



Csci 211 Computer System Architecture
 – Datapath and Control Design
 – Appendixes A & B

Xiuzhen Cheng
cheng@gwu.edu



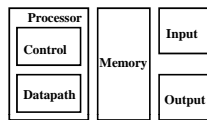
Outline

- Single Cycle Datapath and Control Design
- Pipelined Datapath and Control Design



The Big Picture

- The Five Classic Components of a Computer



- Performance of a machine is determined by:
 - Instruction count; Clock cycle time; Clock cycles per instruction
- Processor design (datapath and control) will determine:
 - Clock cycle time; Clock cycles per instruction
 - Who will determine Instruction Count?
 - Compiler, ISA

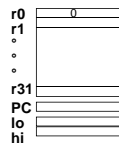


How to Design a Processor: Step by Step

1. Analyze instruction set => datapath requirements
 1. the meaning of each instruction is given by the register transfers
 2. datapath must include storage element for registers
 3. datapath must support each register transfer
 2. Select the set of datapath components and establish clocking methodology
 3. Assemble the datapath meeting the requirements
 4. Analyze the implementation of each instruction to determine the settings of the control points that effects the register transfer
 5. Assemble the control logic
- Use MIPS ISA to illustrate these five steps!



Example: MIPS



Programmable storage
 $2^{32} \times \text{bytes}$
 31 x 32-bit GPRs (R0=0)
 32 x 32-bit FP regs (paired DP)

Data types ?
Format ?
Addressing Modes?
Memory Addressing?

HI, LO, PC

Arithmetic logical

Add, AddU, Sub, SubU, And, Or, Xor, Nor, SLT, SLTU,
 Addi, AddIU, SLTI, SLTIU, Andl, Orl, Xorl, LUI
 SLL, SRL, SRA, SLLV, SRLV, SRAV

Memory Access

LB, LBU, LH, LHU, LW, LWL, LWLW
 SB, SH, SW, SWL, SWR

Control

J, JAL, JR, JALR

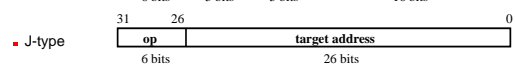
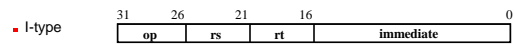
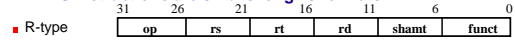
BEq, BNE, BLEZ, BGTZ, BLTZ, BGEZ, BLTZAL, BGEZAL

32-bit instructions on word boundary



MIPS Instruction Format

- All MIPS instructions are 32 bits long. 3 formats:



- The different fields are:

- op: operation ("opcode") of the instruction
- rs, rt, rd: the source and destination register specifiers
- shamt: shift amount
- funct: selects the variant of the operation in the "op" field
- address / immediate: address offset or immediate value
- target address: target address of jump instruction



MIPS Instruction Formats Summary

- Minimum number of instructions required
 - Information flow: load/store
 - Logic operations: logic and/or/not, shift
 - Arithmetic operations: addition, subtraction, etc.
 - Branch operations:
- Instructions have different number of operands: 1, 2, 3
- 32 bits representing a single instruction
- Disassembly is simple and starts by decoding opcode field.

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

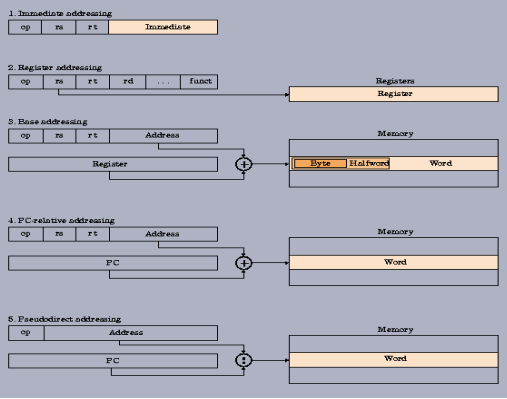


MIPS Addressing Modes

- Register addressing**
 - Operand is stored in a register. R-Type
- Base or displacement addressing**
 - Operand at the memory location specified by a register value plus a displacement given in the instruction. I-Type
 - Eg: lw, \$t0, 25(\$s0)
- Immediate addressing**
 - Operand is a constant within the instruction itself. I-Type
- PC-relative addressing**
 - The address is the sum of the PC and a constant in the instruction. I-Type
 - Eg: beq \$t2, \$t3, 25 # if (\$t2==\$t3), goto PC+4+100
- Pseudodirect addressing**
 - The 26-bit constant is logically shifted left 2 positions to get 28 bits. Then the upper 4 bits of PC+4 is concatenated with this 28 bits to get the new PC address. J-type, e. g., j 2500



