

Advanced Computer Architecture

Lecture No. 20

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 5
5.1.5, 5.1.6

Summary

- Structural RTL for Pipeline Stages
- Instruction Propagation Through the Pipeline
- Pipeline Hazards
- Data Dependence Distance
- Data Forwarding
- Compiler Solution to Hazards
- SRC Hazard Detection and Correction
- RTL for Hazard Detection and Pipeline Stall

Structural RTL for Pipeline Stages

The Register Transfer Language for each phase is given as follows:

Instruction Fetch

$$\begin{aligned} \text{IR2} &\leftarrow \text{M}[\text{PC}]; \\ \text{PC2} &\leftarrow \text{PC}+4; \end{aligned}$$

Instruction Decode & Operand fetch

$$\begin{aligned} \text{X3} &\leftarrow \text{l-s2:}(\text{rel2:PC2,disp2:}(\text{rb}=0):?,(\text{rb}\neq 0):\text{R}[\text{rb}]),\text{brl2:PC2,alu2:R}[\text{rb}], \\ \text{Y3} &\leftarrow \text{l-s2:}(\text{rel2:c1,disp2:c2),alu2:}(\text{imm2:c2,!imm2:R}[\text{rc}]), \\ \text{MD3} &\leftarrow \text{store2:R}[\text{ra}], \text{IR3} \leftarrow \text{IR2,stop2:Run} \leftarrow 0, \\ \text{PC} &\leftarrow \text{!branch2:PC+4,branch2:}(\text{cond}(\text{IR2,R}[\text{rc}]):\text{R}[\text{rb}],!\text{cond}(\text{IR2,R}[\text{rc}]):\text{PC+4}; \end{aligned}$$

ALU operation

$$\begin{aligned} \text{Z4} &\leftarrow (\text{l-s3: X3+Y3, brl3: X3, Alu3: X3 op Y3,} \\ \text{MD4} &\leftarrow \text{MD3,} \\ \text{IR4} &\leftarrow \text{IR3;} \end{aligned}$$

Memory access

$$\begin{aligned} \text{Z5} &\leftarrow (\text{load4: M}[\text{Z4}], \text{ladr4}\sim\text{branch4}\sim\text{alu4:Z4}), \\ \text{store4: (M}[\text{Z4}] &\leftarrow \text{MD4),} \end{aligned}$$

IR5 $\leftarrow$ IR4;

Write back

regwrite5: (R[ra] $\leftarrow$ Z5);

Instruction Propagation through the Pipeline

Consider the following SRC code segment flowing through the pipeline. The instructions along with their addresses are

```
200: add r1, r2, r3
204: ld r5, [4(r7)
208: br r6
212: str r4, 56
...
400
```

We shall review how this chunk of code is executed.

First Clock Cycle

Add instruction enters the pipeline in the first cycle. The value in PC is incremented from 200 to 204.

Second Clock Cycle

Add moves to decode stage. Its operands are fetched from the register file and moved to X3 and Y3 at the end of clock cycle, meanwhile the Instruction ld r5, [4+r7] is fetched in the first stage and the PC value is incremented from 204 to 208.

Third Clock Cycle

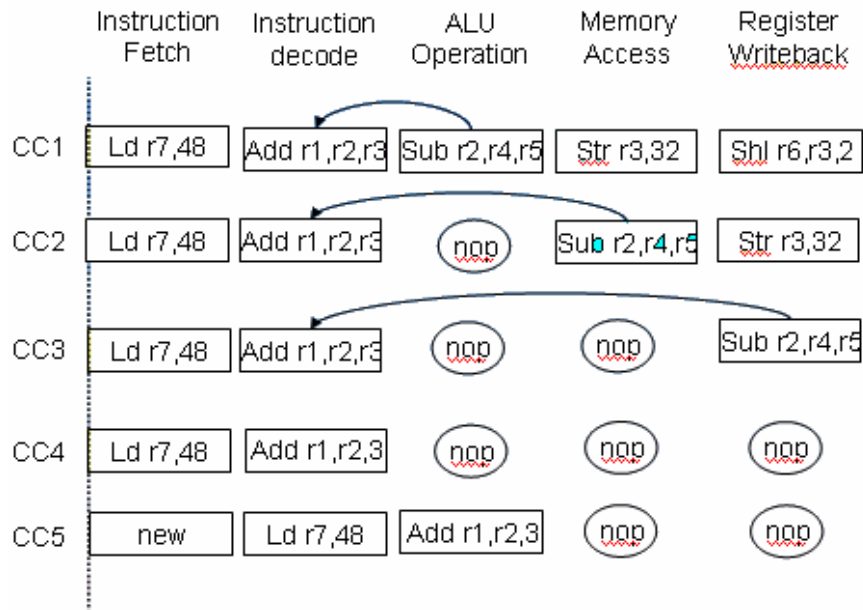
Add instruction moves to the execute stage, the results are written to Z4 on the trailing edge of the clock. Ld instruction moves to decode stage. The operands are fetched to calculate the displacement address. Br instruction enters the pipeline. The value in PC is incremented from 208 to 212.

Fourth Clock Cycle

Add does not access memory. The result is written to Z5 at the trailing edge of clock. The address is being calculated here for ld. The results are written to Z4. Br is in the decode stage. Since this branch is always true, the contents of PC are modified to new address. Str instruction enters the pipeline. The value in PC is incremented from 212 to 216.

Fifth Clock Cycle

The result of addition is written into register r1. Add instruction completes. Ld accesses data memory at the address specified in Z4 and result stored in Z5 at falling edge of clock. Br instruction just propagates through this stage without any calculation. Str is in the decode stage. The operands are being fetched for address calculation to X3 and Y3. The instruction at address 400 enters the pipeline. The value in PC is incremented from 400 to 404.



