

# Review of Instruction Sets & Pipelines

Dr. Arjan Durrresi  
Louisiana State University  
Baton Rouge, LA 70810  
Durrresi@Csc.LSU.Edu

These slides are available at:  
[http://www.csc.lsu.edu/~durrresi/CSC7080\\_06/](http://www.csc.lsu.edu/~durrresi/CSC7080_06/)



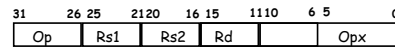
- Overview of pipelining
- Pipelining datapath
- Pipeline control

## A "Typical" RISC

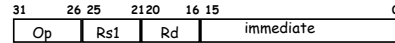
- 32-bit fixed format instruction (3 formats)
  - 32 32-bit GPR (R0 contains zero, DP take pair)
  - 3-address, reg-reg arithmetic instruction
  - Single address mode for load/store:
    - base + displacement
    - no indirection
  - Simple branch conditions
  - Delayed branch
- see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

## Example: MIPS (- DLX)

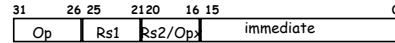
### Register-Register



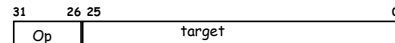
### Register-Immediate



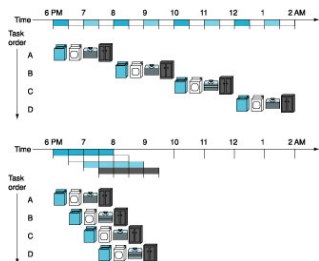
### Branch



### Jump / Call



## Pipelining Analogy



## Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

## Example

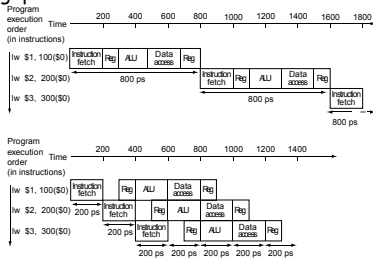
Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Register access	Register access
Store word	Instruction fetch	Register access	ALU	Register access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

## Example

Instruction class	Instruction memory	Register read	ALU	Data memory	Register write	Total
R-type	200	100	200	0	100	600ps
Load word	200	100	200	200	100	800ps
Store word	200	100	200	200	0	700ps
Branch	200	100	200	0		500ps
Jump	200					200ps

## Pipelining

- Improve performance by increasing instruction throughput



## Pipeline performance

- The pipeline execution clock must have the worst-case clock cycle of 200 ps even though some stages take only 100ps.
- Without pipelining 3 instructions -  $3 \times 800 = 2400ps$
- With pipelining 3 instructions - 1400ps

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

- Ideal conditions and large number of instructions
- If there 1,000,003 instructions
  - Pipeline -  $1000000 \times 200 + 1400 = 200,001,400$
  - Nonpipeline -  $1,000,000 \times 800 + 2400 = 800,002,400$
  - Ratio  $\approx 4$
- Pipelining improves performance by increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction

## Pipelining

- What makes it easy
  - all instructions are the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores
- What makes it hard?
  - structural hazards: suppose we had only one memory
  - control hazards: need to worry about branch instructions
  - data hazards: an instruction depends on a previous instruction
- We'll build a simple pipeline and look at these issues
- We'll talk about modern processors and what really makes it hard:
  - exception handling

## Designing Instruction Set for Pipelining

- MIPS instructions are the same length.
  - This makes much easier to fetch instructions in the first stage and to decode them in the second stage.
- In IA-32 instructions vary from 1-17 bytes, pipelining is considerably more challenging.
  - All recent IA-32 architectures translate instructions into microoperations, which are pipelined
- MIPS has only few instruction formats, with the source registers fields in the same place
  - This symmetry - the second stage can read the register file at the same time that the hardware is decoding the type of instruction
  - Without this symmetry we would need to split stage 2