MIPS Addressing Modes



MIPS Instruction Subset Core

- ADD and SUB**
 - addu rd, rs, rt
 - subu rd, rs, rt
- OR Immediate:**
 - ori rt, rs, imm16
- LOAD and STORE Word**
 - lw rt, rs, imm16
 - sw rt, rs, imm16
- BRANCH:**
 - beq rs, rt, imm16

inst	Register Transfers
ADDU	$R[rd] \leftarrow R[rs] + R[rt];$ $PC \leftarrow PC + 4$
SUBU	$R[rd] \leftarrow R[rs] - R[rt];$ $PC \leftarrow PC + 4$
ORI	$R[rt] \leftarrow R[rs] \mid \text{zero_ext}(\text{Imm16});$ $PC \leftarrow PC + 4$
LOAD	$R[rt] \leftarrow \text{MEM}[R[rs] + \text{sign_ext}(\text{Imm16})];$ $PC \leftarrow PC + 4$
STORE	$\text{MEM}[R[rs] + \text{sign_ext}(\text{Imm16})] \leftarrow R[rt];$ $PC \leftarrow PC + 4$
BEQ	if ($R[rs] == R[rt]$) then $PC \leftarrow PC + 4 + ([\text{sign_ext}(\text{Imm16})] \ll 2)$ else $PC \leftarrow PC + 4$

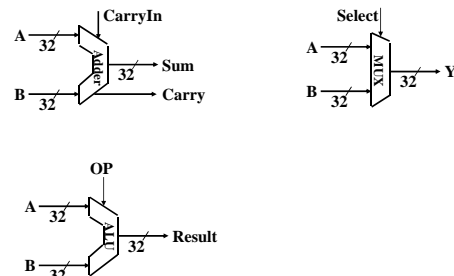


Step 1: Requirements of the Instruction Set

- Memory**
 - instruction & data: instruction=MEM[PC]
- Registers (32 x 32)**
 - read RS; read RT; Write RT or RD
- PC, what is the new PC?**
 - Add 4 or extended immediate to PC
- Extender: sign-extension or 0-extension?**
 - Add and Sub register or extended immediate



Step 2: Components of the Datapath



Storage Element: Register File

- Register File consists of 32 registers:
 - Two 32-bit output busses: busA and busB
 - One 32-bit input bus: busW
- Register is selected by:
 - RA (number) selects the register to put on busA (data)
 - RB (number) selects the register to put on busB (data)
 - RW (number) selects the register to be written via busW (data) when Write Enable is high
- Clock input (CLK)
 - The CLK input is a factor ONLY during write operation
 - During read operation, behaves as combinational logic:
 - RA or RB valid => busA or busB outputs valid after "access time."

Storage Element: Idealized Memory

- Memory (idealized)
 - One input bus: Data In
 - One output bus: Data Out
- Memory word is selected by:
 - Address selects the word to put on Data Out
 - Write Enable = 1: address selects the memory word to be written via the Data In bus
- Clock input (CLK)
 - The CLK input is a factor ONLY during write operation
 - During read operation, behaves as a combinational logic block:
 - Address valid => Data Out valid after "access time."

Step 3: Assemble DataPath meeting our requirements

- Instruction Fetch
 - Instruction = MEM[PC]
 - Update PC
- Read Operands and Execute Operation
 - Read one or two registers
 - Execute operation

