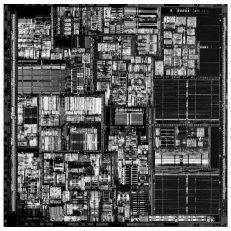
EEL 4713/5764: Computer Architecture

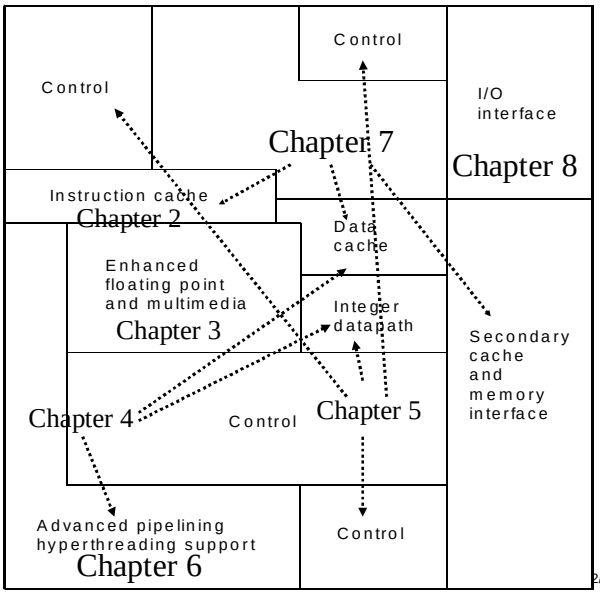
Chapter 6:
Enhancing Performance with Pipelining

Dr. Ming Yu
Dept. of Electrical & Computer Engineering
FAMU-FSU College of Engineering

11/13/2008

The Pentium 4 For Our Chapters





2/119

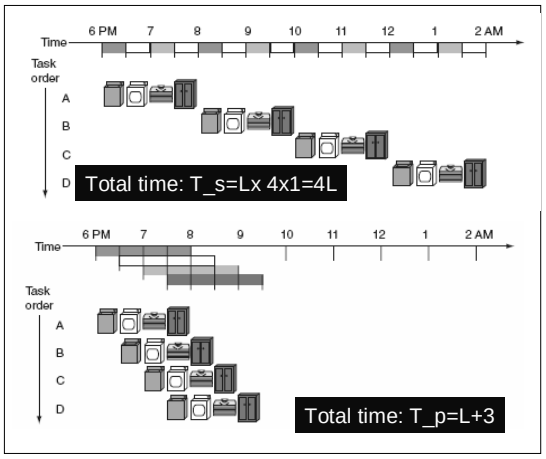
Table of Contents

1. An Overview of Pipelining
2. A Pipelined Datapath
3. Pipelined Control
4. Data Hazards and Forwarding
5. Data Hazards and Stalls
6. Branch Hazards
7. Using a Hardware Description Language to Describe and Model a Pipeline
8. Exceptions
9. Advanced Pipelining: Extracting More Performance
10. Real Stuff: The Pentium 4 Pipeline
11. Concluding Remarks

6.1 An Overview of Pipelining: An Example

- The time from placing a single dirty sock in the washer until it is dried, folded, and put away is not shorter for pipelining;
  - the reason pipelining is faster for many loads is that everything is working in parallel, so more loads are finished per hour.
- **Pipelining improves throughput** of our laundry system **without improving the time to complete a single load**.
  - Hence, pipelining would not decrease the time to complete one load of laundry,
  - but when we have many loads of laundry to do, the improvement in throughput decreases the total time to complete the work.

Fig. 6.1: The laundry analogy for pipelining



- A, B, C, and D each has dirty clothes to be washed, dried, folded, and put away.
- The washer, dryer, folder, and storer each takes 30 minutes for its task.
- Sequential laundry takes 8 hours for four loads of wash, while pipelined laundry takes just 3.5 hours.

- We show the pipeline stage of different loads over time
  - by showing copies of the four resources on this two-dimensional time line
  - but we really have just one of each resource!

5/119

Solution to the Example

- Assume
  - the time for processing a load in serial is T_s;
  - the time for a load in pipelining is T_p;
  - there are a total of L loads;
  - each load has **4** steps to do the job;
  - each step costs one unit of time.

- Solution:
- In serial: T_s=Lx 4x1=4L;
- In pipeline: T_p=L+3;

$$\lim_{L \to +\infty} \frac{T_s}{T_p} = \lim_{L \to +\infty} \frac{4L}{L+3} = 4$$

6/119

Pipeline Instruction Execution

- MIPS instructions classically take five steps:

1. Fetch instruction from memory
2. Read registers while decoding the instruction
   - The format of MIPS instructions allows reading and decoding to occur simultaneously
3. Execute the operation or calculate an address
4. Access an operand in data memory
5. Write the result into a register

Hence, the MIPS pipeline we explore in this chapter has five stages.

Example: p. 372

To make this discussion concrete, we limit our attention to eight instructions: **`lw, sw, add , sub, and, or, slt, beq`**.

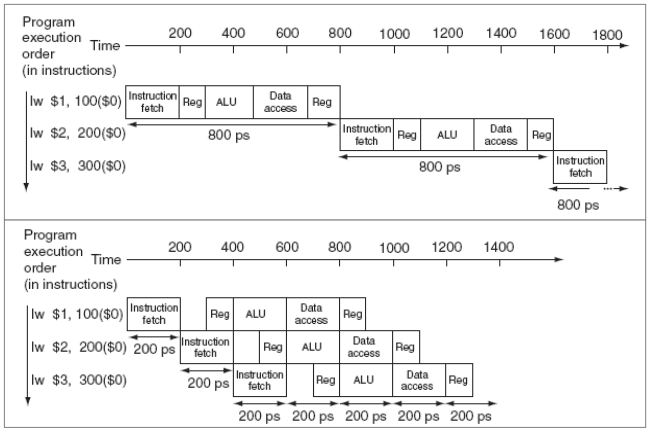
- **Problem**: compare the average time between instructions of a single-cycle implementation, in which all instructions take 1 clock cycle, to a pipelined implementation.
- **Answer**:
  - In the single-cycle model, every instruction takes exactly 1 clock cycle, so the clock cycle must be stretched to accommodate the slowest instruction.
  - The single-cycle design must allow for the slowest instruction — in Figure 6.2 it is `lw` — so the time required for every instruction is 800 ps.
  - All the pipeline stages take a single clock cycle, so the clock cycle must be long enough to accommodate the slowest operation.
  - the pipelined execution clock cycle must have the worst-case clock cycle of 200 ps even though some stages take only 100ps.

Fig. 6.2:

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

- Fig. 6.2: Total time for each instruction calculated from the time for each component.
- This calculation assumes that the multiplexors, control unit, PC accesses, and sign extension unit have no delay.

Fig. 6.3: Single-cycle, nonpipelined execution vs. pipelined execution



- Both use the same hardware components, whose time is listed in Figure 6.2.
  - A fourfold speedup on **average time between instructions**: from 800 ps down to 200 ps.
- Compare this figure to Figure 6.1: For the laundry, we assumed all stages were equal. If the dryer were slowest, then the dryer stage would set the stage time.
  - The computer pipeline stage times are limited by the slowest resource, either the ALU operation or the memory access.
- We assume that:
  - **the write to the register file occurs in the first half of the clock cycle and**
  - **the read from the register file occurs in the second half**

ISA Design Principle

- First, all MIPS instructions are of the same length.
  - all recent implementations of the IA-32 actually translate IA-32 instructions into simple micro-operations that look like MIPS instructions
  - the Pentium 4 actually pipelines the micro-operations rather than the native IA-32 instructions!
- Second, MIPS has only a few instruction formats, with the source register fields being located in the same place in each instruction.
  - The second stage can begin reading the register file at the same time that the hardware is determining what type of instruction was fetched.
- Third, memory operands only appear in loads or stores in MIPS.
- Fourth, as discussed in Chapter 2, operands must be aligned in memory.
  - we need not worry about a single data transfer instruction requiring two data memory accesses;
  - the requested data can be transferred between processor and memory in a single pipeline stage.

Designing instruction sets for pipelining

- **Hazards:**
  - the situations in pipelining when the next instruction cannot execute in the following clock cycle
- **Structural hazard:**
  - An occurrence in which a planned instruction cannot execute in the proper clock cycle
    - because the hardware cannot support the combination of instructions that are set to execute in the given clock cycle.
- **Data hazard:**
  - Also called **pipeline data hazard**
  - An occurrence in which a planned instruction cannot execute in the proper clock cycle
    - because the data that is needed to execute the instruction is not yet available
- **Forwarding:**
  - Also called **bypassing**
  - A method of resolving a data hazard by retrieving the missing data element from internal buffers
    - rather than waiting for it to arrive from programmer-visible registers or memory
  - The name forwarding comes from the idea that the result is passed forward from an earlier instruction to a later instruction
  - Bypassing comes from passing the result by the register file to the desired unit

Data Hazard and Forwarding

- In a computer pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline

```
add    $s0, $t0, $t1
sub    $t2, $s0, $t3
```

- Without intervention, a data hazard could severely stall the pipeline.
  - The **add** instruction doesn't write its result until the fifth stage,
  - meaning that we would have to add three **bubbles** to the pipeline.
- As soon as the ALU creates the sum for the **add**, we can supply it as an input for the subtract.
- Adding extra hardware to retrieve the missing item early from the internal resources is called forwarding or bypassing .

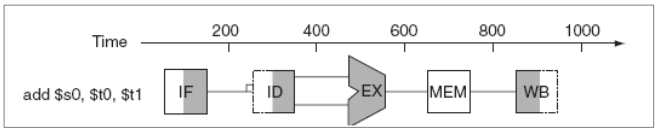
13/119

Hazard, Stall, and Bubble

- Load-use data hazard:
  - A specific form of data hazard
  - in which the data requested by a load instruction has not yet become available when it is requested.
- Pipeline stall:
  - Also called **bubble**
  - A stall initiated in order to resolve a hazard
- Control hazard:
  - Also called branch hazard
  - An occurrence in which the proper instruction cannot execute in the proper clock cycle
    - because the instruction that was fetched is not the one that is needed
    - that is, the flow of instruction addresses is not what the pipeline expected.

14/119

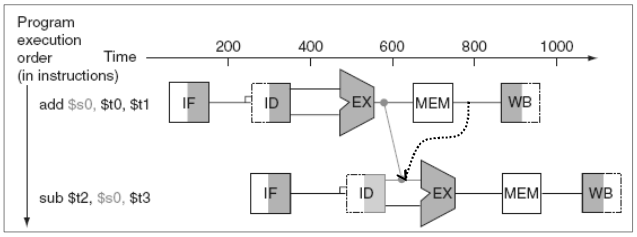
Fig. 6.4: Graphical representation of the instruction pipeline



- The symbols for the five stages:
  - **IF: for the instruction fetch stage**
    - with the box representing instruction memory
  - **ID: for the instruction decode/register file read stage**
    - with the drawing showing the register file being read
  - **EX: for the execution stage**
    - with the drawing representing the ALU
  - **MEM: for the memory access stage**
    - with the box representing data memory
  - **WB: for the write back stage**
    - with the drawing showing the register file being written
- The shading indicates the element is used by the instruction:
  - MEM has a white background: because add does not access the data memory.
  - shading on the right half of a reg file/mem: it is reading in that stage
  - shading on the left half: it is writing in that stage
  - Examples:
    - the right half of ID is shaded: because the register file is read.
    - the left half of WB is shaded: because the register file is written.

15/119

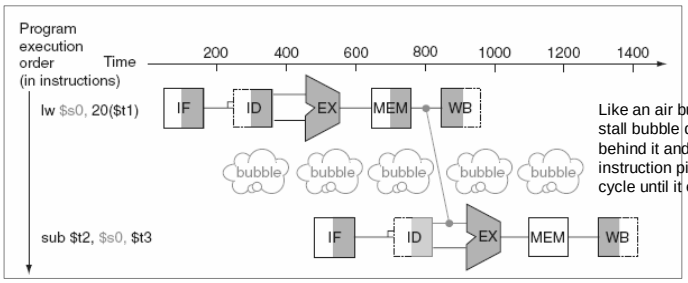
Example: p. 376



- Fig. 6.5: Graphical representation of **forwarding**
- The connection shows the forwarding path
  - from the output of the EX stage for `add`
  - to the input of the EX stage for `sub`
  - replacing the value from register `$s0` read in the 2nd stage of `sub`

16/119

Fig. 6.6: add a stall with forwarding



Like an air bubble in a water pipe, a stall bubble delays everything behind it and proceeds down the instruction pipe one stage each cycle until it exits at the end.

- Need a stall even with forwarding when an R-format instruction following a load tries to use the data
  - Without the stall, the path from memory access stage output to execution stage input would be going backwards in time, which is impossible (in time sequence).
  - This figure is actually a simplification, since we cannot know until
    - after the subtract instruction is fetched
    - and decoded whether or not a stall will be necessary
- Section 6.5 shows the details of what really happens in the case of a hazard.

Example: p. 378

- Reordering code to avoid pipeline stalls

  Consider the following code segment in C:

  ```
  A = B + E;
  C = B + F;
  ```

  Here is the generated MIPS code for this segment, assuming all variables are in memory and are addressable as offsets from $t0:

  ```
  lw  $t1, 0($t0)   #
  lw  $t2, 4($t0)   #
  add $t3, $t1, $t2 #
  sw  $t3, 12($t0)  #
  lw  $t4, 8($t0)   #
  add $t5, $t1, $t4 #
  sw  $t5, 16($t0)  #
  ```

- Problem: find the hazards in the following code segment and reorder the instructions to avoid any pipeline stalls.