## Designing Instruction Set for Pipelining

- Memory operands only appears in loads or stores in MIPS
  - We can use the execute stage to calculate the memory and then access memory in the following stage
  - If we operate on operand in memory, like IS-32, stages 3 and 4 would expand to an address stage, memory stage and then execute stage
- Operands must be aligned in memory
  - We can have instructions requiring two data memory access, the transfer can be done in a single pipeline stage

## Pipeline Hazards

- Situations when the next instructions cannot execute in the following clock cycle
- These events are called hazards
  - Structural - The hardware cannot support the combination of instructions that we want to execute at the same clock cycle
  - Data - One step must wait for the next step to complete
  - Control - Need to make decision based on the results of one instruction while others are executing

## Structural Hazards

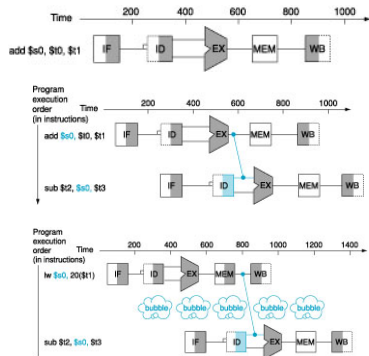
- MIPS instruction set was designed to be pipelined, making easy to avoid structural hazards
- Suppose we had a single memory instead of two
  - In slide 7, if we had a fourth instruction
  - In the same cycle the first instruction is accessing data memory while the fourth is fetching an instruction from the same memory
  -

## Data Hazards

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

- The add instruction doesn't write its result until the fifth stage
- Compilers could help but not satisfactory
- Forwarding or Bypassing - adding extra hardware to retrieve the missing item earlier from the internal resources
- Doesn't work always. Suppose we have a load instead of add, the result would be available after the fourth stage. We need to stall - pipeline stall
- Reorder code to avoid pipeline stalls

## Data Hazards



## Data Hazards

A = B + C

C = B + F

MIPS code:

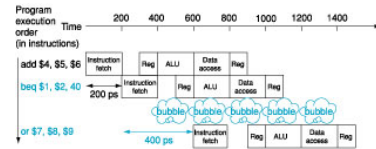
```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

Both add have a hazard because of the immediate preceding lw

## Data 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)
```

## Control Hazards



- We need to decide the next step on the result of a branch
- One-stage pipeline stall
- Prediction: One simple approach is to always predict that branches will be untaken. When you are right the pipeline proceeds at full speed. Only when branches are taken the pipeline stalls.

## Control Hazards



## Control Hazards - Solutions

- Stall - wait until the pipeline determines the outcome of the branch
  - Suppose with extra hardware and in one extra cycle the pipeline can determine the outcome of the branch
  - In SPECint2000 - branches are 13%
  - If other instructions have CPI = 1 than branches increase CPI to 1.13, same slowdown
  - If more than one cycle is needed to determine the outcome of the branch - larger slowdown
- Predict
  - One simple approach always predict that branches will be **untaken**
  - When branches are taken - the pipeline stall
- Branch prediction
  - In loops - branch backwards. It could be predicted taken for branches that jumps to an earlier address
- Dynamic prediction
  - Keep a history for branches
- Delayed Decision - used in MIPS
  - Continue with instructions not affected by the branch

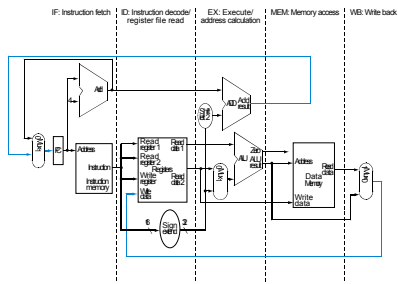
## Pipeline Performance

- After the memory system, the effective operation of the pipeline is usually the most important factor in determining the CPI of a processor

## Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

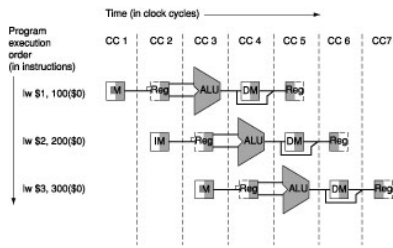
## Basic Idea



## Pipelined Datapath

- Instructions and data move generally from left to right through the five stages
- There are two exceptions
  - The write-back stage, which does the results back into the register file
    - Could lead to data hazards
  - The selection of the next value of PC, choosing between the incremented PC and the branch address from the MEM state
    - Could lead to control hazards
- Represent each instruction with its own datapath

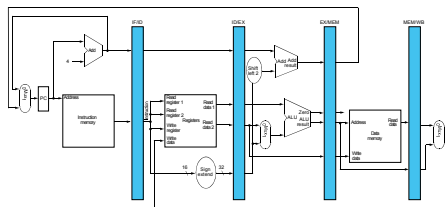
## Pipelined datapath



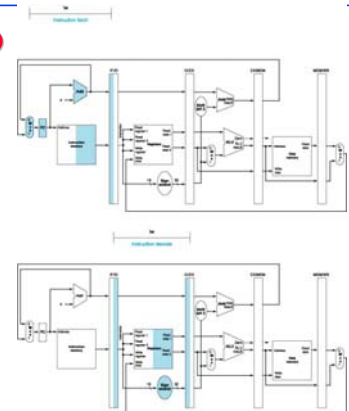
## Pipelined datapath

- Add registers to hold data to share multiple datapaths
- No register after write-back stage
- The PC can be thought as a pipeline register
  - It feeds the IF stage to the pipeline
  - Its content must be saved when an exception occurs

## Pipelined Datapath



## IF and ID



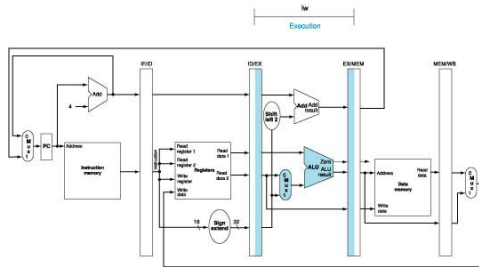
## Instruction Fetch

- ❑ The instruction is read from memory using the address in the PC and then placed to IF/ID register
- ❑ IF/ID register is similar to ?
- ❑ The PC is increased by 4 and written back to PC. Also saved in the IF/ID in case needed by a branch instruction (beq)

## Instruction Decode

- ❑ IF/ID pipeline register supplies
  - The 16-bit immediate field, which is sign-extended to 32 bits
  - Register numbers to read the two registers
- ❑ All three values and the incremented PC address are stored in the ID/EX pipeline register

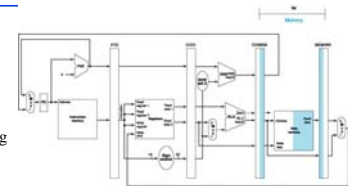
## Execute



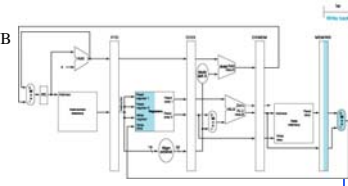
Read the contents of register 1 and the sign-extended immediate from ID/EX pipeline registers and adds them using ALU  
The sum is placed in EX/MEM pipeline register

## Memory access, Write back

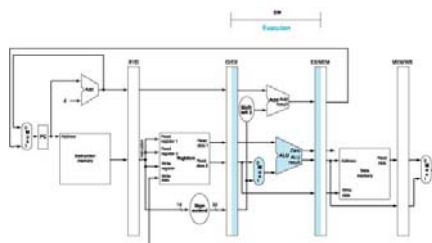
Read data from memory using the address from EX/MEM



