 1. Pipelining:

A technique used in advanced microprocessors where the microprocessor begins executing a second instruction before the first has been completed. That is, several instructions are in the *pipeline* simultaneously, each at a different processing stage. The pipeline is divided into segments and each segment can execute its operation concurrently with the other segments. When a segment completes an operation, it passes the result to the next segment in the pipeline and fetches the next operation from the preceding segment. The final results of each instruction emerge at the end of the pipeline in rapid succession.

#### **Instruction Cycle:**

An **instruction cycle** (sometimes called fetch-and-execute **cycle**, fetch-decode-execute **cycle**, or FDX) is the basic operation **cycle** of a computer. It is the process by which a computer retrieves a program **instruction** from its memory, determines what actions the **instruction** requires, and carries out those actions.

#### **Finite State Machine:**

A **finite-state machine** (FSM) or **finite-state** automaton (plural: automata), or simply a **state machine**, is a mathematical model of computation used to design both computer programs and sequential logic circuits. It is conceived as an abstract **machine** that can be in one of a **finite** number of states

#### **Branch predication:**

In computer architecture, a **branch predictor** is a digital circuit that tries to guess which way a **branch** (e.g. an if-then-else structure) will go before this is known for sure. The purpose of the **branch predictor** is to improve the flow in the instruction pipeline.

#### **Dynamic Scheduling:**

**Dynamic** priority **scheduling** is a type of scheduling algorithm in which the priorities are calculated during the execution of the system. The goal of **dynamic** priority scheduling is to adapt to dynamically changing progress and form an optimal configuration in self-sustained manner. A static schedule is some kind of list containing the order of the processes and the durations in which they are scheduled. **Presence or absences of control hazard change the pipeline.** This exercise asks, “How much faster would the machine be” which should make you immediately think speedup. In this case, we are interested in how the presence or absence of control hazards changes the pipeline speedup

#### **How the operands and operations for media and signal processing differ Operands:**

Vertex – A common 3D data type dealt in graphics applications – four components: (x, y, z) and w=color or hidden surfaces – vertex values are usually 32-bit floating-point values – Three vertices specify a graphics primitive such as a triangle • Pixel – Typically 32 bits, consisting of four 8-bit channels • R (red), G (green), B (blue), and A (attribute: eg. transparency) • DSPs add fixed point – fractions between -1 and +1 (divide by  $2^{n-1}$ ) • Blocked floating point – a block of

variables with common exponent – accumulators, registers that are wider to guard against round-off error to aid accuracy in fixed-point arithmetic. Operation: Data for multimedia operations is often narrower than the 64-bit data word – normally in single precision, not double precision • Single-instruction multiple-data (SIMD) or vector instructions – A partitioned add operation on 16-bit data with a 64-bit ALU would perform four 16-bit adds in a single clock cycle • Hardware cost: prevent carries between the four 16-bit partitions of the ALU – Two 32-bit floating-point operations (paired single operations) • The two partitions must be insulated to prevent operations on one half from affecting the other

**What is the penalty in clock cycles?**

Storing the target instruction of an unconditional branch effectively removes one instruction. If there is a BTB hit in instruction fetch and the target instruction is available, then that instruction is fed into decode in place of the branch instruction. The penalty is –1 cycle. In other words, it is a performance gain of 1 cycle.

**Determine the improvement from branch folding for unconditional branches:**

If the BTB stores only the target address of an unconditional branch, fetch has to retrieve the new instruction. This gives us a CPI term of  $5\% \cdot (90\% \cdot 0 + 10\% \cdot 2)$  or 0.01. The term represents the CPI for unconditional branches (weighted by they frequency of 5%). If the BTB stores the target instruction instead, the CPI term becomes  $5\% \cdot (90\% \cdot (-1) + 10\% \cdot 2)$  or –0.035. The negative sign denotes that it reduces the overall CPI value. The hit percentage to just break even is simply 20%.

**Computer A has a clock cycle time of 250 ps and a CPI of 2 for some program, and computer B has a clock cycle time of 500 ps and a CPI of 1.2 for the same program. Which computer is faster for this program and how much?**

$ET = IC \cdot CPI \cdot CT \quad \quad \quad ETA = IC \cdot 2 \cdot 250 = 500IC \quad \quad \quad ETB = IC \cdot 1.2 \cdot 500 = 600IC$

**Impact of increasing the size of branch-prediction:**

A single predictor predicting a single branch is generally more accurate than is that same predictor serving more than one instructions; and It is less likely that two branches in a program share a single predictor. Therefore, increasing the size of predictor buffer does not have significant effect on two branches in a program