Pipeline Hazards

The instructions in the pipeline at any given time are being executed in parallel. This parallel execution leads to the problem of instruction dependence. A hazard occurs when an instruction depends on the result of previous instruction that is not yet complete.

Classification of Hazards

There are three categories of hazards

1. Branch Hazard
2. Structural Hazard
3. Data Hazard

Branch hazards

The instruction following a branch is always executed whether or not the branch is taken. This is called the branch delay slot. The compiler might issue a nop instruction in the branch delay slot. Branch delays cannot be avoided by forwarding schemes.

Structural hazards

A structural hazard occurs when attempting to access the same resource in different ways at the same time. It occurs when the hardware is not enough to implement pipelining properly e.g. when the machine does not support separate data and instruction memories.

Data hazards

Data hazard occur when an instruction attempts to access some data value that has not yet been updated by the previous instruction. An example of this RAW (read after write) data hazard is;

```
200: add r2, r3, r4
```

```
204: sub r7, r2, r6
```

The register r2 is written in clock cycle 5 hence the sub instruction cannot proceed beyond stage 2 until the add instruction leaves the pipeline.

Data Hazard Detection & Correction

Data hazards can be detected easily as they occur when the destination register of an instruction is the same as the source register of another instruction in close proximity. To remedy this situation, dependent instructions may be delayed or stalled until the ones ahead complete. Data can also be forwarded to the next instruction before the current instruction completes, however this requires forwarding hardware and logic. Data can be forwarded to the next instruction from the stage where it is available without waiting for the completion of the instruction. Data is normally required at stage 2 (operand fetch) however data is earliest available at stage 3 output (ALU result) or stage 4 output (memory access result). Hence the forwarding logic should be able to transfer data from stage 3 to stage 2 or from stage 4 to stage 2.

Data Dependence Distance

Designing a data forwarding unit requires the study of dependence distances. Without forwarding, the minimum spacing required between two data dependent instructions to avoid hazard is four. The load instruction has a minimum distance of two from all other instructions except branch. Branch delays cannot be removed even with forwarding.

Table 5.1 of the text shows numbers related to dependence distances with respect to some important instruction categories.

Compiler Solution to Hazards

Hazards can be detected by the compiler, by analyzing the instruction sequences and dependencies. The compiler can insert bubbles (nop instruction) between two instructions that form a hazard, or it could reorder instructions so as to put sufficient distance between dependent instructions. The compiler solution to hazards is complex, expensive and not very efficient as compared to the hardware solution

SRC Hazard Detection and Correction

The SRC uses a hazard detection unit. The hazard can be resolved using either pipeline stalls or by data forwarding.

Pipeline stalls

Consider the following sequence of instructions going through the SRC pipeline

200: shl r6, r3, 2

204: str r3, 32

208: sub r2, r4, r5

212: add r1, r2, r3

216: ld r7, 48

There is a data hazard between instruction three and four that can be resolved by using pipeline stalls or bubbles

When using pipeline stalls, nop instructions are placed in between dependent instructions. The logic behind this scheme is that if opcode in stage 2 and 3 are both alu, and if ra in stage 3 is the same as rb or rc in stage 2, then a pause signal is issued to insert a bubble between stage 3 and 2. Similar logic is used for detecting hazards between stage 2 and 4 and stage 4 and 5.

Data Forwarding

By adding data forwarding mechanism to the SRC data path, the stalls can be completely eliminated at least for the ALU instructions. The hazard detection is required between stages 3 and 4, and between stages 3 and 5. The testing and forwarding circuits employ wider IRs to store the data required in later stages. The logic behind this method is that if the ALU is activated for both 3 and 5 and ra in 5 is the same as rb in 3 then Z5 which hold the currently loaded or calculated result is directly forwarded to X3. Similarly, if both are ALU operations and instruction in stage 3 does not employ immediate operands then value of Z5 is transferred to Y3. Similar logic is used to forward data between stage 3 and 4.

RTL for Hazard Detection and Pipeline Stall

The following RTL expression detects data hazard between stage 2 and 3, then stalls stage 1 and 2 by inserting a bubble in stage 3

$$\text{alu3} \& \text{alu2} \& ((\text{ra3} = \text{rb2}) \sim ((\text{ra3} = \text{rc2}) \& !\text{imm2})): \\ (\text{pause2}, \text{pause1}, \text{op3} \leftarrow 0)$$

Meaning:

If opcode in stage 2 and 3 are both ALU, and if ra in stage 3 is same as rb or rc in stage 2, issue a pause signal to insert a bubble between stage 3 and 2

Advanced Computer Architecture-CS501

Following is the complete RTL for detecting hazards among ALU instructions in different stages of the pipeline

Data Hazard between	RTL for detection and stalling
Stage 2 and 3	$\text{alu3} \& \text{alu2} \& ((\text{ra3} = \text{rb2}) \sim ((\text{ra3} = \text{rc2}) \& !\text{imm2})) :$ $(\text{pause2}, \text{pause1}, \text{op3} \leftarrow 0)$
Stage 2 and 4	$\text{alu4} \& \text{alu2} \& ((\text{ra4} = \text{rb2}) \sim ((\text{ra4} = \text{rc2}) \& !\text{imm2})) :$ $(\text{pause2}, \text{pause1}, \text{op3} \leftarrow 0)$
Stage 2 and 5	$\text{alu5} \& \text{alu2} \& ((\text{ra5} = \text{rb2}) \sim ((\text{ra5} = \text{rc2}) \& !\text{imm2})) :$ $(\text{pause2}, \text{pause1}, \text{op3} \leftarrow 0)$