CS 211: Computer Architecture

Instructor: Prof. Bhagi Narahari
Dept. of Computer Science
Course URL: www.seas.gwu.edu/~narahari/cs211/

Summary: Architecture Trends ?

- Moore's law: density doubles every 18-24 months
  - smaller processors, faster clocks
  - Price drops due to volume and dev. costs what next?
- Interconnect delays could dominate over feature delay
  - Need for simpler architectures
  - Distributed logic and control
- More functionality
  - communicating processors
  - network of embedded processors
- To extract max performance
  - Thumb rules: Amdahl's law, Parallelism, Locality
  - Software and compiler support needed!!!

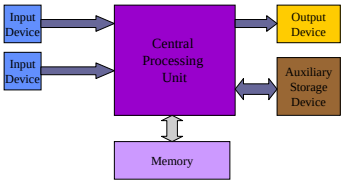
Next: Review Computer Organization in an hour!

- Overview of Computer Organization
  - Components
  - Sample processor design process

Review: Computer Organization Basics

- What are the components of a CPU
- What is the microarchitecture level ?
- What is an ISA - Instruction set architecture ?
- How does a sample processor design look ?
  - A simple processor architecture
- what is the basic concept of pipelining

A Computer

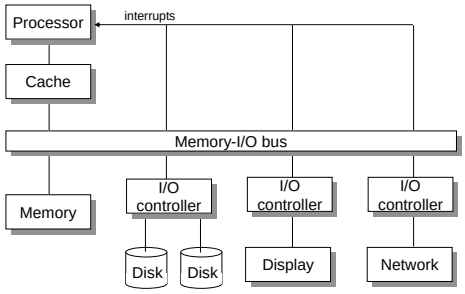


The computer is composed of input devices, a central processing unit, a memory unit and output devices.

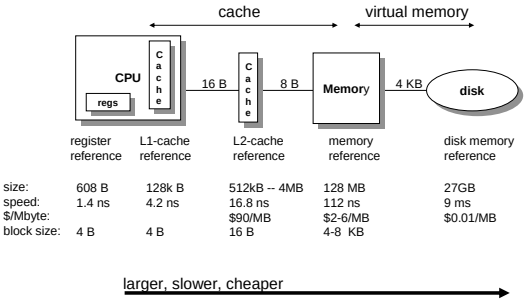
Memory Unit

- An ordered sequence of storage cells, each capable of holding a piece of data.
- Volatile Memory
  - RAM – Random Access Memory
- Non-volatile Memory
  - ROM – Read Only Memory

Computer System



Memory Hierarchy: The Tradeoff



(Numbers are for a 21264 at 700MHz)

Central Processing Unit (CPU)

- The CPU has two components:
  - Arithmetic and Logic Unit (ALU)
    - Performs arithmetic operations
    - Performs logical operations
  - Control Unit
    - Controls the action of the other computer components so that instructions are executed in the correct sequence
- Note: we will get back to Memory design after covering processor design

Input/Output (I/O) Devices

- I/O devices are the components of a computer system that accepts data to be processed and presents the results of the processing
- Input Device Examples
  - Keyboard
  - Mouse
- Output Device Examples
  - Video display
  - Printer
- We won't touch on this in this course!

The Operating System

- The Operating System (OS) is a set of programs that manages all of the computer's resources
- Unix, Linux, Windows 98, Me, NT, 2000, XP, and MacOS are all examples of modern operating systems
- Not the focus of this course!

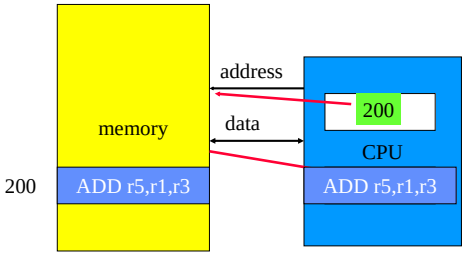
Computer Architecture

- The computer architecture encompasses the user's view of the computer.
- This includes such things as the assembly language instruction set, the number and types of internal registers, the memory management system and the model for exception handling.