Answer: p. 379

- Both `add` instructions have a hazard because of their respective dependence on the immediately preceding `lw` instruction. Moving up the third `lw` instruction eliminates both hazards:

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
lw  $t4, 8($t0)
add $t3, $t1,$t2
sw  $t3, 12($t0)
add $t5, $t1,$t4
sw  $t5, 16($t0)
```

- On a pipelined processor with forwarding, the reordered sequence will complete in two fewer cycles than the original version.
- Each MIPS instruction writes at most one result and does so near the end of the pipeline.
  - Forwarding is harder if there are multiple results to forward per instruction or they need to write a result early on in instruction execution.

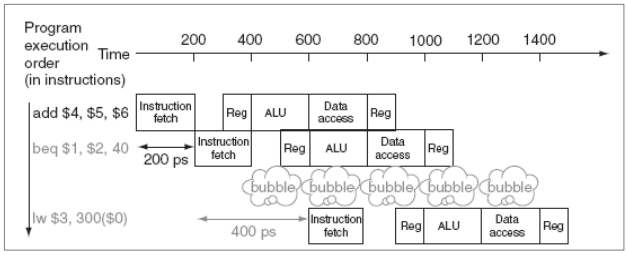
Control Hazard

- A control hazard arises from the need to make a decision based on the results of one instruction while others are executing.
- Why we have control hazard?
  - Does not want to follow the instruction sequence, in stead, the program wants to branch to somewhere else!
- For example:
  - In order to maintain a pipeline, we need to start to run a next instruction after every stage time
  - But for branch instruction, we need to wait several stage time, then we know where is the next instruction.
  - Not enough time to make the decision!

Example: p. 380

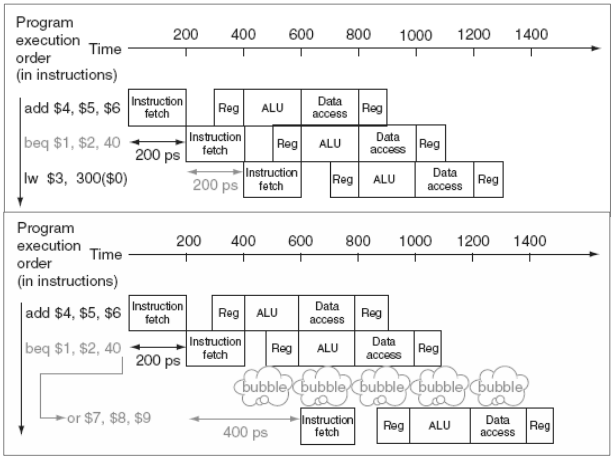
- Performance of “Stall on Branch”
- Estimate the impact on the clock cycles per instruction (CPI) of stalling on branches. Assume all other instructions have a CPI of 1.
- See the next two figures:

Fig. 6.7: Pipeline showing stalling



- Fig. 6.7: Pipeline showing stalling on every conditional branch as solution to control hazards.
- There is a one-stage pipeline stall, or bubble, after the branch.
  - In reality, the process of creating a stall is slightly more complicated
    - as we will see in Section 6.6
  - The effect on performance, however, is the same as would occur if a bubble were inserted.

Fig. 6.8: Predicting that branches are not taken as a solution to control hazard



- The top drawing shows the pipeline when the branch is not taken.
- The bottom drawing shows the pipeline when the branch is taken.
- As noted in Fig. 6.7, the insertion of a bubble in this fashion simplifies what actually happens, at least during the first clock cycle immediately following the branch.

Solutions to Control Hazards

- Three solutions to control hazards:
  - **Stall**: Just operate sequentially until the first batch is dry and then repeat until you have the right formula. This conservative option certainly works, but it is slow.
  - **Predict**: use prediction to handle branches. One simple approach is to always predict that branches will be untaken.
    - When you're right, the pipeline proceeds at full speed.
    - Only when branches are taken does the pipeline stall.
  - **Delayed Decision**: always executes the next sequential instruction, with the branch taking place after that one instruction delay (in MIPS).
    - Since delayed branches are useful when the branches are short, no processor uses a delayed branch of more than 1 cycle.
    - For longer branch delays, hardware-based branch prediction is usually used.
- Branch Prediction: a method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.
- Taken or untaken:
  - Untaken branch: the one that falls through to the successive instruction.
  - A taken branch: the one that causes transfer to the branch target.

Pipeline Overview Summary

- Latency (pipeline): the number of stages in a pipeline or the number of stages between two instructions during execution.
- Pipelining increases the number of simultaneously executing instructions and the rate at which instructions are started and completed.
- Pipelining does not reduce the time it takes to complete an individual instruction, also called the latency.
  - For example, the five-stage pipeline still takes 5 clock cycles for the instruction to complete.
- Pipelining improves instruction throughput rather than individual instruction execution time or latency.
  - Instruction sets can either simplify or make life harder for pipeline designers, who must already cope with structural, control, and data hazards.
- Branch prediction, forwarding, and stalls help make a computer fast while still getting the right answers.

Understanding Program Performance

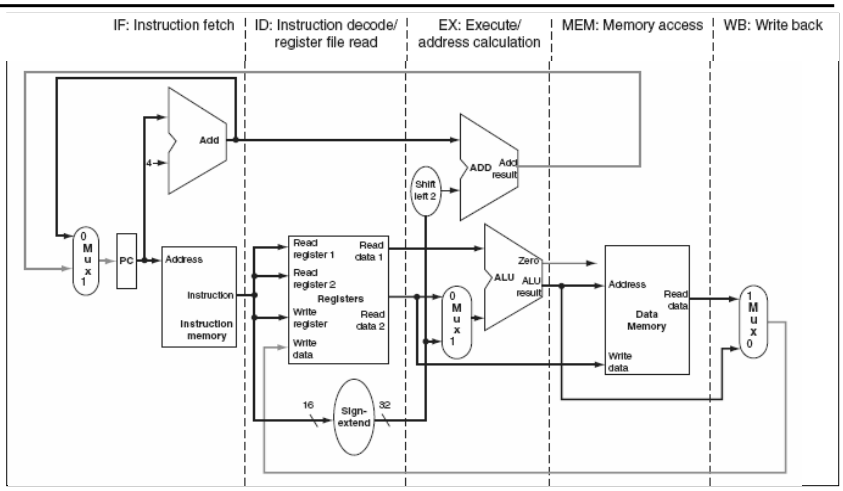
- Structural hazards usually revolve around the floating-point unit, which may not be fully pipelined,
  - while control hazards are usually more of a problem in integer programs, which tend to have higher branch frequencies as well as less predictable branches.
- Data hazards can be performance bottlenecks in both integer and floating-point programs.
- Comparison:
  - Often it is easier to deal with data hazards in floating-point programs because the lower branch frequency and more regular access patterns allow the compiler to try to schedule instructions to avoid hazards.
  - It is more difficult to perform such optimizations in integer programs that have less regular access involving more use of pointers.
- There are more ambitious compiler and hardware techniques for reducing data dependences through scheduling.

6.2: A Pipelined Datapath

- Breaking an instruction into five stages means a five-stage pipeline, which in turn means that up to five instructions will be in execution during any single clock cycle.
- Five stages:
  1. **IF**: Instruction fetch
  2. **ID**: Instruction decode and register file read
  3. **EX**: Execution or address calculation
  4. **MEM**: Data memory access
  5. **WB**: Write back

27/119

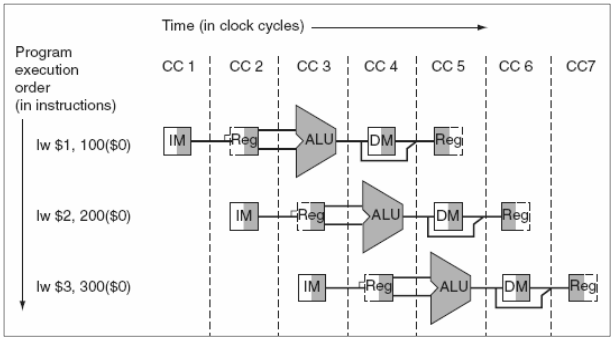
Fig. 6.9: The single-cycle datapath from Chapter 5



- Each step of the instruction can be mapped onto the datapath from left to right
  - The two exceptions are the update of the PC and the write-back step, shown in color:
    - which sends either the ALU result or the data from memory to the left to be written into the register file
- Note: normally we use color lines for control, but these are data lines!

28/119

Fig. 6.10: Instructions being executed using the single-cycle datapath in Fig. 6.9, (assuming **pipelined execution**).



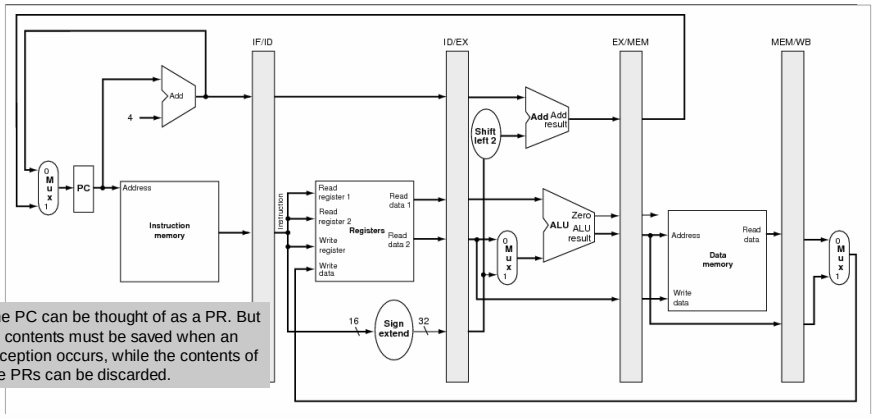
- **IM:** the instruction memory and the PC in the IF stage
- **Reg:** the register file and sign extender in the ID stage, and so on.

Assume that:
- the register file is written in the first half of the clock cycle
- the register file is read during the second half

- Similar to Figures 6.4 through 6.6, this figure pretends that
  - each instruction has its own datapath, and shades each portion according to use.
- Unlike those figures, each stage is labeled by the physical resource used in that stage
  - corresponding to the portions of the datapath in Fig. 6.9
- To maintain proper time order, this stylized datapath breaks the register file into two logical parts:
  1. registers read during register fetch (ID)
     - by drawing the unshaded left half in dashed lines, when it is not being written,
  2. registers written during write back (WB).
     - by the unshaded right half in dashed lines, when it is not being read.

29/119

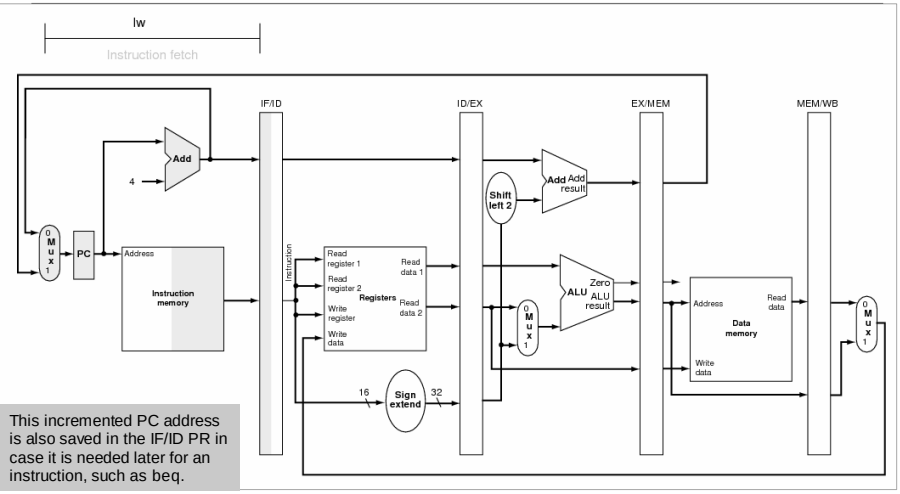
Fig. 6.11: The pipelined version of the datapath in Fig. 6.9



The PC can be thought of as a PR. But its contents must be saved when an exception occurs, while the contents of the PRs can be discarded.

- **The pipeline registers**, in color, separate each pipeline stage.
  - They are labeled by the stages that they separate;
  - Example: the first is labeled IF/ID because it separates the IF and ID stages.
- The registers must be wide enough
  - to store all the data corresponding to the lines that go through them
  - Example: the IF/ID register must be 64bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address.
- We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 128, 97, and 64 bits, respectively.

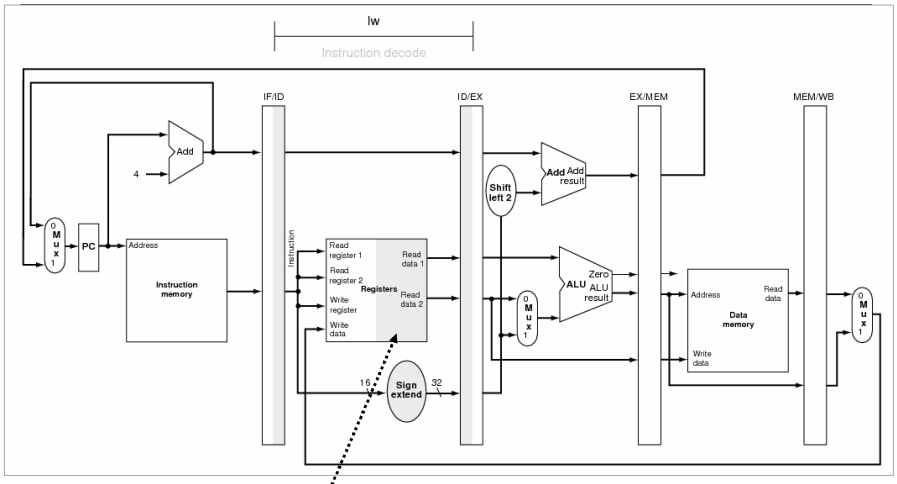
Fig. 6.12 (a): IF and ID: the first pipe stage of an instruction



- With the active portions of the datapath in Fig. 6.11 highlighted, the same convention as that used in Figure 6.4.
- As in Ch 5, no confusion when reading and writing registers because the contents change only on the clock edge.

31/119

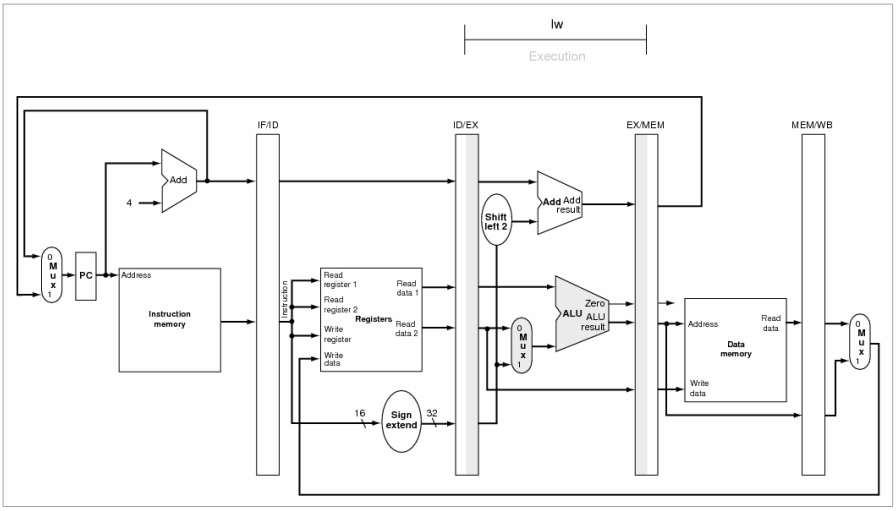
Fig. 6.12 (b): IF and ID: the second pipe stage of an instruction



- Although the load needs only **the top register in stage 2**, the processor doesn't know what instruction is being decoded, so it sign-extends the 16-bit constant and reads both registers into the ID/EX pipeline register.
- We don't need all three operands, but it simplifies control to keep all three.

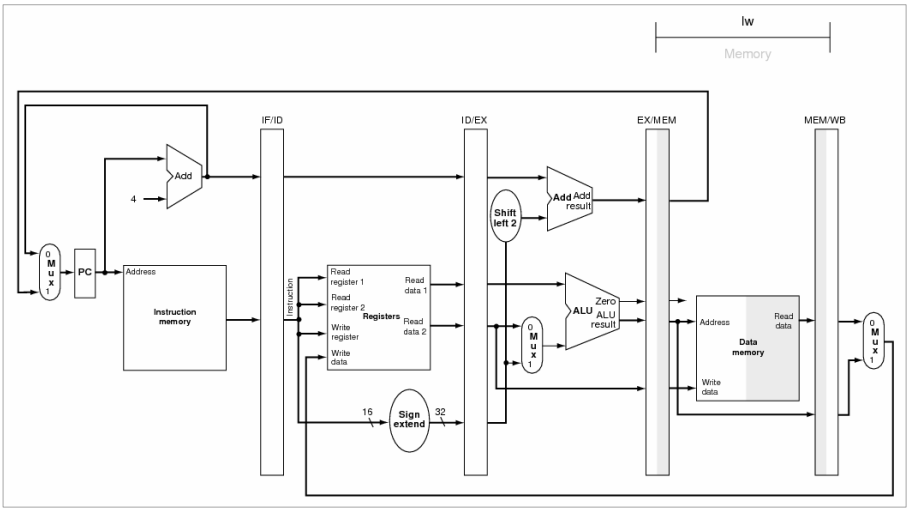
32/119

Fig. 6.13: EX: the third pipe stage of a **load** instruction



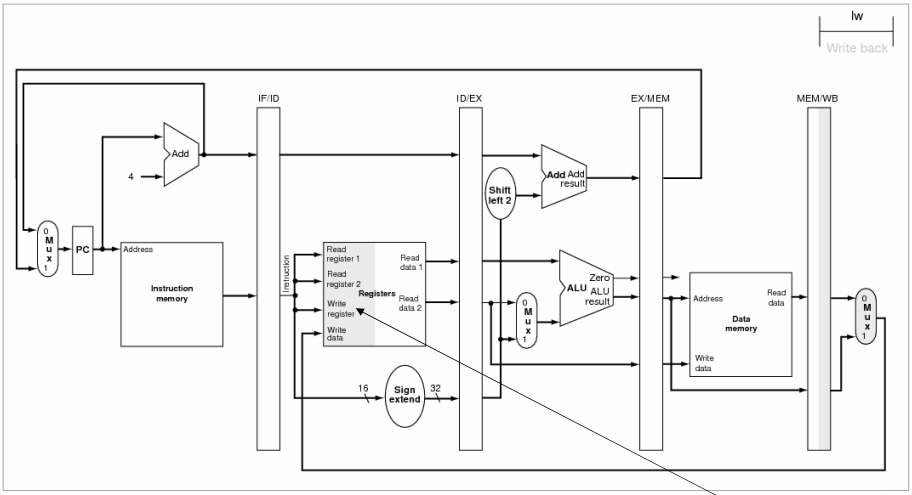
- Highlighting the portions of the datapath in Fig. 6.11 used in this pipe stage:
  - the register is added to the sign-extended immediate
  - and the sum is placed in the EX/MEM pipeline register

Fig. 6.14 (a): MEM and WB: the 4th and 5th pipe stages of a **load** instruction



- Highlighting the portions of the datapath in Fig. 6.11 used in this pipe stage.
- Data memory is read using the address in the EX/MEM pipeline registers, and the data is placed in the MEM/WB pipeline register.

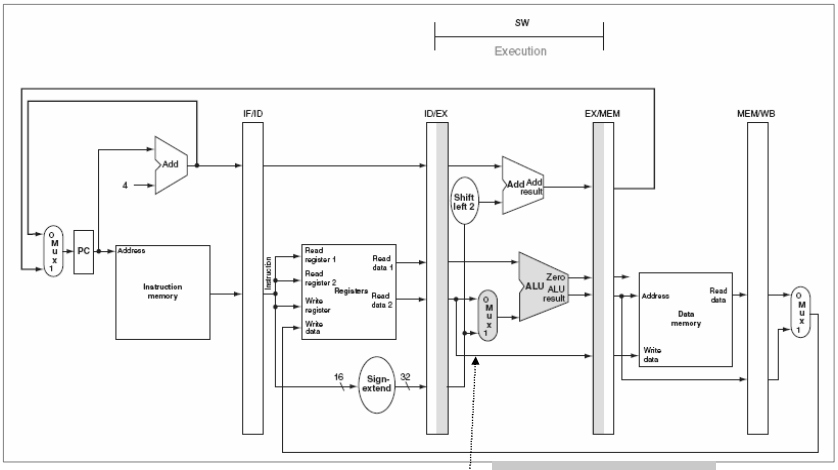
Fig. 6.14 (b): MEM and WB: the 4th and 5th pipe stages of a **load** instruction



- Highlighting the portions of the datapath in Fig. 6.11 used in this pipe stage.
- Next, data is read from the MEM/WB pipeline register and written into the register file in the middle of the datapath.

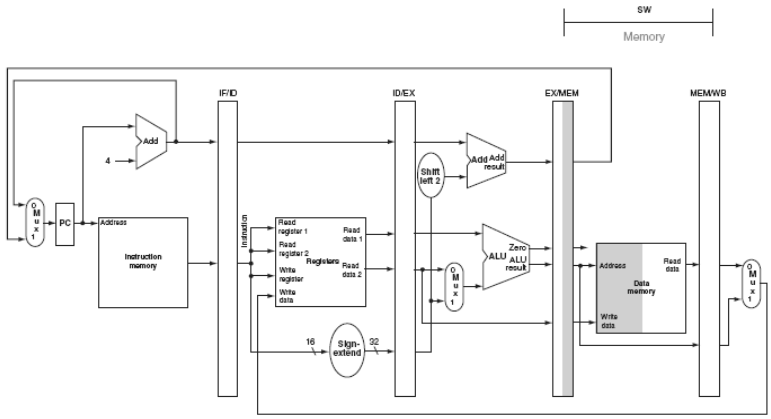
35/119

Fig. 6.15: EX: The third pipe stage of a **store** instruction



- Unlike the third stage of the load instruction in Fig. 6.13, the second register value is loaded into the EX/MEM pipeline register to be used in the next stage.
- Although it wouldn't hurt to always write this second register into the EX/MEM pipeline register, we write the second register only on a store instruction to make the pipeline easier to understand.

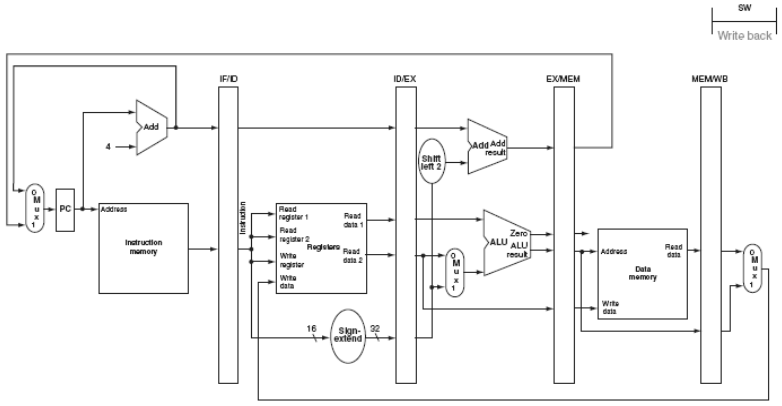
Fig. 6.16 (a): MEM and WB: the 4th and 5th pipe stages of a **store** instruction



- In the fourth stage, the data is written into data memory for the store.
- Note that the data comes from the EX/MEM PR and that nothing is changed in the MEM/WB PR.

37/119

Fig. 6.16 (b): MEM and WB: the 4th and 5th pipe stages of a **store** instruction



- Once the data is written in memory, there is nothing left for the store instruction to do, so nothing happens in stage 5.

38/119

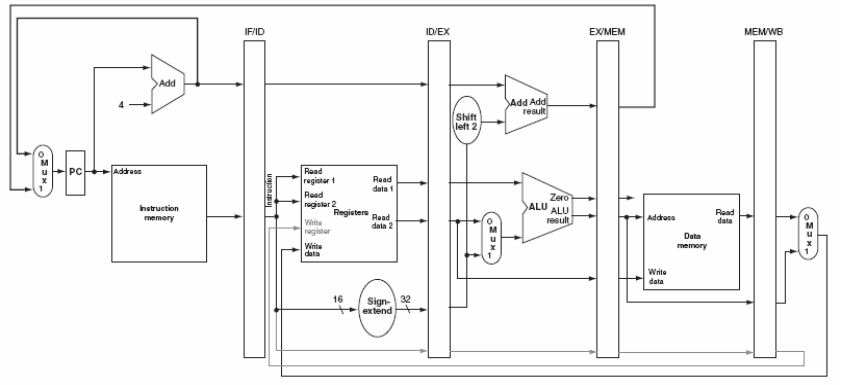
Five Pipe Stages

- Example: the five stages of a store instruction: see Fig. 6.15
- Observations:
  - Any information needed in a later pipe stage must be passed to that stage via a pipeline register.
  - Each logical component of the datapath can be used only within a single pipeline stage
    - such as instruction memory, register read ports, ALU, data memory, and register write port.
    - Otherwise, we would have a structural hazard.

Five Pipe Stages

- Question:
  - in Fig. 6.14 (b) for a load: which instruction supplies the write register number?
- Answer:
  - the instruction in the IF/ID PR supplies the write register number
    - yet this instruction occurs considerably after the load instruction!
- Hence, we need to preserve the destination register number in the load instruction.
  - Just as store passed the register contents from the ID/EX to the EX/MEM for use in the MEM stage,
  - load must pass the register number from the ID/EX through EX/MEM to the MEM/WB for use in the WB stage.
- Another way to think about the passing of the register number:
  - in order to share the pipelined datapath, we need to preserve the instruction read during the IF stage,
  - so each PR contains a portion of the instruction needed for that stage and later stages.

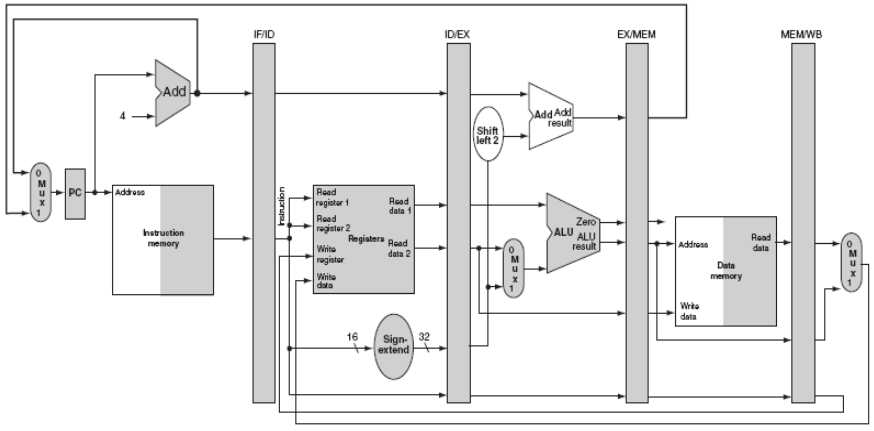
Fig. 6.17: The corrected pipelined datapath to properly handle the load instruction



- The write register # now comes from the MEM/WB along with the data.
- The register # is passed from the ID pipe stage until it reaches the MEM/WB, adding 5 more bits to the last three pipeline registers.
- This new path is shown in color.

41/119

Fig. 6.18:



- The portion of the datapath in Fig. 6.17 that is used in all five stages of a load instruction.

42/119

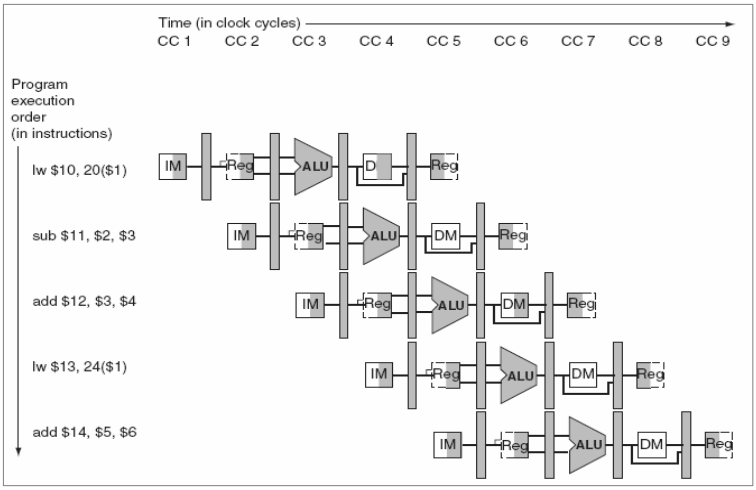
Graphically Representing Pipelines

- Two basic styles of pipeline figures:
  - multiple-clock-cycle pipeline diagrams
    - such as Fig. 6.10 on page 387
    - simpler but do not contain all the details
  - single-clock-cycle pipeline diagrams
    - such as Figs. 6.12 through 6.16.
- Fig. 6.19 shows the multiple-clock-cycle pipeline diagram for these instructions.
  - shows the physical resources used at each stage
- Fig. 6.20 shows the more traditional version of the multiple-clock-cycle pipeline diagram.
  - uses the name of each stage.