Read the data from MEM/WB  
And write it to register file



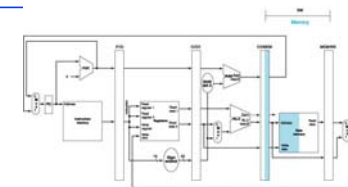
## Store



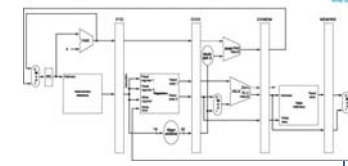
The effective address is placed in the EX/MEM pipeline register

## Memory Write back

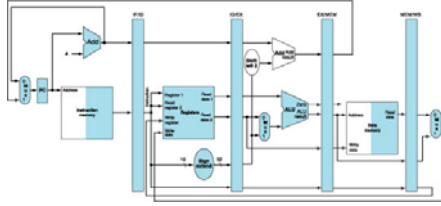
The register containing the data was stored in ID/EX in an earlier stage. The data was then stored in EX/MEM register



Nothing happens in WB

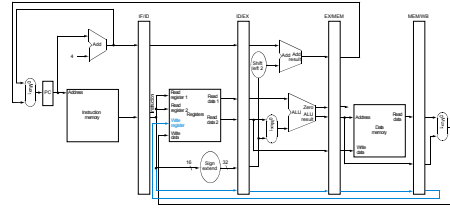


## Corrected Pipelined datapath for load

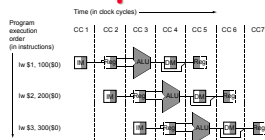


Load must pass the write register number from ID/EX through EX/MEM to the MEM/WB pipeline register for use in WB st

## Pipelined Datapath



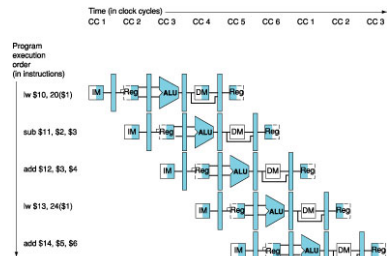
## Graphically Representing Pipelines



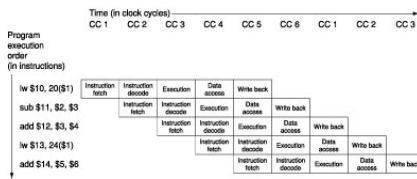
Multiple-clock-cycle pipeline diagrams

- Can help with answering questions like:
  - how many cycles does it take to execute this code?
  - what is the ALU doing during cycle 4?
  - use this representation to help understand datapaths

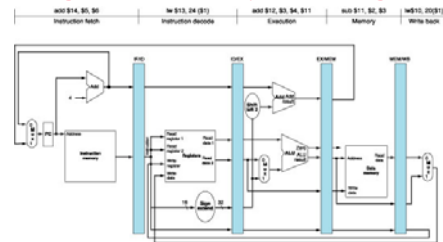
## Multiple-clock-cycle pipeline diagrams



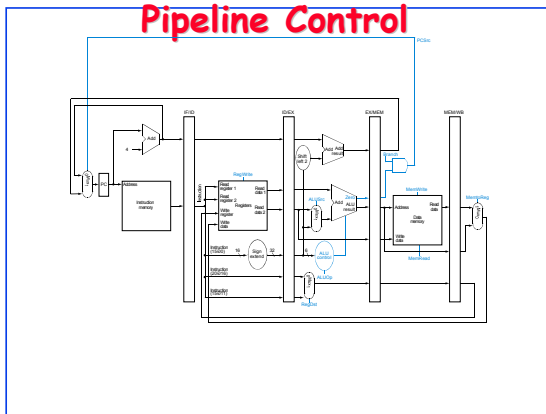
## Diagram



## Single-clock-cycle diagrams



Show the state of the entire datapath during a single clock cycle. It show the details of what is happening in that clock cycle.