Datapath for Instruction Fetch

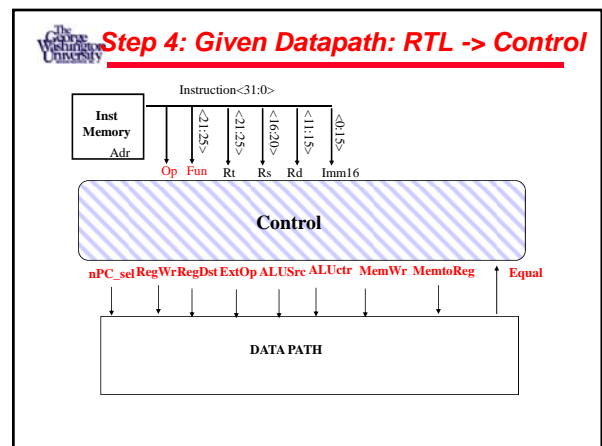
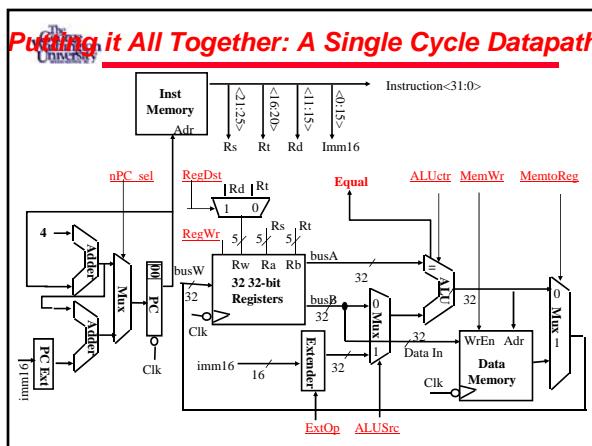
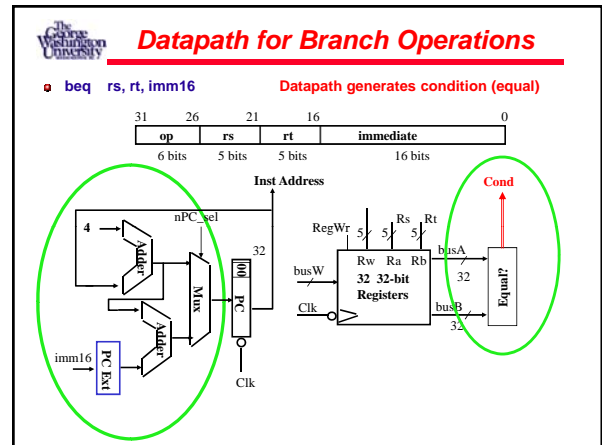
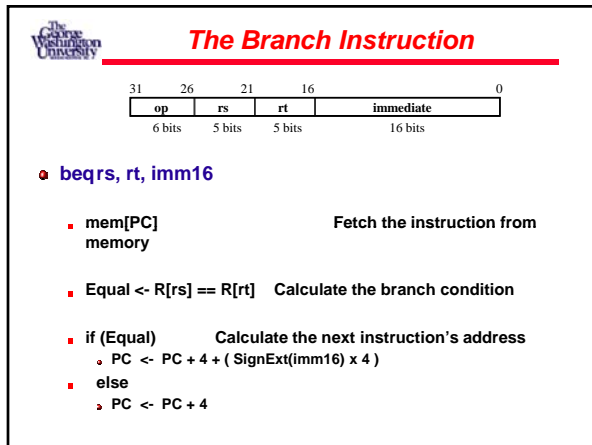
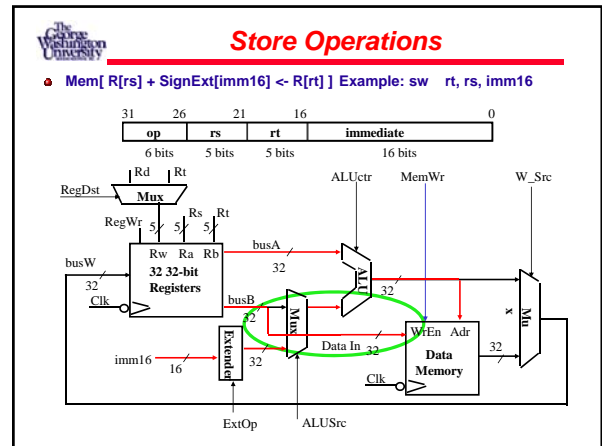
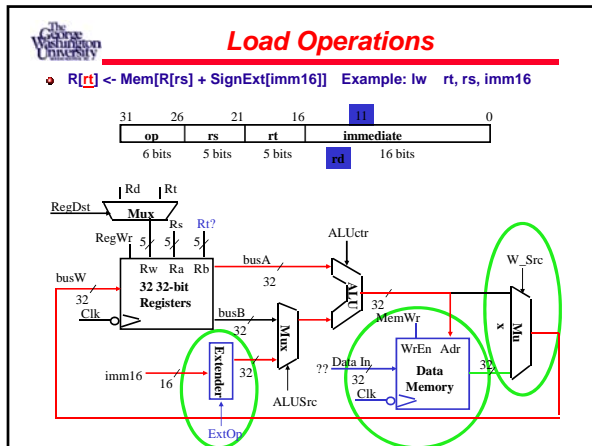
- Fetch the Instruction: mem[PC]
- Update the program counter:
 - Sequential Code: PC <- PC + 4
 - Branch and Jump: PC <- "something else"

Datapath for R-Type Instructions

- $R[rd] \leftarrow R[rs] \text{ op } R[rt]$ Example: addU rd, rs, rt
 - Ra, Rb, and Rw come from instruction's rs, rt, and rd fields
 - ALUctr and RegWr: control logic after decoding the instruction

Logic Operations with Immediate

$R[rt] \leftarrow R[rs] \text{ op } \text{ZeroExt}[\text{imm16}]$
 Eg. Ori \$7, \$8, 0x20



Meaning of the Control Signals

- Rs, Rt, Rd and lmed16 hardwired into datapath
- nPC_sel: 0 => PC ← PC + 4; 1 => PC ← PC + 4 + SignExt(lm16) || 00

Meaning of the Control Signals

- ExtOp: "zero", "sign"
- ALUSrc: 0 => regB; 1 => imm6
- ALUctr: "add", "sub", "or"
- MemWr: write memory
- MemtoReg: 1 => Mem
- RegDst: 0 => "rt"; 1 => "rd"
- RegWr: write dest register

Review on ALU Design

ALU Control Lines	Function
0000	And
0001	Or
0010	Add
0110	Subtraction
0111	Slt, beq
1100	NOR

