Lecture 4 Notes

Parallel Processing CS471

Branch Instructions Recap

In Lecture 2 we covered the MIPS instruction set:

> The MIPS instruction set is an RISC instruction set developed by MPIS Computer Systems. MIPS uses thirty-two 32-bit general purpose registers and a 32-bit program counter (PC).
>
> *From Lecture 2 notes*

We also looked at some instructions from the MIPS instruction set, which included a set of *branch* instructions.

Instruction Description	Example Usage	Instruction Type	Format	Opcode	Function
Branch on Less Than Zero	`bltz $t0, label`	Control Flow	I	1	0
Branch if Equal	`beq $t0, $t1, label`	Control Flow	I	4	-
Branch if Not Equal	`bne $t0, $t1, label`	Control Flow	I	5	-

Table 1: MIPS branch instructions

These instructions would use *PC-relative addressing* to *branch* execution to a specific instruction:

> **PC-Relative Addressing**
>
> Used for branch instructions like `beq`, `bne`, and `bltz`. The effective address is determined by adding a signed intermediate offset to the current PC value.
>
> For example:
>
> ```
> beq $t0, $t1, L
> ```
>
> This instruction branches to `PC + L` if `$t0 == $t1`.
>
> **Note 1.** *Since the immediate offset is sign-extended (signed) it has to be shifted by two bits making the actual immediate address:* `PC + 4 + (L << 2)`.
>
> *From Lecture 2 notes*

Pipeline Hazards Recap

In Lecture 3 we discussed pipeline hazards:

> **Definition 5** (Pipeline Hazard)**.** *A pipeline hazard is an event during pipeline execution that does not permit further execution, which causes the pipeline (or some portion of the pipeline) to stall.*
>
> There are three types of pipeline hazards: resource, data, and control.
>
> *From Lecture 3 notes*

1 Control Hazards [1, 2]

Definition 1 (Control Hazard)**.** *A control hazard (or branch hazard) occurs when the pipeline brings instructions into the pipeline that must subsequently be discarded.*

Control Hazards like other pipeline hazards, occur due to dependency between instructions, which causes uncertainty during execution.

Control Hazard Example

Consider the following set of instructions:

```
1. add  $1, $0, $0
2. beq  $1, $2, L
3. sub  $3, $1, $4
4. and  $5, $3, $6
5. or   $7, $5, $8

L: add  $9, $9, $9
```

We say that instructions 3, 4, and 5, have a *control dependence* on the branch at instruction 2, meaning that whether they should be executed depends on the result of the branch instruction. Similarly the instruction at L also has a control dependence on instruction 2.

Lets consider the ideal cases for both possibilities:

Instruction	Cycle 1	2	3	4	5	6	7	8	9	10
`add $1, $0, $0`	IF	ID	EX	MEM	WB					
`beq $1, $2, LABEL`		IF	ID	EX	MEM	WB				
`add $9, $9, $9`			IF	ID	EX	MEM	WB			

(a) Branch taken

Instruction	Cycle 1	2	3	4	5	6	7	8	9	10
`add $1, $0, $0`	IF	ID	EX	MEM	WB					
`beq $1, $2, LABEL`		IF	ID	EX	MEM	WB				
`sub $3, $1, $4`			IF	ID	EX	MEM	WB			
`and $5, $3, $6`				IF	ID	EX	MEM	WB		
`or $7, $5, $8`					IF	ID	EX	MEM	WB	

(b) Branch not taken

Figure 1

However, we won't know whether the the branch will be taken or not until the instruction 2 reaches the EX stage in the pipeline.

Lets again consider our two possibilities and examine them for control hazards:

Instruction	Cycle 1	2	3	4	5	6	7	8	9	10	11
`add $1, $0, $0`	IF	ID	EX	MEM	WB						
`beq $1, $2, LABEL`		IF	ID	EX	MEM	WB					
`sub $3, $1, $4`			IF	ID							
`and $5, $3, $6`				IF							
`add $9, $9, $9`					IF	ID	EX	MEM	WB		

(a) Branch taken, causing a control hazard. The branch instruction enters the pipeline in clock cycle 2, but the decision to take the branch is resolved at the end of clock cycle 4. We have incorrectly fetched two instructions, and now must discard them.