### Pipeline control

- We have 5 stages. What needs to be controlled in each stage?
  - Instruction Fetch and PC Increment
  - Instruction Decode / Register Fetch
  - Execution
  - Memory Stage
  - Write Back
- How would control be handled in an automobile plant?
  - a fancy control center telling everyone what to do?
  - should we use a finite state machine?

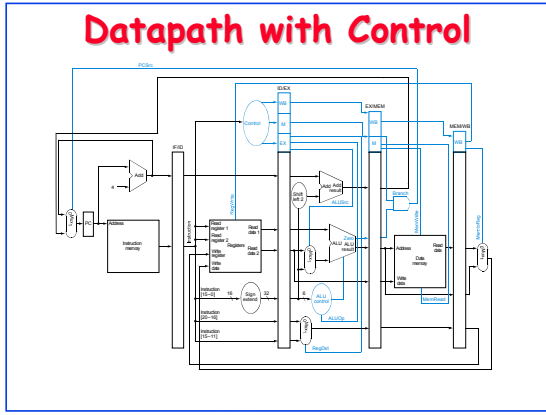
### Control

- No need for separate write signal for PC and pipeline registers
  - The PC is written on each clock
  - IF/ID, ID/EX, EX/MEM, and MEM/WC are written during each clock cycle
- Divide the control lines into five groups according to the pipeline stage

### Pipeline Control

- Pass control signals along just like the data

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			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
lwr	0	0	0	1	0	1	0	1	1
swr	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X



### Dependencies

- Problem with starting next instruction before first is finished
  - dependencies that "go backward in time" are data hazards

## Software Solution

- Have compiler guarantee no hazards
- Where do we insert the "nops" ?

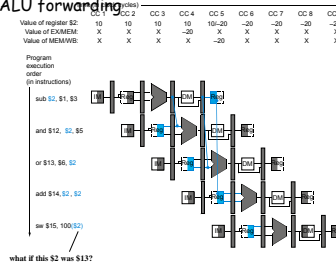
```

sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
    
```

- Problem: this really slows us down!

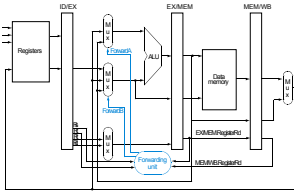
## Forwarding

- Use temporary results, don't wait for them to be written
  - register file forwarding to handle read/write to same register
  - ALU forwarding



## Forwarding

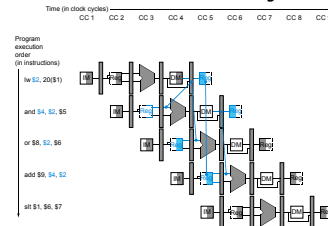
- The main idea (some details not shown)



Detect hazards  
Forward data. Add more multiplexors

## Can't always forward

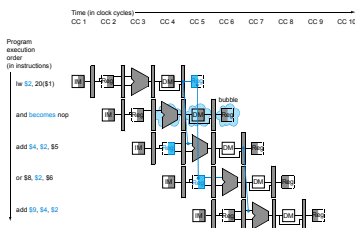
- Load word can still cause a hazard:
  - an instruction tries to read a register following a load instruction that writes to the same register.



- Thus, we need a hazard detection unit to "stall" the load instruction

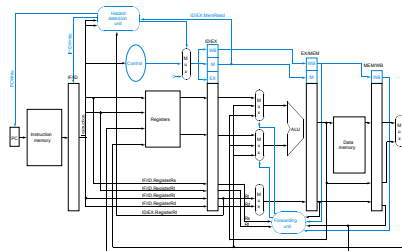
## Stalling

- We can stall the pipeline by keeping an instruction in the same stage



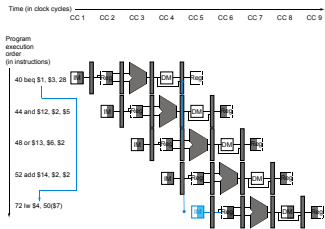
## Hazard Detection Unit

- Stall by letting an instruction that won't write anything go forward

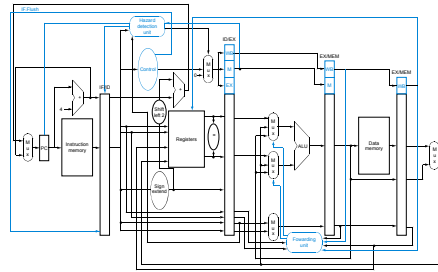


## Branch Hazards

- When we decide to branch, other instructions are in the pipeline!