ALU Control and the Central Control

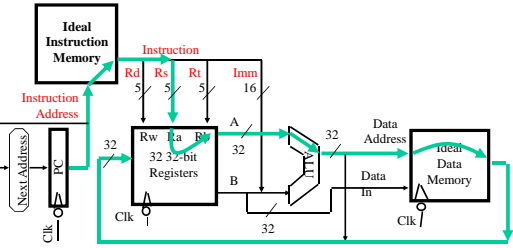
- Two-level design to ease the job
 - ALU Control generates the 4 control lines for ALU operation
 - Func code field is only effective for R-type instructions, whose Opcode field contains 0s.
 - The operation of I-type and J-type instructions is determined only by the 6 bit Opcode field.
 - Lw/sw and beq need ALU even though they are I-type instructions.
 - Three cases: address computation for lw/sw, comparison for beq, and R-Type; needs two control lines from the main control unit: ALUOp: 00 for lw/sw, 01 for beq, 10 for R-type
- Design ALU control
 - Input: the 6 bit func code field for R-type
 - Input: the 2 bit ALUOp from the main control unit.
- Design the main control unit
 - Input: the 6 bit Opcode field.

Step 5: Logic for each control signal

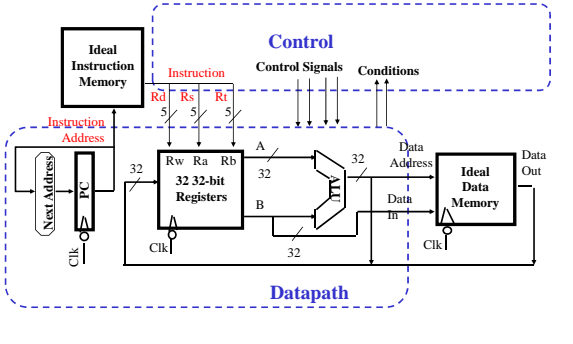
Step 5: Logic for each control signal

An Abstract View of the Critical Path

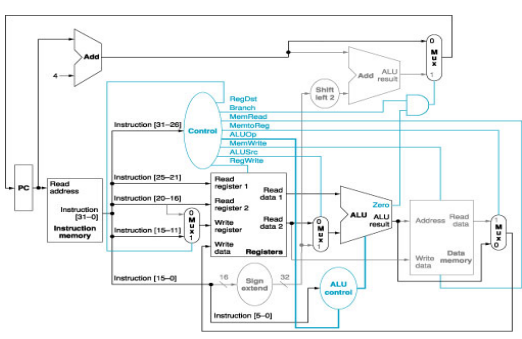
- Register file and ideal memory:
 - The CLK input is a factor ONLY during write operation
 - During read operation, behave as combinational logic:



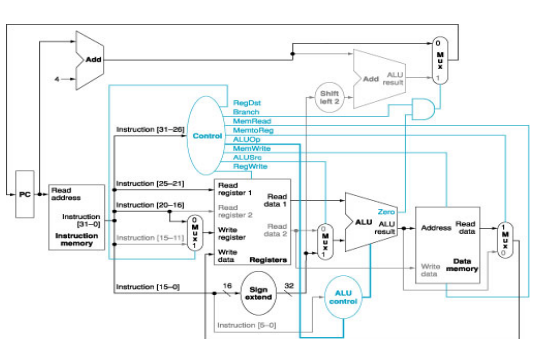
An Abstract View of the Implementation



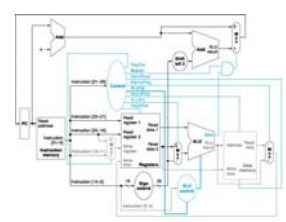
Example: R-type add \$t1, \$t2, \$t3



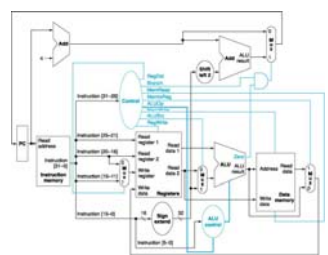
Example: lw

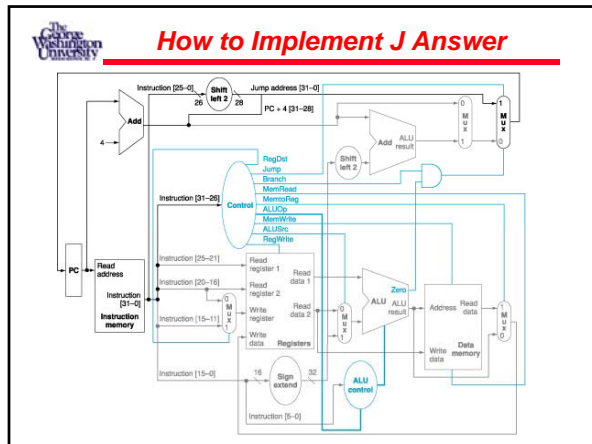


Example: beq



How to Implement jump Instruction?