Instruction	Cycle 1	2	3	4	5	6	7	8	9	10	11
`add $1, $0, $0`	IF	ID	EX	MEM	WB						
`beq $1, $2, LABEL`		IF	ID	EX	MEM	WB					
`sub $3, $1, $4`			IF	ID	EX	MEM	WB				
`and $5, $3, $6`				IF	ID	EX	MEM	WB			
`or $7, $5, $8`					IF	ID	EX	MEM	WB		

(b) Branch not taken, no control hazard occurs.

Figure 2

We can calculate the *penalty* of the control hazard in the above example by comparing the ideal case in Figure 1a and the case in Figure 2a. In 1a execution completes at clock cycle 8, while in 2a execution completes at clock cycle 10. $10 - 8 = 2$ so we say the branch has a penalty of two clock cycles.

Control Hazard Effect on CPI

How much of a penalty a control hazard incurs depends on the depth of the pipeline and at which stage branches are resolved.

We start with the ideal CPI of 1, since penalties are only incurred when branches are taken, we can calculate the CPI as follows:

$$\text{CPI} = 1 + \left(\frac{\text{Number of branch instructions}}{\text{Total number of instructions (IC)}} \times \Pr(\text{Branch is taken}) \times \text{Branch penalty} \right), \quad (1)$$

where the branch penalty = stage at which the pipeline resolves branches - 1.

For example:

Consider a 10-stage pipeline, that resolves branches in the 5th stage of its execution. If we were to execute a program with $\approx 20\%$ branch instructions, and those branch instructions are taken $\approx 50\%$ of the time. We could calculate the CPI for this case as follows:

$$\text{CPI} = 1 + (0.2 \times 0.5 \times 5) = 1.5$$

1.1 Handling Control Hazards

Resolving Branches at an Earlier Stage

One way to reduce the penalty/cost of control hazards is to resolve branches at an earlier stage in pipeline execution, this would a modification to the hardware of the pipeline. In a standard MIPS pipeline the circuitry for handling branches is in the third (EX) stage:

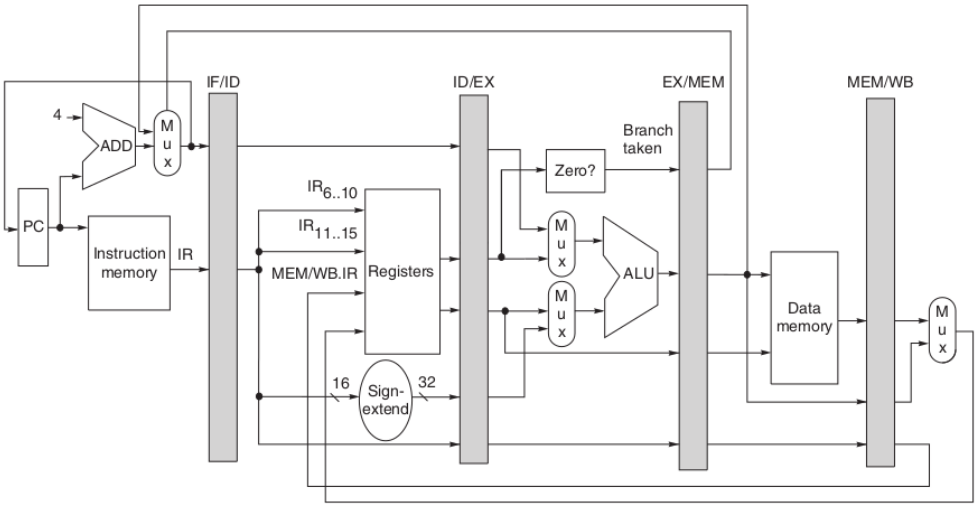

Figure 3: Standard MIPS 5-stage pipeline.