**Compute C = A+B and what is the total code size?**

Stack	Accumulator	Register-memory	Register
Push A(72)	Load A(72)	Load R1, A(80)	Load R1, A(80)
Push B(72)	Add B(72)	Add R2, R1, B(88)	Load R2, B(80)
Add(8)	Store c(72)	Store c, R2(80)	Add R3, R1, R2(80)
Pop c(72)			

**Assuming no stalls, what is the speedup of the pipelined machine over the single stage machine? Speed up=5/1=5**

Pipelined machine with 5 pipeline stages cycle time is 5 ns and the latter is 1 ns . what is speed

up ??Is ka solution ( pipeline depth) /(1 + stall cycles per instruction)

**Given the pipeline stalls 1 cycle for 40% of the instructions, what is the speed up now?**

Speed up =  $(1 \text{ CPI} * 5\text{ns}) / (1.4 \text{ CPI} * 1\text{ns}) = 3.58$  Let us analyze the problem a bit:

i. The Opcode field of the instruction has five bits available i.e. from bit 9 to

13. Hence this amounts to a total of 25

ii. As stated in the question that when operand 2 field is all zeros, each of original Opcodes takes on new meaning. This means 32 additional Opcodes if the value of operand 2 is all zero.

iii. Mode field consists of two bytes hence 22 possible. Theoretically speaking for each combination of mode field there is a whole class of Opcodes available.

iv. w/b field consists of one bit so only two possible values exist. Hence in this case also for each possible value of w/b field, there is a whole class of Opcodes available. = 32 Opcodes. = 4 different modes are

Hence there comes a total of  $(32 + 32) * 4 * 2 = 512$  possible Opcodes. These Opcodes result from different combinations of mode field, w/b field and operand 2 field.

**Suggest an efficient way to provide more opcodes:**

In order to provide room for additional Opcodes, we must sacrifice some functionality of the system. We can't reduce the field size of mode field as it will severely reduce the functionality of the system and the total Opcodes will also remain the same i.e.  $(64 + 64) * 2 * 2 = 512$ . Also the w/b field can't be reduced further as it is already 1 bit of size. We also can't reduce the size of operand fields because they have to access sixteen general purpose registers that is possible only through 4 bit value. Hence some other strategy must be considered. As we have seen that operand 2 can't access general purpose register 0 this is because if it has to access register 0 it will place its address as 0000 but this all zero code actually changes the meaning of original Opcodes. We can apply the same technique to operand 1 also. By making it not to use general purpose register 15 or R15 which is addressed as 1111 we can assign new meanings to all original Opcodes. Hence by making the operand 1 field not use R15 and saying that if this field contains all 1s then all original Opcodes will have different meanings we can increase the number of Opcodes to  $(32 + 32 + 32) * 4 * 2 = 768$ . So we have increased the number of Opcodes by  $768 - 512$  by 256 after sacrificing R15 from operand 1 f

**What is the length of a clock cycle?**

$\text{Time(lw)} = \text{Time(IF)} + \text{Time(ID+Reg.File)} + \text{Time(ALU)} + \text{Time(MemRead)}$        $\text{Time(lw)} = 150 + (70+60) + 200 + 60$        $\text{Time(lw)} = 690\text{ps}$

**What would be the frequency of a processor?**

As we know Frequency = The value of clock cycle for single cycle is 1.

It means that frequency of processor corresponding to single cycle datapath is 1. Frequency can be defined as:  $\text{Frequency} = 1 / (\text{clock rate} + \text{Time(Reg.FileWrite)})$   $1\text{GHz}$  where c is the maximum clock cycle.  $= 1 / 690 * 10^{-12} * 10^9 = 1 / 690 * 10^{-3} = 1000 / 690 = 1.44 \text{ GHz} = 1.45 \text{ GHz}$  **What**

### Would be the length of fastest clock cycle for a 5-stage pipeline

Fastest clock cycle is that whose latency is minimum i.e 60 ps. but when we calculate the frequency, we have to consider the slowest cycle length in multi-cycle data path that is 200 ps.

Frequency = So Frequency =  $1200 \text{ ps}^{-1} = \frac{1}{200 \times 10^{-12}} = \frac{1000}{200} = 5 \text{ GHz}$

### How much faster is the 5-stage pipelined:

Single cycle execution time= 690 ps Execution time for multi cycle = 200ps So  $690/200\text{ps}=3.45$  times faster It means that 5-stage pipelined datapath is 3.45 time faster than single cycle datapath.If we analyze this according to frequency point of view then  $5\text{GHz}/1.45\text{GHz}=3.45$  times faster.