- ### Performance of Single-Cycle Datapath
- **Time needs by functional units:**
 - Memory units: 200 ps
 - ALU and adders: 100 ps
 - Register file (r/w): 50 ps
 - No delay for other units
 - **Two single cycle datapath implementations**
 - Clock cycle time is the same for all instructions
 - Variable clock cycle time per instruction
 - **Instruction mix: 25% loads, 10% stores, 45% ALU, 15% branches, and 5% jumps**
 - **Compare the performance of R-type, lw, sw, branch, and j**

- ### Performance of Single-Cycle Datapath
- **Time needed per instruction:**
 - Variable clock cycle time datapath:
R: 400ps, lw: 600ps, sw: 550ps, branch: 350, j: 200
 - Same clock cycle time datapath: 600ps
 - **Average time needed per instruction**
 - With a variable clock: 447.5ps
 - With the same clock: 600ps
 - **Performance ratio:**
 - $600/447.5 = 1.34$

- ### Remarks on Single Cycle Datapath
- **Single Cycle Datapath ensures the execution of any instruction within one clock cycle**
 - Functional units must be duplicated if used multiple times by one instruction. E.g. ALU. *Why?*
 - Functional units can be shared if used by different instructions
 - **Single cycle datapath is not efficient in time**
 - Clock Cycle time is determined by the instruction taking the longest time. Eg. *lw* in MIPS
 - Variable clock cycle time is too complicated.
 - Multiple clock cycles per instruction
 - Pipelining

- ### Summary
- **5 steps to design a processor**
 1. Analyze instruction set => datapath requirements
 2. Select set of datapath components & establish clock methodology
 3. Assemble datapath meeting the requirements
 4. Analyze implementation of each instruction to determine setting of control points that affects the register transfer
 5. Assemble the control logic
 - **MIPS makes it easier**
 - Instructions same size
 - Source registers always in same place
 - Immediates same size, location
 - Operations always on registers/immediates
 - **Single cycle datapath => CPI=1, CCT => long**

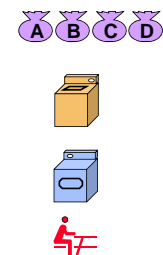
- ### Outline
- Single Cycle Datapath and Control Design
 - Pipelined Datapath and Control Design

Pipelining

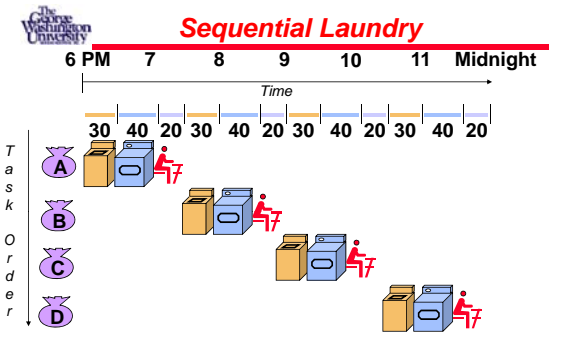
- Pipelining is an implementation technique in which multiple instructions are overlapped in execution
- Subset of MIPS instructions:
 - lw, sw, and, or, add, sub, slt, beq

Pipelining is Natural!

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- "Folder" takes 20 minutes

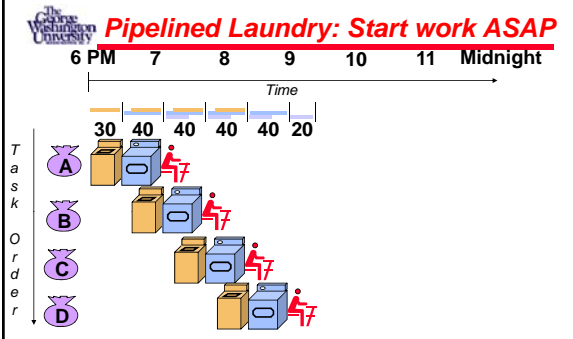


Sequential Laundry



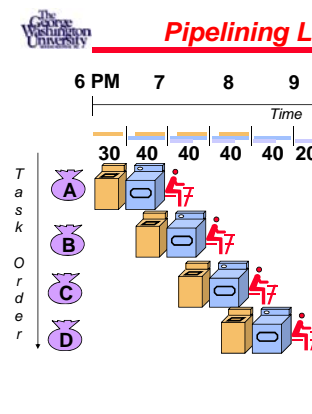
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry: Start work ASAP



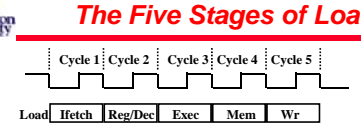
- Pipelined laundry takes 3.5 hours for 4 loads

Pipelining Lessons



- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate is limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipeline stages
- Unbalanced lengths of pipeline stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Stall for Dependencies