```
lw    $10, 20($1)
sub   $11, $2, $3
add   $12, $3, $4
lw    $13, 24($1)
add   $14, $5, $6
```

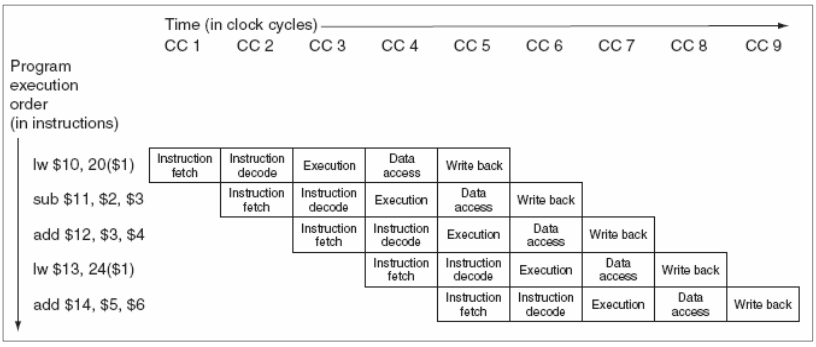
A single-clock-cycle diagram represents a vertical slice through a set of multiple-clock-cycle diagrams, showing the usage of the datapath by each of the instructions in the pipeline at the designated clock cycle.

Fig. 6.19: Multiple-clock-cycle pipeline diagram of five instructions



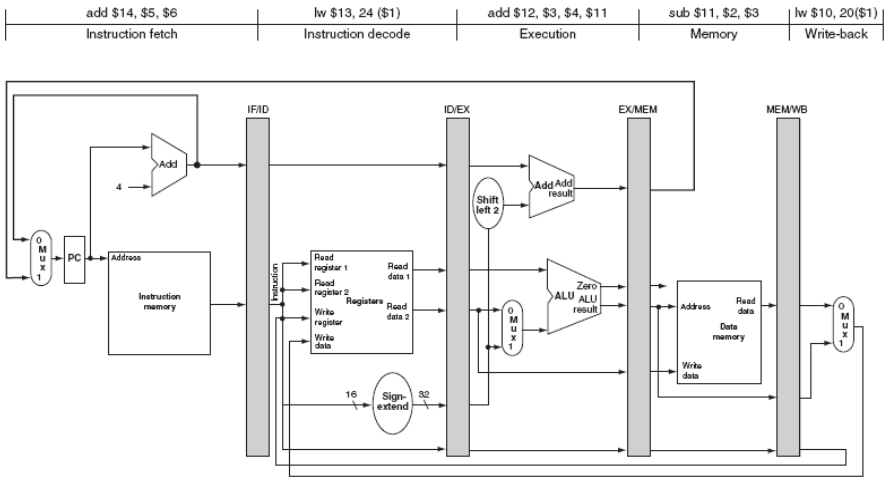
- This pipeline representation shows the complete execution of instructions in a single figure
- Instructions are listed in instruction execution order from top to bottom, and clock cycles move from left to right.
- Unlike Fig. 6.4, here we show the pipeline registers between each stage.
- Fig. 6.20 shows the traditional way to draw this diagram

Fig. 6.20:



- Traditional multiple-clock-cycle pipeline diagram of five instructions in Figure 6.19.

Fig. 6.21:

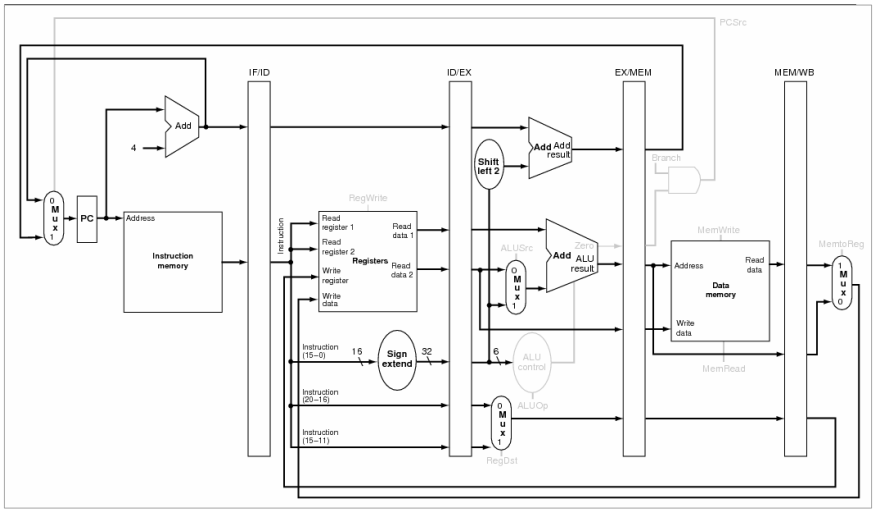


- The single-clock-cycle diagram corresponding to cc5 of the pipeline in Fig. 6.19 and 6.20
- A single-clock-cycle figure is a vertical slice through a multiple-clock-cycle diagram

6.3: Pipeline Control

- As for the single-cycle implementation discussed in Chapter 5, we assume that the PC is written on each clock cycle:
  - so there is no separate write signal for the PC.
- By the same argument, there are no separate write signals for the pipeline registers (IF/ID, ID/EX, EX/MEM, and MEM/WB):
  - since the pipeline registers are also written during each clock cycle.
- To specify control for the pipeline, we need only set the control values during each pipeline stage.
  - Because each control line is associated with a component active in only a single pipeline stage,
  - we can divide the control lines into five groups according to the pipeline stage.
- Since pipelining the datapath leaves the meaning of the control lines unchanged, we can use the same control values as before.
- Implementing control means setting the nine control lines to these values in each stage for each instruction.
  - The simplest way to do this is to extend the pipeline registers to include control information.

Fig. 6.22: The pipelined datapath of Fig 6.17 with the control signals identified



- This datapath borrows the control logic for PC source, register destination number, and ALU control from Chapter 5. Note that we now need the 6-bit funct field of the instruction in the EX stage as input to ALU control, so these bits must also be included in the ID/EX pipeline register.
- These 6 bits are also the 6 LSBs of the immediate field in the instruction, so the ID/EX pipeline register can supply them from the immediate field since sign extension leaves these bits unchanged.

Fig. 6.23:

Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

- A copy of Fig. 5.12 on p. 302.
- This figure shows how the ALU control bits are set depending on the ALUOp control bits and the different funct codes for the R-type instruction.

Fig. 6.24:

Signal name	Effect when deasserted (0)	Effect when asserted (1)
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

- A copy of Fig 5.16 on p. 306. The function of each of seven control signals is defined.
- The ALU control lines (ALUOp) are defined in the second column of Figure 6.23.
- When a 1-bit control to a two-way mux is asserted, the mux selects the input corresponding to 1; otherwise, if the control is deasserted, the mux selects the 0 input.
- Note that PCSrc is controlled by an AND gate in Fig. 6.22.
  - If the Branch signal and the ALU Zero signal are both set, then PCSrc is 1; ow, it is 0.
  - Control sets the Branch signal only during a beq instruction; otherwise, PCSrc is set to 0.

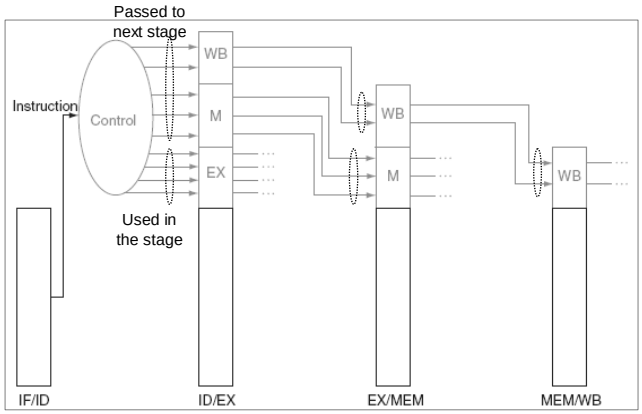
Fig. 6.25:

Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

- The values of the control lines are the same as in Figure 5.18 on page 308
- but they have been shuffled into three groups corresponding to the last three pipeline stages

51/119

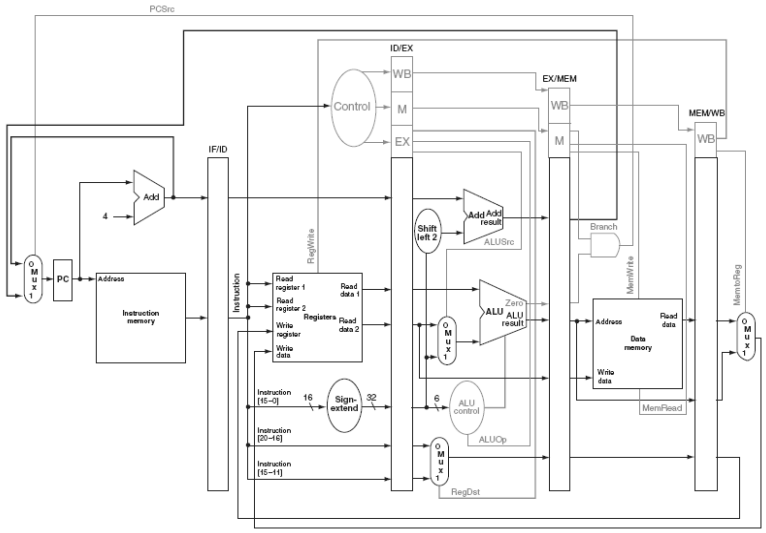
Fig. 6.26: The control lines for the final three stages



- Note that:
  - 4 of the 9 control lines are used in the EX phase
  - with the remaining 5 control lines passed on to the EX/MEM pipeline register extended to hold the control lines
  - 3 are used during the MEM stage
  - and the last 2 are passed to MEM/WB for use in the WB stage

52/119

Fig. 6.27: The pipelined datapath of Figure 6.22



- With the control signals connected to the control portions of the pipe-line registers
- The control values for the last 3 stages are created during the ID stage and then placed in the ID/EX pipeline register
- The control lines for each pipeline stage are used, and remaining control lines are then passed to the next pipeline stage

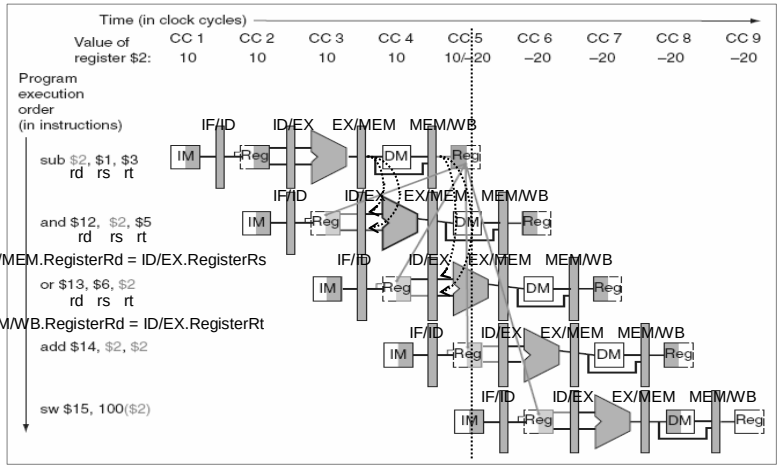
6.4: Data Hazards and Forwarding

- A sequence with many dependences:

```
sub   $2, $1,$3     # Register $2 written by sub
and   $12,$2,$5     # 1st operand($2) depends on sub
or    $13,$6,$2     # 2nd operand($2) depends on sub
add   $14,$2,$2     # 1st($2) & 2nd($2) depend on sub
sw    $15,100($2)   # Base ($2) depends on sub
```

- How would this sequence perform with our pipeline?
- One potential hazard can be resolved by the design of the register file hardware:
  - What happens when a register is read and written in the same clock cycle?
  - We assume that the write is in the first half of the clock cycle and the read is in the second half, so the read delivers what is written.
  - As is the case for many implementations of register files, we have no data hazard in this case.

Fig. 6.28: Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences



- All the dependent actions are shown in color, and "CC 1" at the top of the figure means clock cycle 1.
- Only sub writes into $2, and all the following instructions read $2.
- This register is written in clock cycle 5, so the proper value is unavailable before clock cycle 5. (A read of a register during a clock cycle returns the value written at the end of the first half of the cycle, when such a write occurs.)
- The colored lines from the top datapath to the lower ones show the dependences. Those that must go backwards in time are pipeline data hazards.

Hazard Conditions

- When is the data from the `sub` instruction actually produced?
  - The result is available at the end of the EX stage or CC3.
- When is the data actually needed by the `and` and `or` instructions?
  - At the beginning of the EX stage, or CC4 and CC5, respectively.
- Thus, we can execute this segment without stalls if we simply **forward** the data as soon as it is available to any units that need it before it is available to read from the register file.
- For simplicity, we consider only the challenge of forwarding to an operation in the EX stage,
  - which maybe either an ALU operation or an effective address calculation.
  - This means that when an instruction tries to use a register in its EX stage that an earlier instruction intends to write in its WB stage, we actually need the values as inputs to the ALU.

Hazard Conditions

ID/EX.RegisterRs

the name of the field in that register

the name of the pipeline register

1a: EX/MEM.RegisterRd = ID/EX.RegisterRs

1b: EX/MEM.RegisterRd = ID/EX.RegisterRt

2a: MEM/WB.RegisterRd = ID/EX.RegisterRs

2b: MEM/WB.RegisterRd = ID/EX.RegisterRt

Example: p. 406

Dependence Detection

Classify the dependences in this sequence from p. 403:

```
sub $2, $1, $3      # Register $2 set by sub
and $12,$2, $5      # 1st operand($2) set by sub
or  $13,$6, $2      # 2nd operand($2) set by sub
add $14,$2, $2      # 1st($2) & 2nd($2) set by sub
sw  $15,100($2)     # Index($2) set by sub
```

As mentioned above, the `sub-and` is a type 1a hazard.

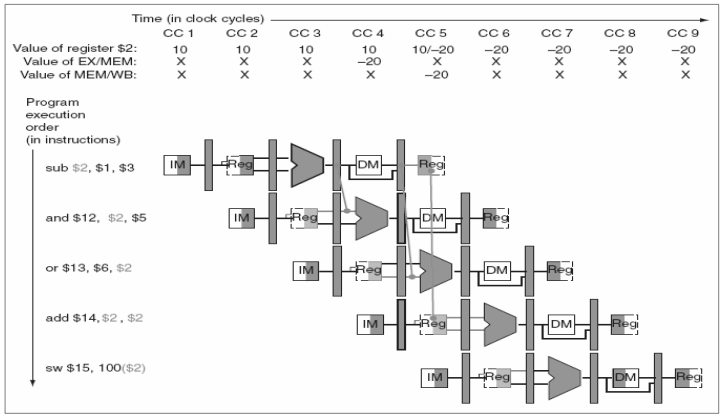
The remaining hazards are as follows:

- The `sub-or` is a type 2b hazard:
  `MEM/WB.RegisterRd = ID/EX.RegisterRt = $2`
- The two dependences on `sub-add` are not hazards because the register file supplies the proper data during the ID stage of `add`.
- There is no data hazard between `sub` and `sw` because `sw` reads `$2` the clock cycle *after* `sub` writes `$2`.

Hazard Detection

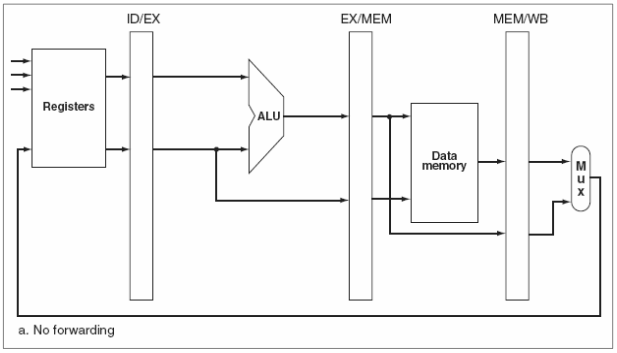
- Forward if necessary: to check to see if the RegWrite signal will be active:
  - examining the WB control field of the PR during the EX and MEM stages determines if RegWrite is asserted.
- Also, MIPS requires that every use of `$0` as an operand must yield an operand value of zero.
  - In the event that an instruction in the pipeline has `$0` as its destination (for example, `sll $0, $1, 2`), we want to avoid forwarding its possibly nonzero result value.
  - Not forwarding results destined for `$0` frees the assembly programmer and the compiler of any requirement to avoid using `$0` as a destination.
- The conditions above thus work properly as long we add
  - EX/MEM.RegisterRd ≠ 0 to the first hazard condition and
  - MEM/WB.RegisterRd ≠ 0 to the second.

Fig. 6.29: The dependences between the pipeline registers move forward in time



- so it is possible to supply the inputs to the ALU needed by the `and` instruction and `or` instruction by forwarding the results found in the pipeline registers.
- The values in the pipeline registers show that the desired value is available before it is written into the register file.
- We assume that the register file forwards values that are read and written during the same clock cycle, so the `add` does not stall, but the values come from the register file instead of a pipeline register.
- Register file "forwarding" — that is, the read gets the value of the write in that clock cycle — is why clock cycle 5 shows register $2 having the value 10 at the beginning and – 20 at the end of the clock cycle.
- As in the rest of this section, we handle all forwarding except for the value to be stored by a store instruction.

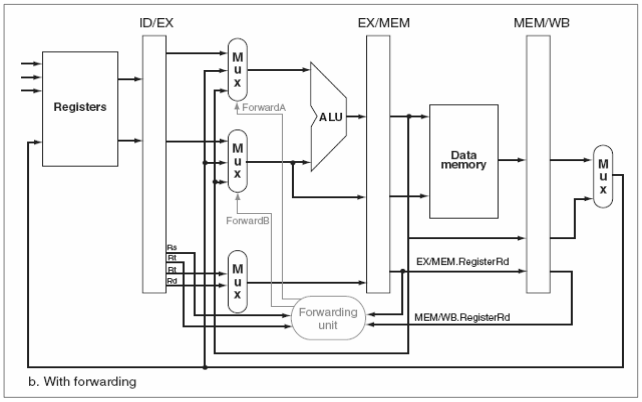
Fig. 6.30 (a):



- The ALU and pipeline registers before adding forwarding

61/119

Fig. 6.30 (b):



- The muxes have been expanded to add the forwarding paths, and we show the forwarding unit.
- The new hardware is shown in color. This figure is a stylized drawing, however, leaving out details from the full datapath such as the sign extension hardware.
- Note that the `ID/EX.RegisterRt` field is shown twice, once to connect to the mux and once to the forwarding unit, but it is a single signal.
- As in the earlier discussion, this ignores forwarding of a store value to a store instruction.

62/119

Fig. 6.31:

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

- The control values for the forwarding multiplexors in Figure 6.30.
- The signed immediate that is another input to the ALU is described in the elaboration at the end of this section.

Hazard Detection

- If we can take the inputs to the ALU from any pipeline register rather than just ID/EX, then we can forward the proper data.
- By adding multiplexors to the input of the ALU and with the proper controls, we can run the pipeline at full speed in the presence of these data dependences.

1. EX hazard:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
```

Hazard Detection

2. MEM hazard:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

- There is no hazard in the WB stage because we assume that the register file supplies the correct result if the instruction in the ID stage reads the same register written by the instruction in the WB stage.
- Such a register file performs another form of forwarding, but it occurs within the register file.

Hazard Detection

- One complication is potential data hazards between the result of the instruction in the WB stage, the result of the instruction in the MEM stage, and the source operand of the instruction in the ALU stage.
- For example, when summing a vector of numbers in a single register, a sequence of instructions will all read and write to the same register:

```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
. . .
```

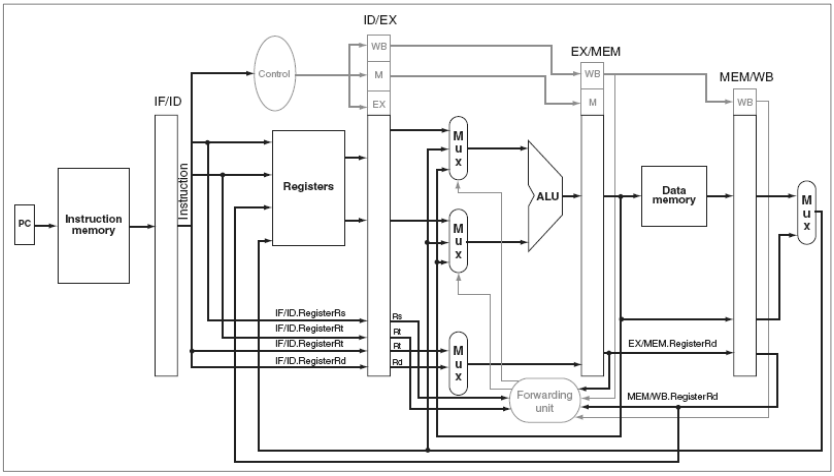
- In this case, the result is forwarded from the MEM stage because the result in the MEM stage is the more recent result.
- Thus the control for the MEM hazard would be (with the additions highlighted):

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)

and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

Fig. 6.32: The datapath modified to resolve hazards via forwarding

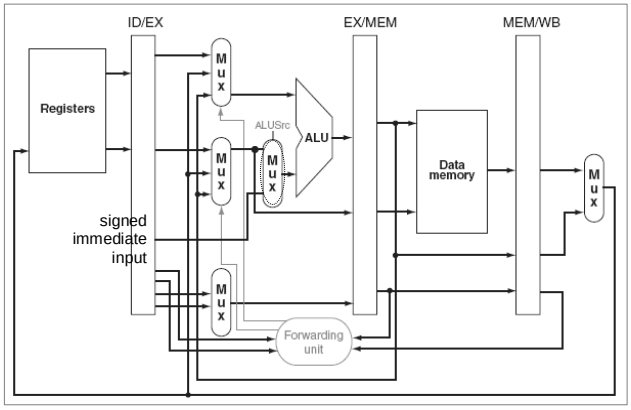


- Compared with the datapath in Fig. 6.27 on p. 404, the additions are the multiplexors to the inputs to the ALU. This figure is a more stylized drawing, however, leaving out details from the full datapath such as the branch hardware and the sign extension hardware.

Elaboration

- Forwarding can also help with hazards when **store** instructions are dependent on other instructions.
  - Since they use just one data value during the MEM stage, forwarding is easy.
- But consider loads immediately followed by stores. We need to add more forwarding hardware to make memory-to-memory copies run faster.
  - It is possible to avoid a stall, since the data exists in the MEM/WB register of a load instruction in time for its use in the MEM stage of a store instruction.
- In addition, the signed-immediate input to the ALU, needed by loads and stores.
  - Since central control decides between register and immediate, and
  - since the forwarding unit chooses the pipeline register for a register input to the ALU,
  - the easiest solution is to add a 2:1 multiplexor that chooses between the ForwardB multiplexor output and the signed immediate.

Fig. 6.33:



- A close-up of the datapath in Figure 6.30 on page 409 shows a 2:1 multiplexor, which has been added to select the signed immediate as an ALU input.

6.5: Data Hazards and Stalls

- Problem:
  - when a `lw` (that needs to write a register) and the following instruction that needs to read the same register, see Fig. 6.34:
  - this hazard cannot be solved by forwarding, and results in a stall.
- Solution:
  - Add a **hazard detect unit**
  - Insert **nops** into the pipeline
  - Then use a **forwarding unit**
- How to detect the hazard?
  - It operates during the ID stage so that it can insert the stall between the load and its use.

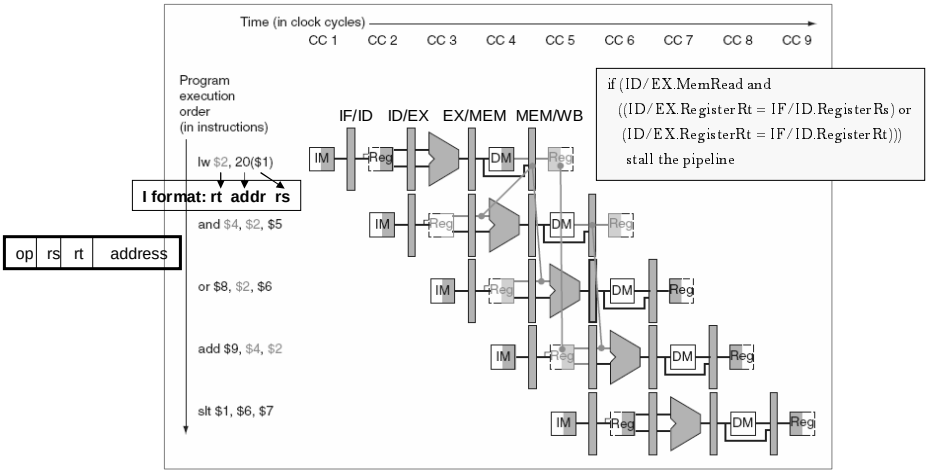
```
if (ID/EX.MemRead and
  ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
  (ID/EX.RegisterRt = IF/ID.RegisterRt)))
  stall the pipeline
```

- tests to see if the instruction is a load
- check to see if the destination register field of the load in the EX stage matches either source register of the instruction in the ID stage.
- If the condition holds, the instruction stalls 1 clock cycle. After this 1-cycle stall, the forwarding unit can handle the dependence and execution proceeds.

How to stall a pipeline?

- If the instruction in the ID stage is stalled, then the instruction in the IF stage must also be stalled;
  - otherwise, we would lose the fetched instruction.
- Preventing these two instructions from making progress is accomplished simply by
  - preventing the PC register and the IF/ID from changing.
  - Provided these registers are preserved,
    - the instruction in the IF stage will continue to be read using the same PC, and
    - the registers in the ID stage will continue to be read using the same instruction fields in the IF/ID.
- Returning to our favorite analogy, it's as if
  - you restart the washer with the same clothes and
  - let the dryer continue tumbling empty.
- Of course, like the dryer, the back half of the pipeline starting with the EX stage
  - must be doing something;
  - what it is doing is executing instructions that have no effect: **nops**.
    - nop: an instruction that does no operation to change state.

Fig. 6.34: A pipelined sequence that needs **stall**

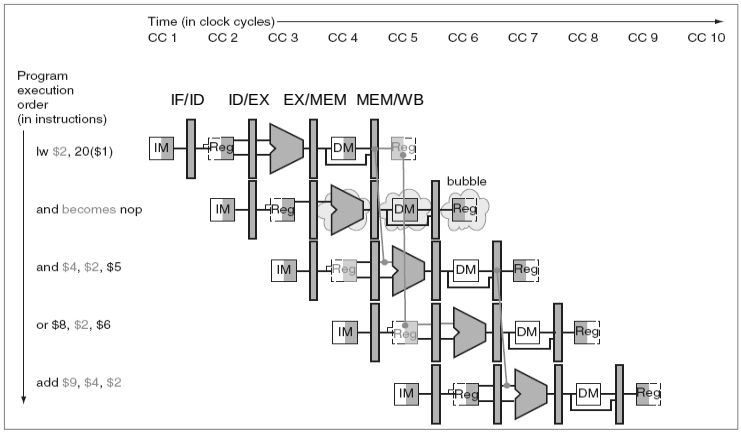


- A load-use stall: since the dependence between the `lw` and the following instruction (`and`) goes backwards in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit.

How to insert nops into a pipeline?

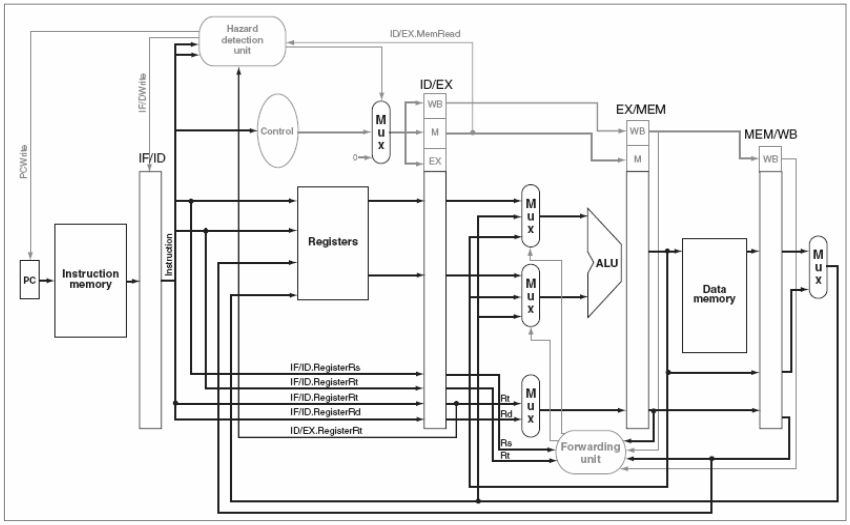
- In Fig. 6.25, we see that
  - deasserting all nine control signals (setting to 0) in the EX, MEM, and WB stages
    - only the signals RegWrite and MemWrite need be 0,
    - while the other control signals can be don't cares.
  - will create a "do nothing" or nop instruction.
- By identifying the hazard in the ID stage,
  - we can insert a bubble into the pipeline
  - by changing the EX, MEM, and WB control fields of the ID/EX to 0.
- These control values
  - are percolated (passed through or filtered) forward at each clock cycle with the proper effect:
    - no registers or memories are written if the control values are all 0.
- Fig. 6.35 shows what really happens in the hardware:
  - the pipeline execution slot associated with and is turned into a nop and
  - all instructions beginning with and are delayed one cycle.
- The hazard forces the `and` and `or` instructions
  - to repeat in clock cycle 4 what they did in clock cycle 3:
    - and reads registers and decodes, and
    - `or` is refetched from instruction memory.
- Such repeated work is what a stall looks like,
  - but its effect is to stretch the time of the `and` and `or` instructions and
  - delay the fetch of the add instruction.

Fig. 6.35: The way stalls are really inserted into the pipeline



- A bubble is inserted beginning in clock cycle 4, by changing the and instruction to a nop. Note that the and instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5 (vs. the unstalled position in clock cycle 4). Likewise the `or` instruction is fetched in clock cycle 3, but its ID stage is delayed until clock cycle 5 (vs. the unstalled clock cycle 4 position). After insertion of the bubble, all the dependences go forward in time and no further hazards occur.

Fig. 6.36: Pipelined control with hazard detection and the forwarding unit



- Two multiplexors are used for the forwarding. The hazard detection unit controls the writing of the PC, the IF/ID registers, and the multiplexor that chooses between the real control values and all 0s. The hazard detection unit stalls and deasserts the control fields if the load-use hazard test above is true.