We can reduce the penalty of cost hazards by moving the zero test and branch-target calculation into the ID phase of the pipeline:

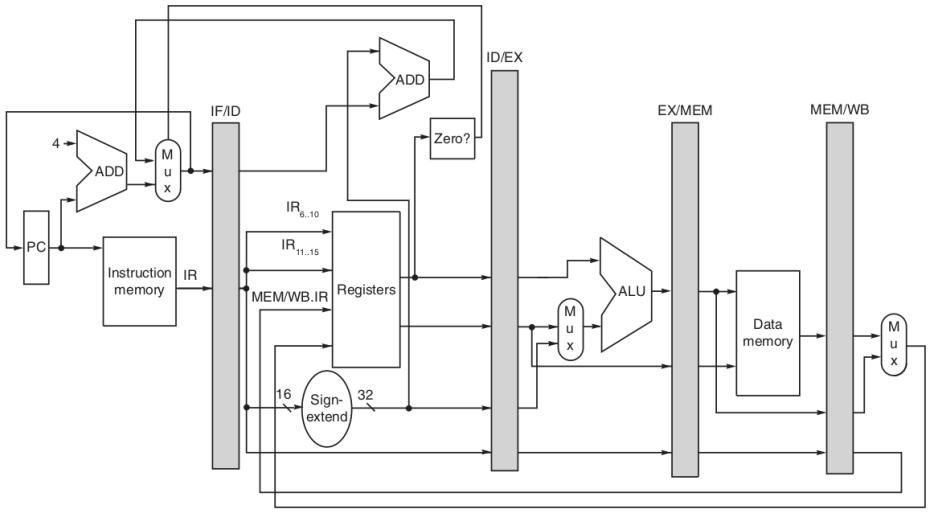

Figure 4: Modified MIPS pipeline. In the modified pipeline branch instructions are resolved in the second (ID) stage.

Resolving branch instructions in the second (ID) stage reduces the penalty for branches from two clock cycles to only one.

Delay Slot

Definition 2 (Delay Slot)**.** *A delay slot s an instruction slot immediately following a branch instruction in the pipeline that is always executed, regardless of whether the branch is taken or not.*

Delay slots eliminate the potential of a control hazard, they rely on the compiler to place a useful instruction in the slot. This way regardless of branch resolution we don't discard any instructions that are already in the pipeline.

Note 1. *If there isn't a useful instruction to place in the stall a* `nop` *'dummy' instruction is placed in the slot.*

Delay Slot Example

Consider the following instructions:

```
1. beq  $1, $2, L
2. add  $4, $3, $3
3. and  $5, $3, $6

L: add  $7, $1, $2
```

Instruction	Cycle 1	2	3	4	5	6	7	8	9
`beq $1, $2, L`	IF	ID	EX	MEM	WB				
`add $4, $3, $3`		IF	ID	EX	MEM	WB			
`and $5, $3, $6`			IF						
`add $7, $1, $2`				IF	ID	EX	MEM	WB	

(a) Effect on execution using a single delay slot.

Instruction	Cycle 1	2	3	4	5	6	7	8	9
`beq $1, $2, L`	IF	ID	EX	MEM	WB				
`add $4, $3, $3`		IF	ID	EX	MEM	WB			
`and $5, $3, $6`			IF	ID	EX	MEM	WB		
`add $7, $1, $2`				IF	ID	EX	MEM	WB	

(b) Effect on execution using two delay slots.

Figure 5

In the example above we assume that instructions 2 and 3 are useful instructions, in the case that they aren't they would be replaced by dummy instructions.

Note 2. *The original MIPS architecture contained a single delay slot.*

Branch Prediction

References

[1] John L. Hennessy and David A. Patterson. Appendix c. In *Computer Architecture*, The Morgan Kaufmann Series in Computer Architecture and Design, pages 648–729. Morgan Kaufmann, Oxford, England, 5th edition, sep 2011.

[2] William Stallings. Processor structure and function. In *Computer Organization and Architecture*, chapter 12, pages 432–476. Pearson, Upper Saddle River, NJ, 8 edition, apr 2009.