The Five Stages of Load



- Ifetch: Instruction Fetch
 - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- Wr: Write the data back to the register file

Pipelining

- **Improve performance by increasing throughput**

*Ideal speedup is number of stages in the pipeline.
Do we achieve this? NO!*

*The computer pipeline stage time are limited by the slowest resource, either the ALU operation, or the memory access
Fill and drain time*

Single Cycle, Multiple Cycle, vs. Pipeline

Single Cycle Implementation:

Load: 3 cycles, Store: 3 cycles, Write: 3 cycles

Multiple Cycle Implementation:

Load: 1 cycle, Reg: 1 cycle, Exec: 1 cycle, Mem: 1 cycle, Wr: 1 cycle, Store: 5 cycles, R-type: 2 cycles

Pipeline Implementation:

Load: 1 cycle, Reg: 1 cycle, Exec: 1 cycle, Mem: 1 cycle, Wr: 1 cycle

Why Pipeline?

- **Suppose we execute 100 instructions**
- **Single Cycle Machine**
 - 45 ns/cycle x 1 CPI x 100 inst = 4500 ns
- **Multicycle Machine**
 - 10 ns/cycle x 4.6 CPI (due to inst mix) x 100 inst = 4600 ns
- **Ideal pipelined machine**
 - 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns

Why Pipeline? Because the resources are there!

Can pipelining get us into trouble?

- **Yes: Pipeline Hazards**
 - **Structural hazards:** attempt to use the same resource two different ways at the same time
 - E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)
 - Single memory cause structural hazards
 - **Data hazards:** attempt to use item before it is ready
 - E.g., one sock of pair in dryer and one in washer; can't fold until you get sock from washer through dryer
 - instruction depends on result of prior instruction still in the pipeline
 - **Control hazards:** attempt to make a decision before condition is evaluated
 - E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in
 - branch instructions
- **Can always resolve hazards by waiting**
 - pipeline control must detect the hazard
 - take action (or delay action) to resolve hazards

Slow Down From Stalls

- Perfect pipelining with no hazards → an instruction completes every cycle (total cycles ~ num instructions) → speedup = increase in clock speed = num pipeline stages
- With hazards and stalls, some cycles (= stall time) go by during which no instruction completes, and then the stalled instruction completes
- Total cycles = number of instructions + stall cycles
- Slowdown because of stalls = $1 / (1 + \text{stall cycles per instr})$



Speed Up Equation for Pipelining

Compared to unpipelined,

$$CPI_{\text{pipelined}} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

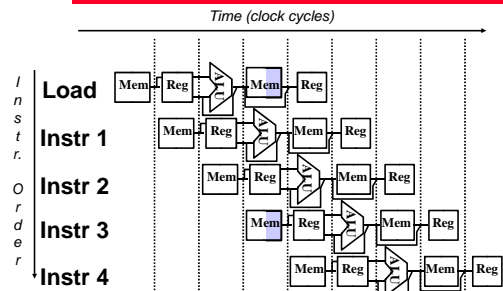
$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

For simple RISC pipeline, CPI = 1:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$



Single Memory is a Structural Hazard



Detection is easy in this case! (right half highlight means read, left half write)



Structural Hazards limit performance

- Example: if 1.3 memory accesses per instruction and only one memory access per cycle then
 - average CPI ≥ 1.3
 - otherwise resource is more than 100% utilized

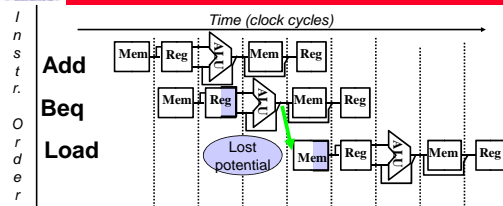


Example: Dual-port vs. Single-port

- Machine A: Dual ported memory ("Harvard Architecture")
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed
 - $\text{SpeedUp}_A = \text{Pipeline Depth} / (1 + 0) \times (\text{clock}_{\text{unpipe}} / \text{clock}_{\text{pipe}})$
 - = Pipeline Depth
 - $\text{SpeedUp}_B = \text{Pipeline Depth} / (1 + 0.4 \times 1) \times (\text{clock}_{\text{unpipe}} / (\text{clock}_{\text{unpipe}} / 1.05))$
 - = (Pipeline Depth / 1.4) \times 1.05
 - = 0.75 \times Pipeline Depth
 - $\text{SpeedUp}_A / \text{SpeedUp}_B = \text{Pipeline Depth} / (0.75 \times \text{Pipeline Depth}) = 1.33$
- Machine A is 1.33 times faster



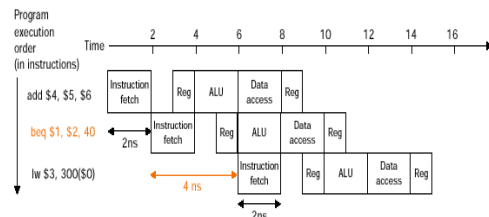
Control Hazard Solution #1: Stall



- Stall: wait until decision is clear
- Impact: 2 lost cycles (i.e. 3 clock cycles per branch instruction) => slow
- Move decision to end of decode by improving hardware
 - save 1 cycle per branch
- If 20% instructions are BEQ, all others have CPI 1, what is the average CPI?