## Flushing Instructions



Note: we've also moved branch decision to ID stage

## Reducing the Delay of Branches

- The next PC for a branch is selected in the MEM stage
- Many branches can be done without ALU
- Move the branch up - flush less instructions when the branch is taken
  - Computing the branch target address - We have the value in the IF/ID register
  - Evaluating the branch decision in ID
    - Branch equal - using gates
    - New forwarding logic is needed - formerly handled by ALU forwarding logic
    - Data hazards can occur - if one of the operands are produced by a preceding ALU (EX) - need to stall
    - Only the next instruction need to be flushed if the branch is taken
    - Use IF.Flush that zeros the instruction field in IF/ID register

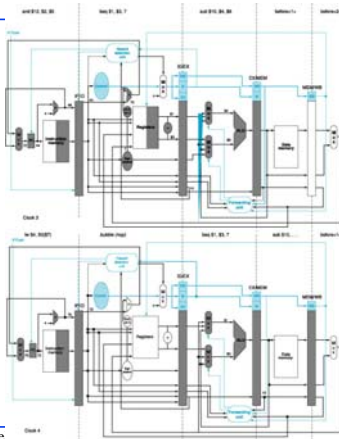
## Pipelined Branch

- What happens when the branch is taken

```

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, $6, $2
52
56
72 lw $4, 50($7)
    
```

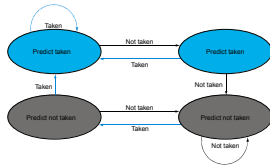
There us only one pipeline bubble on a taken branch



## Branches

- If the branch is taken, we have a penalty of one cycle
- For our simple design, this is reasonable
- With deeper pipelines, penalty increases and static branch prediction drastically hurts performance
- Solution: dynamic branch prediction
- Keep a branch history table - 1 bit that says whether the branch was recently taken or not
  - Considers a loop that branches nine time in a row, then is not taken once
  - Mispredicted on the first and last loop iteration - prediction 80%
- Use 2-bit prediction. A prediction must be wrong twice before changing. Mispredicted only once.

## Branches

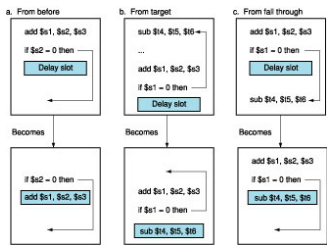


A 2-bit prediction scheme. Mispredicted only once

## Branch Prediction

- Sophisticated Techniques:
  - A "branch target buffer" to help us look up the destination
  - Correlating predictors that base prediction on global behavior and recently executed branches (e.g., prediction for a specific branch instruction based on what happened in previous branches)
  - Tournament predictors that use different types of prediction strategies and keep track of which one is performing best.
  - A "branch delay slot" which the compiler tries to fill with a useful instruction (make the one cycle delay part of the ISA)
- Branch prediction is especially important because it enables other more advanced pipelining techniques to be effective!
- Modern processors predict correctly 95% of the time!

## Scheduling the branch delay slot



a) The delay slot is scheduled with independent instruction from before the Branch. This is the best choice. In (b) the target instruction would need to be Copied, because might be reached by other paths. (b) is preferred when the branch is taken with high probability. In (b) and (c) it should be allowed to execute sub \$t4 when the branch goes in unexpected direction, \$t4 is not used later.