### Normalize the loop:

- i. By normalize the loop, it leads to a modified c code as shown below;
- ii. The GCD test shows the potential for dependences written an array indexed

For (i = 1; i < 50; i++) { A[2 \* i] = a [(100 \* i) + 1] ;multiple constant by 2 }

by the function,  $Ai + b$  and  $ci + d$  only, If the condition  $(d-b) \bmod \text{GCD}(c, a) = 0$  is satisfied. Now, applying GCD test, in that case we will get,  $a=2$ ,  $b=0$ ,  $c=100$  that allows us to determine dependence in loop. Thus, GCD will be,  $\text{GCD}(2, 100) = 2$  and  $d-b=1$ . Here, as 1 is factor of 2. Thus, GCD test indicates that there is dependence in the code. In reality, there is no dependence in the code. Since the loop lead it value from  $a[101]$ ,  $a[201]$ .... $a[5001]$  and again these values to  $a[2]$ ,  $a[4]$ ..... $a[100]$ . **affine index:**An array index is affine if it can be written in the form of an expression. Here,  $a$  and  $b$  are constant, and  $i$  is the loop index variable. E.g. in the loop

For (i = 1 ; i < 100; i = i + 1){ X[2 \* i + 3] = x[2 \* i] \* 5.0; } Here, the index value  $X[2 * i + 3]$  is affine with  $a = 2$  and  $b = 3$

### How compiler finds dependences:

The compiler detects the dependence using dependence analysis algorithm and this algorithm works on assumptions that:-Array indices are affine -There exist GCD of two affine indices .

For (i = 1000; i > 0; i = i - 1) X[i] = X[i] + s ;

### What is meant by anti dependence?

An anti-dependence occurs between an instruction that reads a register (or memory location) and a subsequent instruction writes a new value to the same location. Equivalently, two instructions have an anti-dependence if swapping their order would result in a true dependence. An anti-dependence exists between the sw instruction, which reads \$4, and the or instruction, which overwrites \$4 with a new value.

### What is the penalty in clock cycles:

Storing the target instruction of an unconditional branch effectively removes one instruction. If there is a BTB hit in instruction fetch and the target instruction is available, then that instruction

is fed into decode in place of the branch instruction. The penalty is  $-1$  cycle. In other words, it is a performance gain of 1 cycle.

#### **Enhancement to produce a performance gain:**

If the BTB stores only the target address of an unconditional branch, fetch has to retrieve the new instruction. This gives us a CPI term of  $5\% \cdot (90\% \cdot 0 + 10\% \cdot 2)$  or 0.01. The term represents the CPI for unconditional branches (weighted by their frequency of 5%). If the BTB stores the target instruction instead, the CPI term becomes  $5\% \cdot (90\% \cdot (-1) + 10\% \cdot 2)$  or  $-0.035$ . The negative sign denotes that it reduces the overall CPI value. The hit percentage to just break even is simply 20%.

#### **Find the die yield for a processor:**

Chip with the following manufacturing cost factors: die size =  $380 \text{ mm}^2$ , estimated defect rate = 0.75 per  $\text{cm}^2$ ,  $\alpha = 4$   $(1 + 0.75 \cdot 380/4)^{-4} = 0.0000000366985$ .

#### **Compare performance of two programs 1: 2sec , 1.5sec wala:**

(2 pts) For P1, M2 is  $4/3$  (2 sec/1.5 sec) times as fast as M1. For P2, M1 is 2 times as fast (10 sec/5 sec) as M2.

#### **Computer Execution rate Performance:**

A  $5 \times 10^9 / 2 \text{ sec} = 2.5 \times 10^9 = 2.5 \times 10^9 \text{ IPS}$       B  $6 \times 10^9 / 1.5 \text{ sec} = 4 \times 10^9 = 4 \times 10^9 \text{ IPS}$

#### **Execution rate on M1:**

For program 1, M2 is  $2.0/1.5 = 1.33$  times as fast as M1. For program 2, M1 is  $10.0/5.0 = 2$  times as fast as M2. b) For program 1: Execution rate on M1 =  $5 \times 10^9 / 2.0 = 2.5 \times 10^9 \text{ IPS}$  (Instructions Per Second). Execution rate on M2 =  $6 \times 10^9 / 1.5 = 4 \times 10^9 \text{ IPS}$ . c) **CPI** = Execution time  $\times$  Clock rate / Instruction Count For program 1: CPI on M1 =  $2.0 \times 3 \times 10^9 / (5 \times 10^9) = 1.2$  cycles per instruction CPI on M2 =  $1.5 \times 4 \times 10^9 / (6 \times 10^9) = 1.25$  cycles per instruction CPU execution time = Instruction Count  $\times$  CPI / Clock rate .CPU execution time =  $7.5 \times 10^9 \times 1.2 / (5 \times 10^9) = 1.8$  seconds    % of CPU time =  $1.8 / 3 = 0.6$  or 60% of the total wall time

**Static and dynamic schedule**=A static schedule is some kind of list containing the order of the processes and the durations in which they are scheduled. For example, a simple communicating device could have the following static schedule in pseudo-code:

Code: repeat forever:

execute task 1 for 10 ms

execute task 2 for 20 ms

execute task 1 for 5 ms

execute task 3 for 15 ms..