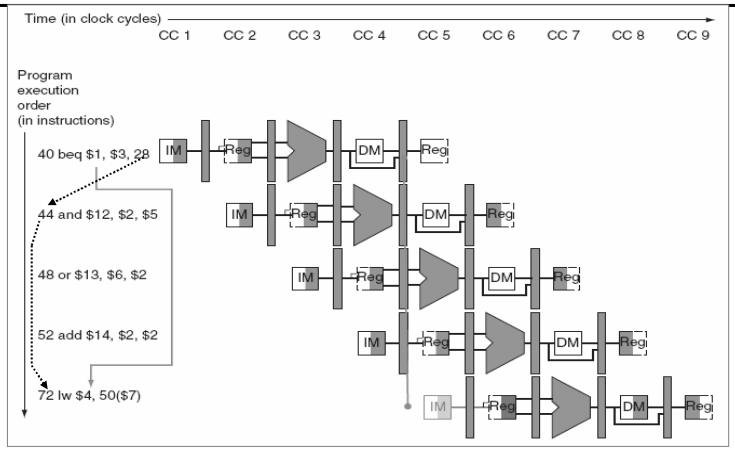
75/119

6.6: Branch Hazards

- An instruction must be fetched at every clock cycle to sustain the pipeline
  - But the decision about whether to branch doesn't occur until the MEM pipeline stage
- This delay in determining the proper instruction to fetch is called a control hazard or branch hazard.
- Comparison:
  - control hazards occur less frequently than data hazards, and
  - there is nothing as effective against control hazards as forwarding is for data hazards.
- Solutions:
  - Two schemes for resolving control hazards
    - Assume branch not taken
    - Reducing the delay of branches
  - One optimization to improve these schemes
    - Dynamic branch prediction

76/119

Fig. 6.37: The impact of the pipeline on the branch instruction



- The numbers to the left of the instruction (40, 44, . . .) are the addresses of the instructions. Since the branch instruction decides whether to branch in the MEM stage - clock cycle 4 for the beq instruction above - the three sequential instructions that follow the branch will be fetched and begin execution.
- Without intervention, those three following instructions will begin execution before beq branches to lw at location 72.
- Fig. 6.7 assumed extra hardware to reduce the control hazard to 1 clock cycle; this figure uses the nonoptimized datapath.

77/119

Assume Branch Not Taken

- Branch Stalling: stalling until the branch is complete is too slow!
- Improvement: assume that the branch will not be taken
  - thus continue execution down the sequential instructions
  - If the branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target.
    - Discarding instructions: to flush instructions in the IF, ID, and EX stages of the pipeline.
- **Flush** (instructions):
  - to discard instructions in a pipeline, usually due to an unexpected event.
  - by changing the original control values to 0s

78/119

Reducing the Delay of Branches

- Computing the branch target address:
  - just move the branch adder from the EX stage to the ID stage;
    - The PC value and the immediate field are already in the IF/ID
  - Fewer instructions need be flushed
    - the one currently being fetched
  - of course, the branch target address calculation will be performed for all instructions, but only used when needed.
- Evaluating the branch decisions
  - Moving the branch test to the ID stage implies additional forwarding and hazard detection hardware
    - since a branch dependent on a result still in the pipeline must still work properly with this optimization
- Two difficulties in evaluating the branch decisions:
  - 1. During ID: need to decide whether a bypass to the equality unit is needed,
    - Forwarding for the operands of branches was formerly handled by the ALU forwarding logic, but the introduction of the equality test unit in ID will require new forwarding logic.
  - 2. Because the values in a branch comparison are needed during ID but may be produced later in time
    - it is possible that a data hazard can occur and a stall will be needed.

Example: p. 419

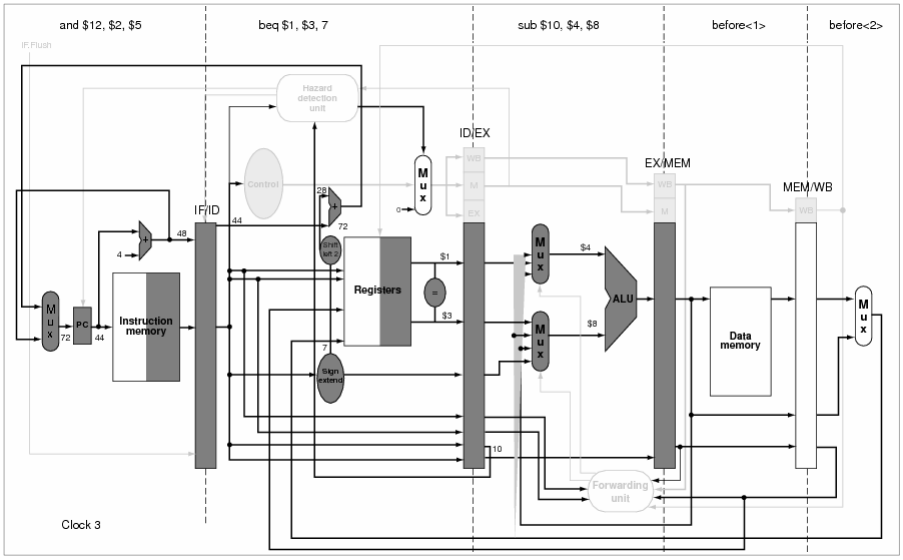
- Pipeline Branch

  Show what happens when the branch is taken, assuming that:
  - the pipeline is optimized for branches that are not taken and
  - we moved the branch execution to the ID stage

```
36  sub $10, $4, $8 #
40  beq $1,  $3, 7  #  PC-relative branch to 40 + 4 + 7 * 4 = 72
44  and $12, $2, $5 #
48  or  $13, $2, $6 #
52  add $14, $4, $2 #
56  slt $15, $6, $7 #
 . . .              #
72  lw  $4,  50($7) #
```

- Fig. 6.38 shows what happens when a branch is taken.
  - Unlike Fig. 6.37, there is only one pipeline bubble on a taken branch.

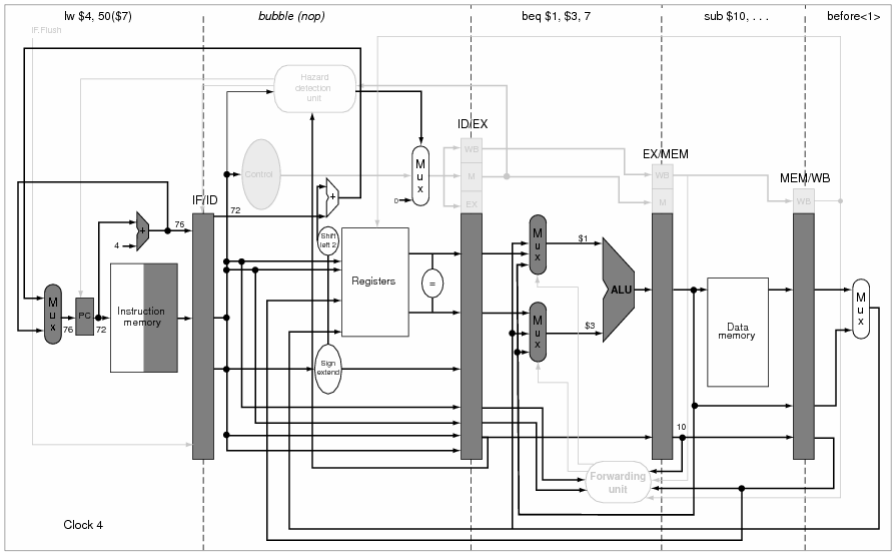
Fig. 6.38 (a):



- The ID stage of clock cycle 3 determines that a branch must be taken, so it selects 72 as the next PC address and zeros the instruction fetched for the next clock cycle.

81/119

Fig. 6.38 (b):



- Clock cycle 4 shows the instruction at location 72 being fetched and the single bubble or nop instruction in the pipeline as a result of the taken branch. (Since the nop is really `sll $0, $0, 0`, it's arguable whether or not the ID stage in clock 4 should be highlighted).

Dynamic Branch Prediction

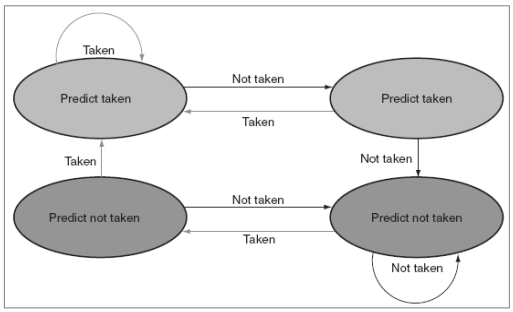
- Dynamic branch prediction:
  - Assuming a branch is not taken is one simple form of branch prediction
    - in an aggressive pipeline, a simple static prediction scheme will probably waste too much performance.
  - One approach is to look up the address of the instruction
    - to see if a branch was taken the last time this instruction was executed, and,
    - if so, to begin fetching new instructions from the same place as the last time.
- Implementations:
  - With more hardware, it is possible to try to predict branch behavior during program execution.
  - One implementation of that approach is a branch prediction buffer or branch history table.
- Branch prediction buffer: also called branch history table:
  - a small memory
    - indexed by the lower portion of the address of the branch instruction
    - contains one or more bits indicating whether the branch was recently taken or not.
- Prediction is just a hint that is assumed to be correct, so fetching begins in the predicted direction.
  - If the hint turns out to be wrong, the incorrectly predicted instructions are deleted,
  - the pre-diction bit is inverted and stored back, and
  - the proper sequence is fetched and executed.

Example: p. 421

Loops and Prediction

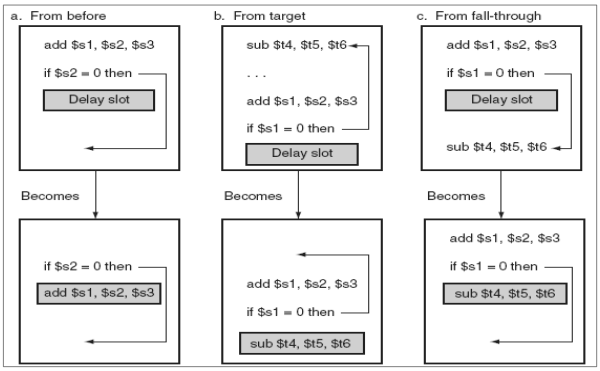
- Problem:
  - Consider a loop branch that branches nine times in a row, then is not taken once. What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the prediction buffer?
- Answer:
  - The steady-state prediction behavior will mispredict on the first and last loop iterations.
    - Mispredicting the last iteration is inevitable since the prediction bit will say taken: the branch has been taken nine times in a row at that point.
    - The misprediction on the first iteration happens because the bit is flipped on prior execution of the last iteration of the loop, since the branch was not taken on that exiting iteration.
  - Thus, the prediction accuracy for this branch that is taken 90% of the time is only 80% (two incorrect predictions and eight correct ones).

Fig. 6.39: The FSM in a 2-bit prediction scheme



- By using 2 bits rather than 1, a branch that strongly favors taken or not taken — as many branches do — will be mispredicted only once.
- The 2 bits are used to encode the four states in the system.
- The 2-bit scheme is a general instance of a counter-based predictor,
  - which is incremented when the prediction is accurate and decremented otherwise,
  - and uses the midpoint of its range as the division between taken and not taken.

Fig. 6.40: Scheduling the branch delay slot



- The top box in each pair shows the code before scheduling
- the bottom box shows the scheduled code

- In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice.
- Strategies (b) and (c) are used when (a) is not possible. In the code sequences for (b) and (c), the use of `$s1` in the branch condition prevents the `add` instruction (whose destination is `$s1`) from being moved into the branch delay slot.
- In (b) the branch-delay slot is scheduled from the target of the branch; usually the target instruction will need to be copied because it can be reached by another path. Strategy (b) is preferred when the branch is taken with high probability, such as a loop branch.
- Finally, the branch may be scheduled from the not-taken fall-through as in (c).
- To make this optimization legal for (b) or (c), it must be OK to execute the `sub` instruction when the branch goes in the unexpected direction. By "OK" we mean that the work is wasted, but the program will still execute correctly. This is the case, for example, if `$t4` were an unused temporary register when the branch goes in the unexpected direction.

Example: p. 425

- Problem: compare performance for
  - single cycle
  - multicycle
  - pipelined control

  using the SPECint2000 instruction mix
- Given parameters:
  - See Examples on p. 315 and 330, and
  - assuming the same cycle times per unit as the Example on p. 315.
  - For pipelined execution, assume that half of the load instructions are immediately followed by an instruction that uses the result, that the branch delay on misprediction is 1 clock cycle, and that one-quarter of the branches are mispredicted.
  - Assume that jumps always pay 1 full clock cycle of delay, so their average time is 2 clock cycles.
  - Ignore any other hazards.

Answer: p. 425

From the Example on p. 315 ("Performance of Single-Cycle Machines"), we get the following functional unit times:

- 200 ps for memory access
- 100 ps for ALU operation
- 50 ps for register file read or write

For the single-cycle datapath, this leads to a clock cycle of 200 + 50 + 100 + 200 + 50 = 600 ps

The Example on p. 330 ("CPI in a Multicycle CPU") has the following instruction frequencies:

- 25% loads
- 10% stores
- 11% branches
- 2% jumps
- 52% ALU instructions

Furthermore, the Example on p. 330 showed that the CPI for the multiple design was 4.12. The clock cycle for the multicycle datapath and the pipelined design must be the same as the longest functional unit: 200 ps.

Answer: p. 425

For the pipelined design,

- loads take 1 clock cycle when there is no load-use dependence and 2 when there is. Hence, the average clock cycles per load instruction is 1.5.
- stores take 1 clock cycle, as do the ALU instructions.
- branches take 1 when predicted correctly and 2 when not, so the average clock cycles per branch instruction is 1.25.
- jump's CPI is 2.

Hence the average CPI is:

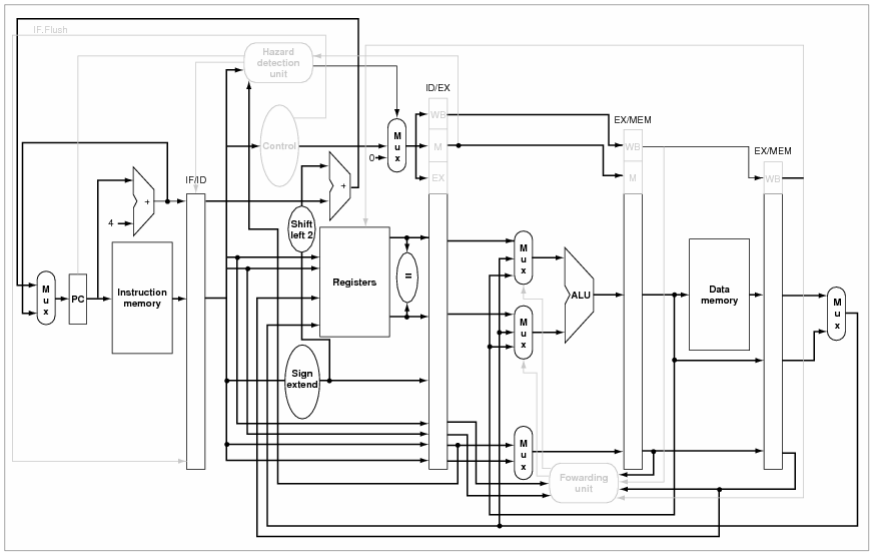
1.5x25% + 1x10% + 1x52% + 1.25x11% + 2x2% = 1.17

Let's compare the three designs by the average instruction time.