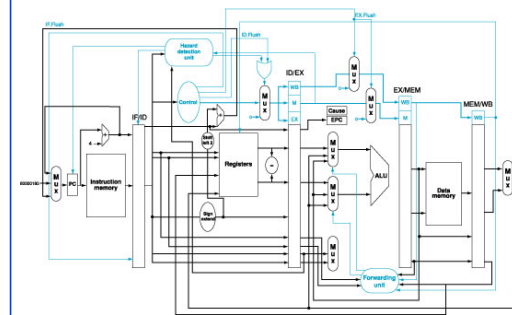
## Comparing Performance

- Compare: Single-cycle, multicycle and pipelined control using SPECint2000
- Single-cycle: memory access = 200ps, ALU = 100ps, register file read and write = 50ps
  - $200+50+100+200+50=600ps$
- Multicycle: 25% loads, 10% stores, 11% branches, 2% jumps, 52% ALU
  - CPI = 4.12, The clock cycle = 200ps (longest functional unit)
- Pipelined
  - 1 clock cycle when there is no load-use dependence
  - 2 when there is, average 1.5 per load
  - Stores and ALU take 1 clock cycle
  - Branches - 1 when predicted correctly and 2 when not, average 1.25
  - Jump - 2
  - $1.5 \times 25\% + 1 \times 10\% + 1 \times 52\% + 1.25 \times 11\% + 2 \times 2\% = 1.17$
- Average instruction time: single-cycle = 600ps, multicycle =  $4.12 \times 200 = 824$ , pipelined  $1.17 \times 200 = 234ps$
- Memory access 200ps is the bottleneck. How to improve?

## Exceptions

- Suppose add \$1, \$2, \$1 has an arithmetic overflow
- We need to transfer control to the exception routine
- We must flush the instructions that follow from the pipeline and begin fetching instructions from the new address
- We have to flush instructions in ID and EX states
  - ID.Flush is ORed with the stall signal from Hazard Detection unit
  - EX.Flush
- Add additional input to the PC multiplexor to fetch the instruction
- Save the offending instruction to the Exception Program Counter (EPC)

## Datapath to handle Exceptions



```

40 sub $t1,$2,$4
44 and $t2,$2,$5
48 or $t3,$2,$6
4C add $t, $2,$1
50 slt $t5,$6,$7
54 lw $t6,$5($7)

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

```

Louisiana State University 4- Pipelining - 67 CSC7080 SP06

## Exceptions

- ❑ Hardware and operating system must work in conjunction
- ❑ Hardware:
  - Stop the offending instruction in midstream,
  - Let all prior instruction complete
  - Flush all following instructions
  - Set the register to show the cause of the exception
  - Save the address of the offending instruction
  - Jump to a prearranged address
- ❑ Operating system
  - Look at the cause of exception and act appropriately

Louisiana State University 4- Pipelining - 68 CSC7080 SP06

## Improving Performance

- ❑ Try and avoid stalls! E.g., reorder these instructions:

```

lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)

```
- ❑ Dynamic Pipeline Scheduling
  - Hardware chooses which instructions to execute next
  - Will execute instructions out of order (e.g., doesn't wait for a dependency to be resolved, but rather keeps going!)
  - Speculates on branches and keeps the pipeline full (may need to rollback if prediction incorrect)
- ❑ Trying to exploit instruction-level parallelism

Louisiana State University 4- Pipelining - 69 CSC7080 SP06

## Improvements

- ❑ Increase the depth of pipeline to overlap more instructions. The amount of parallelism is higher.
- ❑ Replicate the internal components so that multiple instructions can be launched - multiple issue.
  - CPI less than 1. Use instructions per clock cycle IPC
  - 6GHz four multiple issue processor - at peak rate 24 billion instructions per second IPC = 4, CPI = 0.25. Assuming 5 stage pipeline - 2instructions are ins execution at any time.
  - Today's high end microprocessors attempt to issue from 3 to 8 instructions in every cycle