Control Hazard Solution #1: Stall



Control Hazard Solution #2: Predict

Instruction Order: Add, Beq, Load

- ◆ **Predict:** guess one direction then back up if wrong
- ◆ **Impact:** 0 lost cycles per branch instruction if right, 1 if wrong (right - 50% of time)
 - ◆ Need to "Squash" and restart following instruction if wrong
 - ◆ Produce CPI on branch of $(1 * .5 + 2 * .5) = 1.5$
 - ◆ Total CPI might then be: $1.5 * .2 + 1 * .8 = 1.1$ (20% branch)
- ◆ **More dynamic scheme:** history of each branch (- 90%)

Control Hazard Solution #2: Predict

Program execution order (in instructions): add \$4, \$5, \$6; beq \$1, \$2, 40; lw \$3, 300(\$0)

Program execution order (in instructions): add \$4, \$5, \$6; beq \$1, \$2, 40; or \$7, \$8, \$9

Control Hazard Solution #3: Delayed Branch

Instruction Order: Add, Beq, Misc, Load

- ◆ **Delayed Branch:** Redefine branch behavior (takes place after next instruction)
- ◆ **Impact:** 0 extra clock cycles per branch instruction if can find instruction to put in "slot" (- 50% of time)
- ◆ **The longer the pipeline, the harder to fill**
- ◆ **Used by MIPS architecture**

Control Hazard Solution #3: Delayed Branch

Program execution order (in instructions): beq \$1, \$2, 40; add \$4, \$5, \$6 (Delayed branch slot); lw \$3, 300(\$0)

Scheduling Branch Delay Slots (Fig A.14)

A. From before branch
 add \$1, \$2, \$3
 if \$2=0 then
 delay slot
 becomes
 if \$2=0 then
 add \$1, \$2, \$3

B. From branch target
 sub \$4, \$5, \$6
 add \$1, \$2, \$3
 if \$1=0 then
 delay slot
 becomes
 add \$1, \$2, \$3
 sub \$4, \$5, \$6

C. From fall through
 add \$1, \$2, \$3
 if \$1=0 then
 delay slot
 becomes
 add \$1, \$2, \$3
 if \$1=0 then
 sub \$4, \$5, \$6

- ◆ **A is the best choice, fills delay slot & reduces instruction count (IC)**
- ◆ **In B, the sub instruction may need to be copied, increasing IC**
- ◆ **In B and C, must be okay to execute sub when branch fails**

More On Delayed Branch

- ◆ **Compiler effectiveness for single branch delay slot:**
 - ◆ Fills about 60% of branch delay slots
 - ◆ About 80% of instructions executed in branch delay slots useful in computation
 - ◆ About 50% (60% x 80%) of slots usefully filled
- ◆ **Delayed Branch downside: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot**
 - ◆ Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
 - ◆ Growth in available transistors has made dynamic approaches relatively cheaper



Evaluating Branch Alternatives

A simplified pipeline speedup equation for Branch:

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Assume 4% unconditional branch, 6% conditional branch-untaken, 10% conditional branch-taken

Scheduling scheme	Branch penalty	CPI	speedup v. unpipelined	speedup v. stall
Stall pipeline	3	1.60	3.1	1.0
Predict taken	1	1.20	4.2	1.33
Predict not taken	1	1.14	4.4	1.40
Delayed branch	0.5	1.10	4.5	1.45

* Branch penalty resulted from decision making and/or address computation

* Predict taken: still needs one cycle to compute address



Branch Stall Impact

Two part solution:

- Determine branch taken or not sooner, AND
- Compute taken branch address earlier

MIPS branch tests if register = 0 or ≠ 0

MIPS Solution:

- Move Zero test to ID/RF stage
- Adder to calculate new PC in ID/RF stage
- 1 clock cycle penalty for branch versus 3



Data Hazard on r1

An instruction depends on the result of a previous instruction still in the pipeline

```

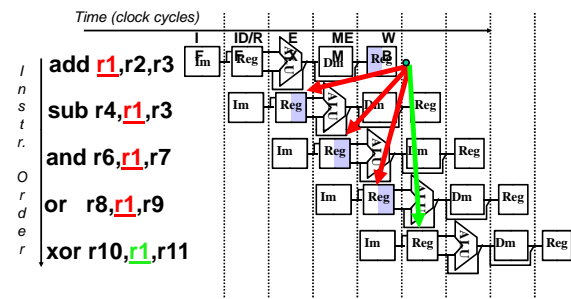
add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or r8,r1,r9
xor r10,r1,r11