- For the single-cycle design, it is fixed at 600 ps.
- For the multicycle design, it is 200 x 4.12 = 824 ps.
- For the pipelined design, the average instruction time is 1.17 x 200 = 234 ps, making it almost twice as fast as either approach.

89/119

Fig. 6.41: the final datapath and control for this chapter



90/119

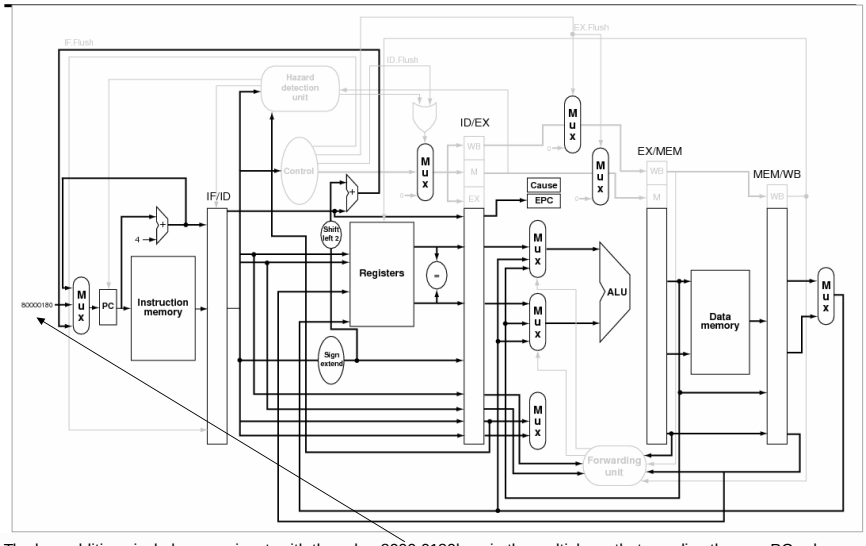
6.8: Exceptions

- Another form of control hazard involves exceptions.
  - For example, suppose the following instruction,

    ```
    add  $1,$2,$1
    ```

    has an arithmetic overflow.
- We need to transfer control to the exception routine immediately after this instruction
  - because we wouldn't want this invalid value to contaminate other registers or memory locations.
- Just as we did for the taken branch, we must
  - flush the instructions that follow the add instruction from the pipeline and
  - begin fetching instructions from the new address.
  - use the same mechanism as for taken branches, but this time the exception causes the deasserting of control lines.
- To flush instructions in the ID stage, we use the multiplexor already in the ID stage that zeros control signals for stalls.
  - When we dealt with branch mispredict, we saw how to flush the instruction in the IF stage by turning it into a nop.

Fig. 6.42: The datapath with controls to handle exceptions



- The key additions include a new input, with the value 8000 0180hex, in the multiplexor that supplies the new PC value; a Cause register to record the cause of the exception; and an Exception PC register to save the address of the instruction that caused the exception. The 8000 0180hex input to the multiplexor is the initial address to begin fetching instructions in the event of an exception. Although not shown, the ALU overflow signal is an input to the control unit.

Example: p. 429

Exception in a pipelined computer

- Given this instruction sequence:

```
40hex       sub   $11,  $2, $4
44hex       and   $12,  $2, $5
48hex       or    $13,  $2, $6
4Chex       add   $1,   $2, $1
50hex       slt   $15,  $6, $7
54hex       lw    $16,  50($7)
...
```

assume the instructions to be invoked on an exception begin like this:

```
40000040hex       sw    $25, 1000($0)
40000044hex       sw    $26, 1004($0)
...
```

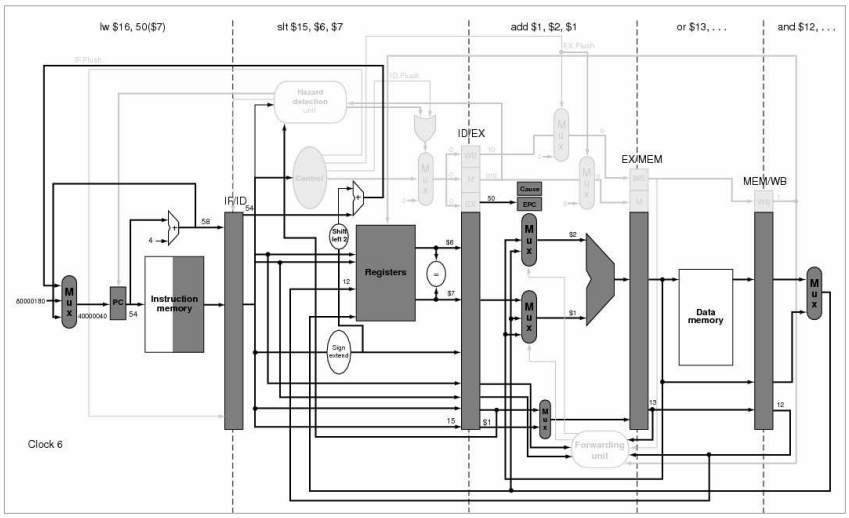
Show what happens in the pipeline if an overflow exception occurs in the `add` instruction.

Answer: p. 429

- Figure 6.43 shows the events:
  - starting with the `add` instruction in the EX stage.
  - The overflow is detected during that phase, and
    - 4000 0040_{hex} is forced into the PC.
  - Clock cycle 7 shows that the `add` and following instructions are flushed,
    - and the first instruction of the exception code is fetched.
- Note that:
  - the address of the instruction following the `add` is saved:

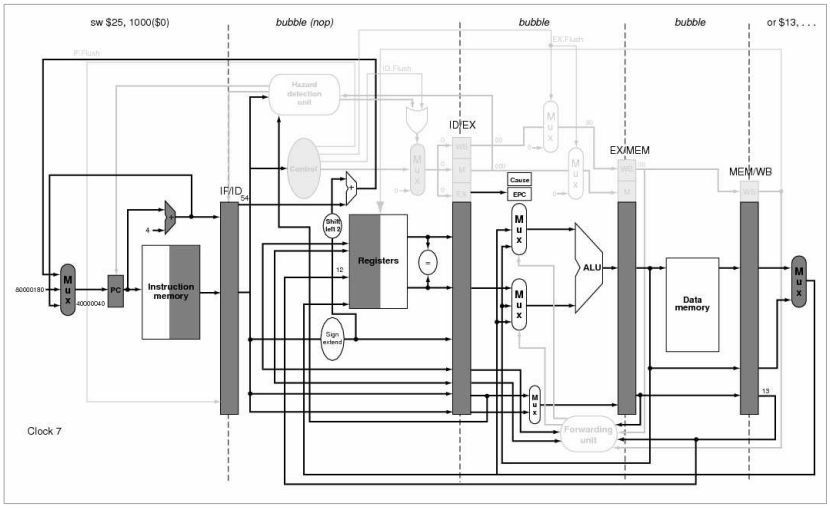
    $4C_{hex} + 4 = 50_{hex}$.

Fig. 6.43 (a): The result of an exception due to arithmetic overflow in the add instruction



- The overflow is detected during the EX stage of clock 6, saving the address following the add in the EPC register (4C + 4 = 50hex). Overflow causes all the Flush signals to be set near the end of this clock cycle, deasserting control values (setting them to 0) for the `add`.

Fig. 6.43 (b): The result of an exception due to arithmetic overflow in the add instruction



- Clock cycle 7 shows the instructions converted to bubbles in the pipeline plus the fetching of the first instruction of the exception routine — `sw $25,1000($0)` — from instruction location 4000 0040hex. Note that the `and` and `or` instructions, which are prior to the `add`, still complete.
- Although not shown, the ALU overflow signal is an input to the control unit.

Other Exceptions

- Other causes of exceptions:
  - I/O device request
  - Invoking an operating system service from a user program
  - Using an undefined instruction
  - Hardware malfunction
- With five instructions active in any clock cycle, the challenge is to associate an exception with the appropriate instruction.
  - The normal solution is to prioritize the exceptions so that it is easy to determine which is serviced first.
- I/O device requests and hardware malfunctions are not associated with a specific instruction,
  - so the implementation has some flexibility as to when to interrupt the pipeline.
- The EPC captures the address of the interrupted instructions,
  - and the MIPS Cause register records all possible exceptions in a clock cycle,
  - so the exception software must match the exception to the instruction.

Precise or Imprecise Exceptions

- Imprecise interrupt:
  - Also called imprecise exception.
  - Interrupts or exceptions in pipelined computers that are not associated with the exact instruction that was the cause of the interrupt or exception.
- Precise interrupt
  - Also called precise exception.
  - An interrupt or exception that is always associated with the correct instruction in pipelined computers.

6.9: Advanced Pipelining: Extracting More Performance

- **Instruction-Level Parallelism** (**ILP**): the parallelism among instructions.
  - two primary methods for increasing ILP:
    - increase the depth of the pipeline to overlap more instructions.
    - replicate the internal components of the computer
      - so that it can launch multiple instructions in every pipeline stage, i.e., multiple issue.
- **Multiple Issue**: a scheme whereby multiple instructions are launched in 1 clock cycle.
  - Launching multiple instructions per stage allows the instruction execution rate to exceed the clock rate or, the CPI to be less than 1.
  - use **IPC**, (Instructions Per Clock-cycle), particularly as values become less than 1!
- Two major ways to implement a multiple-issue processor:
  - with the major difference being the division of work between the compiler and the hardware.
  - Because the division of work dictates whether decisions are being made
    - Statically: at compile time
    - Dynamically: during execution
- Static Multiple Issue:
  - an approach to implementing a multiple-issue processor where many decisions are made by the compiler before execution.
- Dynamic Multiple Issue:
  - an approach to implementing a multiple-issue processor where many decisions are made during execution by the processor.

99/119

Multiple-Issue Pipeline

- issue slots: the positions from which instructions could issue in a given clock cycle;
  - by analogy these correspond to positions at the starting blocks for a sprint.
- Two primary and distinct responsibilities in a multiple-issue pipeline:
  1. Packaging instructions into issue slots:
     - How does the processor determine how many instructions and which instructions can be issued in a given clock cycle?
     - In static issue processors: this process is at least partially handled by the compiler;
     - In dynamic issue designs: it is normally dealt with at runtime by the processor,
       - although the compiler will often have already tried to help improve the issue rate by placing the instructions in a beneficial order.
  2. Dealing with data and control hazards:
     - In static issue processors: some or all of the consequences of data and control hazards are handled statically by the compiler.
     - In dynamic issue processors: attempt to alleviate at least some classes of hazards using hardware techniques operating at execution time.

100/119

The Concept of Speculation

- One of the most important methods for finding and exploiting more ILP is **speculation**:
  - an approach whereby the compiler or processor guesses the outcome of an instruction to remove it as a dependence in executing other instructions.
  - Example: we might speculate on the outcome of a branch
    - so that instructions after the branch could be executed earlier.
  - The difficulty with speculation is that it may be wrong.
- Speculation may be done:
  - in the compiler: the compiler can use speculation to reorder instructions, moving an instruction across a branch or a load across a store.
  - by the hardware: the processor hardware can perform the same transformation at runtime using techniques we discuss later in this section.
- Recovery mechanisms used for incorrect speculation:
  - In software speculation: the compiler usually inserts additional instructions that check the accuracy of the speculation and provide a fix-up routine.
  - In hardware speculation: the processor usually buffers the speculative results until it knows they are no longer speculative.
- A possible problem:
  - speculating on certain instructions may introduce exceptions that were formerly not present.
  - Example:
    - suppose a load instruction is moved in a speculative manner, but the address it uses is not legal when the speculation is incorrect.
    - The result would be that an exception that should not have occurred will occur.

Static Multiple Issue

- **issue packet**: the set of instructions that issues together in 1 clock cycle;
  - the packet may be determined statically by the compiler or dynamically by the processor.
- Static multiple-issue processors: all use the compiler to
  - assist with packaging instructions
  - and handling hazards.
- In a static issue processor, you can think of the set of instructions that issue in a given clock cycle (e.g., an issue packet) as one large instruction with multiple operations.
  - This view led to the original name for this approach: Very Long Instruction Word (**VLIW**).
  - The Intel IA-64 architecture uses this approach,
    - With a name: Explicitly Parallel Instruction Computer (**EPIC**).
- Most static issue processors also rely on the compiler to take on some responsibility for handling data and control hazards.

An Example: Static Multiple Issue with the MIPS ISA

- Consider a simple two-issue MIPS processor:
  - Issuing two instructions per cycle will require fetching and decoding 64 bits of instructions
    - one of the instructions can be an integer ALU operation or branch, and
    - the other can be a load or store.
  - the instructions be paired and aligned on a 64-bit boundary,
    - with the ALU or branch portion appearing first.
    - if one instruction cannot be used, we require that it be replaced with a no-op.
- To issue an ALU and a data transfer operation in parallel
  - the first need for additional hardware—beyond the usual hazard detection and stall logic—is extra ports in the register file (see Fig. 6.45).
    - In 1 clock cycle we may need to read two registers for the ALU operation
    - and two more for a store,
    - and also one write port for an ALU operation
    - and one write port for a load.
    - Since the ALU is tied up for the ALU operation, we also need a separate adder to calculate the effective address for data transfers.
- This two-issue processor:
  - can improve performance by up to a factor of 2.
  - Doing so, however, requires that twice as many instructions be overlapped in execution,
  - the additional overlap increases the relative performance loss from data and control hazards.
- Example:
  - in our simple five-stage pipeline, loads have a use latency of 1 clock cycle,
    - which prevents one instruction from using the result without stalling.
  - In the two-issue, five-stage pipeline, the result of a load instruction cannot be used on the next clock cycle.
    - This means that the next two instructions cannot use the load result without stalling.