Louisiana State University 4- Pipelining - 70 CSC7080 SP06

## Multiple issue

- ❑ Depending how the work is divided between OS and hardware:
  - Static multiple issue
  - Dynamic multiple issue
- ❑ Two distinct responsibilities:
  - Packing instructions into issue slots
  - Dealing with data and control hazards
- ❑ Speculation: compiler or processor guesses the outcome of an instruction to remove it as a dependence in executing other instructions
  - Check guesses
  - Unroll or back out the effects
  - Speculations can introduce new exceptions

Louisiana State University 4- Pipelining - 71 CSC7080 SP06

Louisiana State University 4- Pipelining - 72 CSC7080 SP06

## Simple multiple issue coding

```

Loop: lw $t0,0($s1)
      addu $t0, $t0 $s2
      sw $t0, 0($s1)
      addi $s1, $s1, -4
      bne $s1, $zero, Loop

                                lw $t0,0($s1)      1
addi $s1, $s1, -4                2
addu $t0, $t0 $s2                3
bne $s1, $zero, Loop sw $t0, 0($s1) 4

5 instructions in 4 cycles, CPI = 0.8, IPC = 1.25

```

## Simple multiple issue coding

	ALU or branch Instructions	Data Transfer Instructions	
Loop:	addi \$s1, \$s1, -4	lw \$t0,0(\$s1)	1
		lw \$t1,12(\$s1)	2
	addu \$t0, \$t0 \$s2	lw \$t1,8(\$s1)	3
	addu \$t0, \$t0 \$s2	lw \$t1,8(\$s1)	4
		lw \$t0,0(\$s1)	1
	addi \$s1, \$s1, -4		2
	addu \$t0, \$t0 \$s2		3
	bne \$s1, \$zero, Loop	sw \$t0, 0(\$s1)	4

5 instructions in 4 cycles, CPI = 0.8, IPC = 1.25

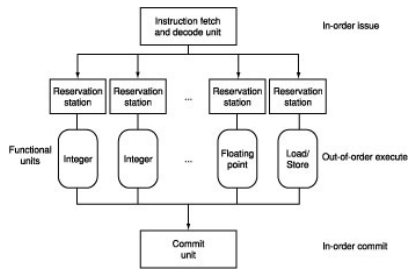
## Simple multiple issue coding

- Use loop unrolling:
  - Multiple copies of the loop are made and the instructions from different iterations are scheduled together
  - Use extra temporary registers
  - The previous example: 4 copies - CPI = 0.57

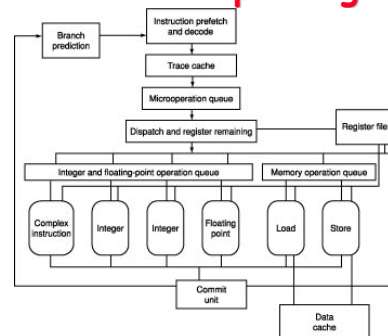
## Advanced Pipelining

- Increase the depth of the pipeline
- Start more than one instruction each cycle (multiple issue)
- Loop unrolling to expose more ILP (better scheduling)
- "Superscalar" processors
  - DEC Alpha 21264: 9 stage pipeline, 6 instruction issue
- All modern processors are superscalar and issue multiple instructions usually with some limitations (e.g., different "pipes")
- VLIW: very long instruction word, static multiple issue (relies more on compiler technology)

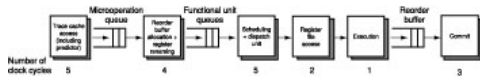
## Advanced Pipelining



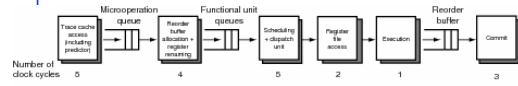
## Advanced Pipelining



# Advanced Pipelining



# Advanced Pipelining



# Summary



□ Pipelining does not improve latency, but does improve throughput