```



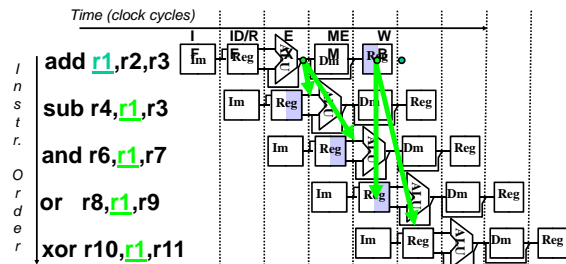
Data Hazard on r1:

- Dependencies backwards in time are hazards



Data Hazard Solution:

- “Forward” result from one stage to another



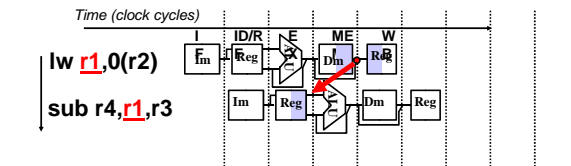
- “or” OK if define read/write properly

- Forwarding can't prevent all data hazard! – lw followed by R-type?



Forwarding (or Bypassing): What about Loads?

- Dependencies backwards in time are hazards



- Can't solve with forwarding:
- Must delay/stall instruction dependent on loads

Forwarding (or Bypassing): What about Loads

- Dependencies backwards in time are hazards

Time (clock cycles)

- Can't solve with forwarding:
- Must delay/stall instruction dependent on loads

Software Scheduling to Avoid Load Hazards

Try producing fast code for

$$a = b + c;$$

$$d = e - f;$$

assuming a, b, c, d, e, and f in memory.

Slow code:

```
LW Rb,b
LW Rc,c
ADD Ra,Rb,Rc
SW a,Ra
LW Re,e
LW Rf,f
SUB Rd,Re,Rf
SW d,Rd
```

Fast code:

```
LW Rb,b
LW Rc,c
LW Re,e
LW Rf,f
ADD Ra,Rb,Rc
SUB Rd,Re,Rf
SW a,Ra
SW d,Rd
```

Compiler optimizes for performance. Hardware checks for safety.

Extending to Multicycle Instructions

Latency is defined to be the number of intervening cycles between an instruction that produces a result and an instruction that uses the result.

The initiation or repeat interval is the number of cycles that must elapse between issuing two operations of a given type

Functional unit	Delay (Latency)	Initiation interval
Integer ALU	1 (0)	1
Data memory	2 (1)	1
FP add	4 (3)	1
FP multiply	7 (6)	1
FP divide	25 (24)	25

Effects of Multicycle Instructions

- Structural hazards if the unit is not fully pipelined (divider)
- Frequent Read-After-Write hazard stalls
- Potentially multiple writes to the register file in a cycle
- Write-After-Write hazards because of out-of-order instr completion
- Imprecise exceptions because of o-o-o instr completion

Note: Can also increase the "width" of the processor: handle multiple instructions at the same time: for example, fetch two instructions, read registers for both, execute both, etc.

Precise Exceptions

- On an exception:
 - must save PC of instruction where program must resume
 - all instructions after that PC that might be in the pipeline must be converted to NOPs (other instructions continue to execute and may raise exceptions of their own)
 - temporary program state not in memory (in other words, registers) has to be stored in memory
 - potential problems if a later instruction has already modified memory or registers
- A processor that fulfills all the above conditions is said to provide precise exceptions (useful for debugging and of course, correctness)

Imprecise Exceptions

- An exception is imprecise if the processor state when an exception is raised does not look exactly as if the instrs. Were executed sequentially in strict program order
 - The pipeline may have already completed instructions that are later in program order than the instruction causing the exception
 - The pipeline may have not yet completed some instructions that are earlier than the one causing the exception
- Example:


```
DIV.D F0, F2, F4
ADD.D F10, F10, F8
SUB.D F12, F12, F14
```

 - Imprecise exception appears when ADD and SUB have completed while DIV raises an exception
 - ADD and SUB have modified registers already!



Dealing With These Effects

- Multiple writes to the register file: increase the number of ports; stall one of the writers during ID; stall one of the writers during WB (the stall will propagate)
- WAW hazards: detect the hazard during ID and stall the later instruction
- Imprecise exceptions: buffer the results if they complete early or save more pipeline state so that you can return to exactly the same state that you left at



Summary: Pipelining

- **What makes it easy**
 - all instructions are the same length
 - just a few instruction formats
 - memory operands appear only in loads and stores; Memory addresses are assigned
- **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 talk about modern processors and what really makes it hard:**
 - trying to improve performance with out-of-order execution, etc.



Summary & Questions

- **Pipelining is a fundamental concept**
 - multiple steps using distinct resources
- **Utilize capabilities of the Datapath by pipelined instruction processing**
 - start next instruction while working on the current one
 - limited by length of longest stage (plus fill/flush)
 - detect and resolve hazards
- **Questions?**