103/119

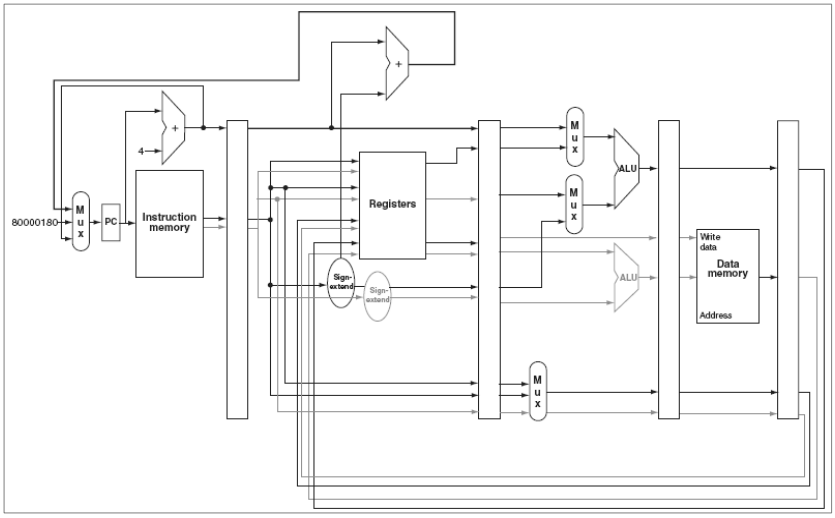
Fig. 6.44: Static two-issue pipeline in operation

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

- The ALU and data transfer instructions are issued at the same time.
- Here we have assumed the same five-stage structure as used for the single-issue pipeline.
  - Although this is not strictly necessary, it does have some advantages.
  - In particular, keeping the register writes at the end of the pipeline simplifies the handling of exceptions
  - and the maintenance of a precise exception model, which become more difficult in multiple-issue processors.

104/119

Fig. 6.45: A static two-issue datapath



- The additions needed for double issue are highlighted: another 32 bits from instruction memory, two more read ports and one more write port on the register file, and another ALU. Assume the bottom ALU handles address calculations for data transfers and the top ALU handles everything else.

Fig. 6.46:

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		lw $t0, 0($s1)	1
	addi $s1,$s1,-4		2
	addu $t0,$t0,$s2		3
	bne $s1,$zero,Loop	sw $t0, 4($s1)	4

- The scheduled code as it would look on a two-issue MIPS pipeline
  - The empty slots are nops

106/119

Fig. 6.47:

	ALU or branch instruction		Data transfer instruction		Clock cycle
Loop:	addi	$s1,$s1,-16	lw	$t0, 0($s1)	1
			lw	$t1,12($s1)	2
	addu	$t0,$t0,$s2	lw	$t2, 8($s1)	3
	addu	$t1,$t1,$s2	lw	$t3, 4($s1)	4
	addu	$t2,$t2,$s2	sw	$t0, 16($s1)	5
	addu	$t3,$t3,$s2	sw	$t1,12($s1)	6
			sw	$t2, 8($s1)	7
	bne	$s1,$zero,Loop	sw	$t3, 4($s1)	8

- The unrolled and scheduled code of Figure 6.46 as it would look on a static two-issue MIPS pipeline
  - The empty slots are nops.
  - Since the first instruction in the loop decrements $s1 by 16, the addresses loaded are the original value of $s1, then that address minus 4, minus 8, and minus 12.

107/119

The Intel IA-64 Architecture

- The IA-64 architecture is a register-register, RISC-style instruction set
  - like the 64-bit version of the MIPS architecture (called MIPS-64), but
  - with several unique features to support explicit, compiler-driven exploitation of ILP.
- Intel calls the approach EPIC (Explicitly Parallel Instruction Computer).
- The major differences between IA-64 and the MIPS architecture are the following:
  - IA-64 has many more registers than MIPS, including 128 integer and 128 floating-point registers, as well as 8 special registers for branches and 641-bit condition registers.
    - In addition, IA-64 supports register windows in a fashion similar to the original Berkeley RISC and Sun SPARC architectures.
  - IA-64 places instructions into bundles that have a fixed format and explicit designation of dependences.
  - IA-64 includes special instructions and capabilities for speculation and for branch elimination,
    - which increase the amount of ILP that can be exploited.
- The IA-64 architecture is designed to achieve the major benefits of a VLIW—
  - implicit parallelism among operations in an instruction and
  - fixed formatting of the operation fields—
    - while maintaining greater flexibility than a VLIW normally allows.
- The IA-64 architecture uses two different concepts to achieve this flexibility:
  - instruction groups: an instruction group is a sequence of consecutive instructions with no register data dependences among them.
  - bundles: IA-64 instructions are encoded in bundles, which are 128 bits wide.
    - Each bundle consists of a 5-bit template field and three instructions, each 41 bits in length.

108/119

Fig. 6.48:

Processor	Maximum instr. issues / clock	Functional units	Maximum ops. per clock	Maximum clock rate	Transistors (millions)	Power (watts)	SPEC int2000	SPEC fp2000
Itanium	6	4 integer/media 2 memory 3 branch 2 FP	9	0.8 GHz	25	130	379	701
Itanium 2	6	6 integer/media 4 memory 3 branch 2 FP	11	1.5 Ghz	221	130	810	1427

- A summary of the characteristics of the Itanium and Itanium 2, Intel's first two implementations of the IA-64architecture.
  - In addition to higher clock rates and more functional units, the Itanium 2 includes an on-chip level 3 cache, versus an off-chip level 3 cache in the Itanium.

Dynamic Multiple-Issue Processors

- **superscalar**: an advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle.
- **dynamic pipeline scheduling**: hardware support for reordering the order of instruction execution so as to avoid stalls.
- **commit unit**: the unit in a dynamic or out-of-order execution pipeline that decides when it is safe to release the result of an operation to programmer-visible registers and memory.
- **reservation station**: a buffer within a functional unit that holds the operands and the operation.
- **reorder buffer**: the buffer that holds results in a dynamically scheduled processor until it is safe to store the results to memory or a register.
- **in-order commit**: a commit in which the results of pipelined execution are written to the programmer-visible state in the same order that instructions are fetched.
- **out-of-order execution**: a situation in pipelined execution when an instruction blocked from executing does not cause the following instructions to wait.

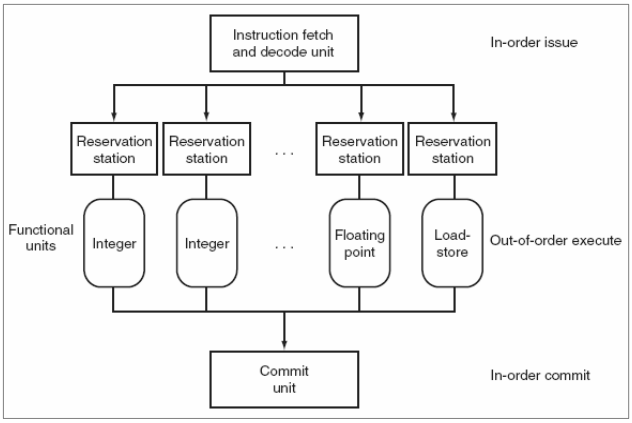
Dynamic Pipeline Scheduling

- Dynamic multiple-issue processors:
  - are also known as superscalar processors, or simply **superscalars**.
  - In the simplest superscalar processors:
    - instructions issue in order, and
    - the processor decides whether zero, one, or more instructions can issue in a given clock cycle.
- Many superscalars include **dynamic pipeline scheduling**:
  - chooses which instructions to execute in a given clock cycle while trying to avoid hazards and stalls.
- Consider the following code sequence:

```
lw   $t0, 20($s2)
addu $t1, $t0, $t2
sub  $s4, $s4, $t3
slti $t5, $s4, 20
```

  Observation:
  - even though the `sub` instruction is ready to execute, it must wait for the `lw` and `addu` to complete first,
    - which might take many clock cycles if memory is slow.
  - Dynamic pipeline scheduling allows such data hazards to be avoided either fully or partially.

Fig. 6.49: The three primary units of a dynamically scheduled pipeline

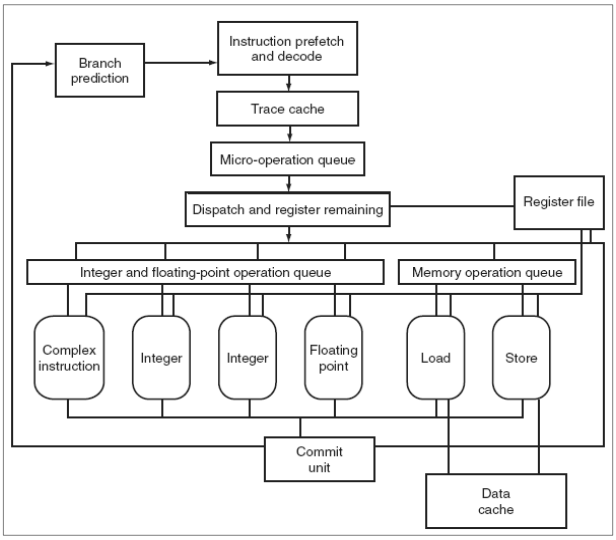


- The final step of updating the state is also called retirement or graduation

6.10: Real Stuff: The Pentium 4 Pipeline

- The Pentium 4 fetched and translated IA-32instructions into micro-operations.
  - The micro-operations are then executed by a sophisticated, dynamically scheduled, speculative **pipeline**,
    - which is capable of sustaining an execution rate of three micro-operations per clock cycle.
- The Pentium 4 combines multiple issue with deep pipelining so as to achieve both a low CPI and a high clock rate.
- Micro-architecture:
  - refer to the detailed internal architecture of a processor.
  - that is, the organization of the processor, including the major functional units, their interconnection, and control.
- The Pentium 4 gains its performance advantage over the Pentium III through several enhancements:
  1. A pipeline that is roughly twice as deep (approximately 20 cycles versus 10)and can run almost twice as fast in the same technology
  2. More functional units (7 versus 5)
  3. Support for a larger number of outstanding operations (126 versus 40)
  4. The use of a trace cache (see Chapter 7) and a much better branch predictor(4K entries versus 512)
  5. Other enhancements to the memory system, which we discuss in Chapter 7

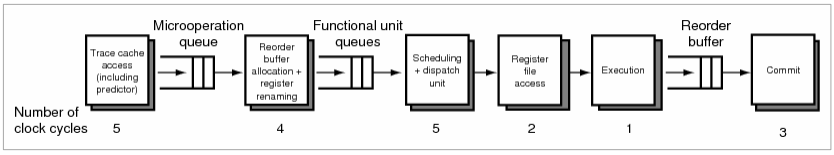
Fig. 6.50: The micro-architecture of the Intel Pentium 4



About Fig. 6.50:

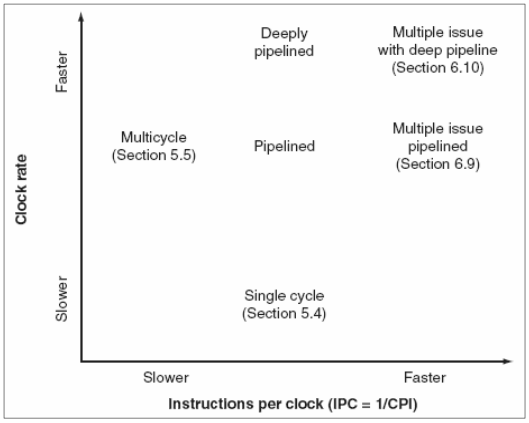
- The extensive queues allow up to 126 micro-operations to be outstanding at any point in time, including 48 loads and 24 stores.
  - There are actually seven functional units, since the FP unit includes a separate dedicated unit for floating-point moves.
  - The load and store units are actually separated into two parts, with the first part handling address calculation and the second part responsible for the actual memory reference.
  - The integer ALUs operate at twice the clock frequency, allowing two integer ALU operations to be completed by each of the two integer units in a single clock cycle.
- As we described in Chapter 5, the Pentium 4 uses a special cache,
  - called the trace cache,
  - to hold predecoded sequences of micro-operations, corresponding to IA-32 instructions.
  - The operation of a trace cache is explained in more detail in Chapter 7.
- The FP unit also handles the MMX multimedia and SSE2 instructions.
- There is an extensive bypass network among the functional units;
  - since the pipeline is dynamic rather than static, bypassing is done by tagging results and tracking source operands,
  - so as to allow a match when a result is produced for an instruction in one of the queues that needs the result.

Fig. 6.51:



- The Pentium 4 pipeline showing the pipeline flow for a typical instruction and the number of clock cycles for the major steps in the pipeline.
  - The major buffers where instructions wait are also shown.

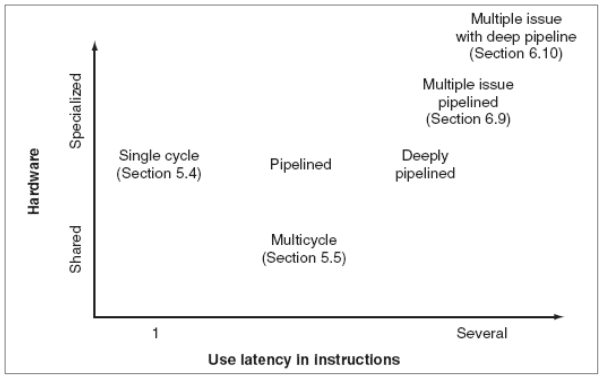
Fig. 6.52:



- The performance consequences of simple (single-cycle) datapath and multicycle datapath from Chapter 5 and the pipelined execution model in this chapter.
- Remember that CPU performance is a function of IPC times clock rate, and hence moving to the upper right increases performance. Although the instructions per clock cycle is slightly larger in the simple datapath, the pipelined datapath is close, and it uses a clock rate as fast as the multicycle datapath.

117/119

Fig. 6.53: The basic relationship between the datapaths in Figure 6.52



- Notice that the x -axis is use latency in instructions, which is what determines the ease of keeping the pipeline full.
- The pipelined datapath is shown as multiple clock cycles for instruction latency
  - because the execution time of an instruction is not shorter
  - it's the instruction throughput that is improved!

118/119

6.12: Concluding Remarks

- Conclusions:
  - Pipelining improves the **average** execution time per instruction.
  - Pipelining improves throughput, but not the inherent execution time, or latency, of instructions; the latency is similar in length to the multicycle approach.
  - Pipelining and multiple issue both attempt to exploit instruction-level parallelism.
- Limitations:
  - The presence of data and control dependences, which can become hazards, are the primary limitations on how much parallelism can be exploited.
- Techniques:
  - Scheduling and speculation, both in hardware and software, are the primary techniques used to reduce the performance impact of dependences.
- Historical Perspective:
  - The switch to longer pipelines, multiple instruction issue, and dynamic scheduling
    - in the mid-1990's has helped sustain the 60% per year processor performance increase since the early 1980s.
  - In the past, it appeared that the choice was between the highest clock rate processors and the most sophisticated superscalar processors.
  - The Pentium 4 combines both and achieves remarkable performance.