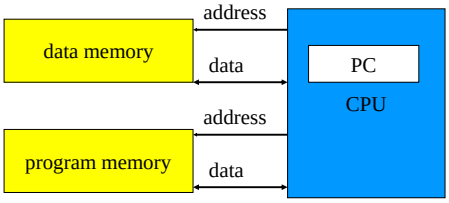
Architecture Models: Von Neumann architecture

- Memory holds data, instructions.
- Central processing unit (CPU) fetches instructions from memory.
  - Separate CPU and memory distinguishes programmable computer.
- CPU registers help out: program counter (PC), instruction register (IR), general-purpose registers, etc.

CPU + memory



Harvard architecture



von Neumann vs. Harvard

- Harvard can't use self-modifying code.
- Harvard allows two simultaneous memory fetches.
- Most DSPs use Harvard architecture for streaming data:
  - greater memory bandwidth;
  - more predictable bandwidth.

Instruction Set Architecture

- The Instruction Set Architecture (ISA) describes a set of instructions whose syntactic and semantic characteristics are defined by the underlying computer architecture.

Programming model

- Programming model: registers visible to the programmer.
- Some registers are not visible (IR).

Multiple implementations

- Successful architectures have several implementations:
  - varying clock speeds;
  - different bus widths;
  - different cache sizes;
  - etc.

Assembly language

- One-to-one with instructions (more or less).
- Basic features:
  - One instruction per line.
  - Labels provide names for addresses (usually in first column).
  - Instructions often start in later columns.
  - Columns run to end of line.

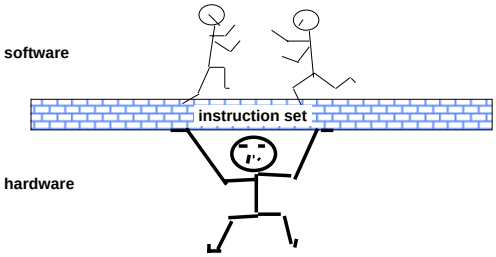
Computer Architecture is ...

Instruction Set Architecture

Organization

Hardware

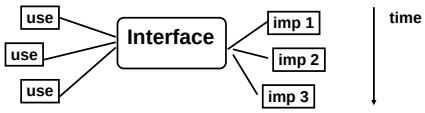
Instruction Set Architecture (ISA)



Interface Design

A good interface:

- Lasts through many implementations (portability, compatability)
- Is used in many differeny ways (generality)
- Provides convenient functionality to higher levels
- Permits an efficient implementation at lower levels



Evolution of Instruction Sets

Single Accumulator (EDSAC 1950)

Accumulator + Index Registers
(Manchester Mark I, IBM 700 series 1953)

Separation of Programming Model from Implementation

High-level Language Based
(B5000 1963)

Concept of a Family
(IBM 360 1964)

General Purpose Register Machines

Complex Instruction Sets
(Vax, Intel 432 1977-80)

Load/Store Architecture
(CDC 6600, Cray 1 1963-76)

RISC
(Mips,Sparc,HP-PA,IBM RS6000,PowerPC . . .1987)

VLIW/"EPIC"?(IA-64. . .1999)

Evolution of Instruction Sets

- Major advances in computer architecture are typically associated with landmark instruction set designs
  - Ex: Stack vs GPR (System 360)
- Design decisions must take into account:
  - technology
  - machine organization
  - programming languages
  - compiler technology
  - operating systems
  - applications
- And they in turn influence these

CISC vs. RISC

- Complex instruction set computer (CISC):
  - many addressing modes;
  - many operations.
- Reduced instruction set computer (RISC):
  - load/store;
  - pipelined instructions.

CISC Processors

- Instruction decoding is performed with large microcode ROMs
- Some instructions require more than a single instruction cycle to execute
- Many addressing modes supported
- Register set was designed to support specific functions

RISC Processors

- Instruction decoding is performed with static (hard-wired) logic for a much faster result
- Instructions are designed to execute in a single instruction cycle
- Data processing instructions operate only on registers. Load and store instructions were designated to access memory
- Register set is large and general purpose (in many cases)

IA - 32

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: 57 new "MMX" instructions are added, Pentium II
- 1999: The Pentium III added another 70 instructions (SSE)
- 2001: Another 144 instructions (SSE2)
- 2003: AMD extends the architecture to increase address space to 64 bits, widens all registers to 64 bits and other changes (AMD64)
- 2004: Intel capitulates and embraces AMD64 (calls it EM64T) and adds more media extensions