This is a static schedule. It's just like the bus schedule: Task three is executed for 15 ms every 50 ms. The schedule never changes, even if task 1 has nothing to do and task 2 is missing its deadlines. On the other hand, when performing dynamic scheduling, whenever the scheduler decides which task to execute next (and for how long), it looks at a list of tasks requesting the processor at that point in time and then decides which to use next.

Examples are the "earliest-deadline first" scheduler. Here, the schedule changes if some task has nothing to do and does not request resources.

### **Branch and jump condition:**

A branch instruction can be either an unconditional branch, which always results in branching, or a conditional branch, which may or may not cause branching depending on some condition. **Jump** instructions modify the program counter so that execution continues at a specified memory address, no matter (almost) the value of the current program counter. [Branch instructions](#), by contrast, are always relative to the current program counter.

### **Find any values that are not live and compiler can delete:**

Only b and c are live after the code segment, so values a,d,e and f are not needed subsequently. Values a and f are needed to compute b and a possible value of c, so statements 1,2,3,5,6, and 7 are needed because they are part of the b and c chain of calculation. Statement 4 produces a value that does not contribute to the live values and is itself not live, thus this statement may be deleted.

### **Register-Register:**

Advantages: \*Simple, fixed-length instruction decoding \*Simple code generation \*Similar number of clock cycles / instruction =Disadvantages \*Higher Instruction count than memory reference \*Lower instruction density leads to larger programs

\***Register- Memory** =Advantages \*Data can be accessed without separate Load first \*Instruction format is easy to encode=Disadvantages \*Operands are not equivalent since a source operand (in a register) is destroyed in operation \*Encoding a register number and memory address in each instruction may restrict the number of registers CPI vary by operand location

\***Memory- Memory** \* Advantages \*Most compact\* Doesn't waste registers for temporary storages \*Disadvantages \*Large variation in instruction size \*Large variation in work per instruction \*Memory bottleneck by memory access

**Static scheduling** (optimized by compiler) – When there is a stall (hazard) no further issue of instructions – Of course, the stall has to be enforced by the hardware

• **Dynamic scheduling** (enforced by hardware) – Instructions following the one that stalls can issue if they do not produce structural hazards or dependencies.

**Static branch predictor:** The second level of branch prediction in the ARM11 MPCore processor uses static branch prediction that is based solely on the characteristics of a branch instruction. It does not make use of any history information. The scheme used in the ARM11 MPCore processor predicts that all forward conditional branches are not taken and all backward branches are taken. Around 65% of all branches are preceded by enough non-branch cycles to be completely predicted. Branch prediction is performed only when the Z bit in CP15 Register c1 is set to 1. See [c1, Control Register](#) for details of this register.

Dynamic prediction works on the basis of caching the previously seen branches in the BTAC, and like all caches suffers from the compulsory miss that exists on the first encountering of the branch by the predictor. A second, static predictor is added to the design to counter these misses, and to mop-up any capacity and conflict misses in the BTAC. The static predictor amounts to an early evaluation of branches in the pipeline, combined with a predictor based on the direction of the branches to handle the evaluation of condition codes that are not known at the time of the handling of these branches. Only items that have not been predicted in the dynamic predictor are handled by the static predictor. The static branch predictor is hard-wired with backward branches being predicted as taken, and forward branches as not taken. The SBP looks at the MSB of the branch offset to determine the branch direction. Statically predicted taken branches incur a one-cycle delay before the target instructions start refilling the pipeline. The SBP works in both ARM and Thumb states. The SBP does not function in Jazelle state. It can be disabled using CP15 Register c1.

**MOV.S and MOV.D**

MOV.S copies a single precision register to another of the same type MOV.D copies a Double precision register to another of the same type..mov s and mov d both are float point instruction in mips

**Relative Performance**

If computer A runs a program in 10 seconds and computer B runs the same program in 15 seconds, how much faster is A than B?

We know that A is  $n$  times faster than B if

$$\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = n$$

Thus the performance ratio is

$$\frac{15}{10} = 1.5$$

and A is therefore 1.5 times faster than B.