- *"This history illustrates the impact of the "golden handcuffs" of compatibility*

  -"adding new features as someone might add clothing to a packed bag"

  -"an architecture that is difficult to explain and impossible to love"

IA-32 Overview

- **Complexity:**
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    e.g., "base or scaled index with 8 or 32 bit displacement"
- **Saving grace:**
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

"what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective"

Quick look at ISA

- **Will use MIPS**
  - Simple RISC ISA
  - Widely used

Instruction set characteristics

- **Fixed vs. variable length.**
- **Addressing modes.**
- **Number of operands.**
- **Types of operands.**

A "Typical" RISC - MIPS

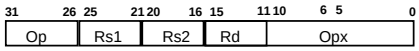
- fixed format instruction (3 formats I,R,J):
  - 32 or 64 bits
- 32 General Purpose Registers (GPR)
  - (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store: base + displacement
  - no indirection
- Simple branch conditions (based on register values)

see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

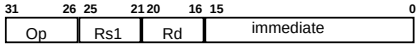
- Delayed branch

Example: 32 bit MIPS

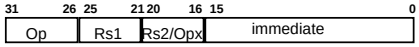
R-type: Register-Register



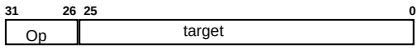
I-type: Register-Immediate (Load,Store)



I-type: Branch



J-type: Jump / Call



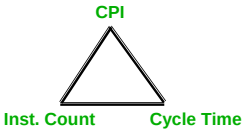
Instruction Types

- When is memory accessed ?
  - How is address computed ?
- When is control flow affected ?
  - How is branch outcome computed
  - How is branch target address computed

Processor Design. . .

- Let's glance at processor and inst. Set design.
  - This is review material…from a typical course on Computer Organization (pre-req)

The Big Picture: The Performance Perspective



- 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

Microarchitecture Design: How ?

- Any design must attempt to meet the requirements
  - Where do the requirements come from ?
  - Ex: need to represent numbers in binary; integers, text, floating point
- How to proceed with design ?

Some History…

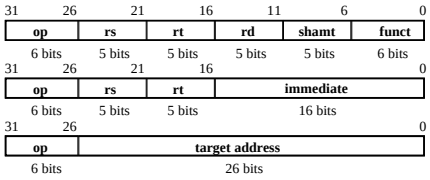
- The Indiana Legislature once introduced legislation declaring that the value of π was exactly 3.2

How to Design a Processor: step-by-step

- 1. Analyze instruction set => datapath requirements
  - the meaning of each instruction is given by the *register transfers*
  - datapath must include storage element for ISA registers
    - possibly more
  - datapath must support each register transfer
- 2. Select set of datapath components and establish clocking methodology
- 3. Assemble datapath meeting the requirements
- 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
- 5. Assemble the control logic
- Let's look at a single cycle ISA…

The MIPS Instruction Formats

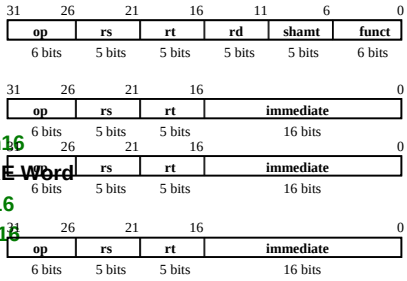
- All MIPS instructions are 32 bits long. The three instruction formats:



  - R-type
  - I-type
  - J-type

- The different fields are:
  - op: operation 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 the jump instruction

Step 1a: The MIPS-Inst Set (eg.)



- 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

- Register rs and rt are the source registers.
- If the instruction has three operand register, then rd is the destination register
- If the instruction has two operand register, then rt is the destination register

Logical Register Transfers

- RTL gives the meaning of the instructions
- All start by fetching the instruction

op | rs | rt | rd | shamt | funct = MEM[PC]

op | rs | rt | Imm16 = MEM[PC]

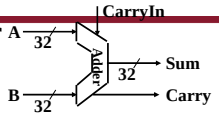
inst	Register Transfers	
ADDU	R[rd] <– R[rs] + R[rt];	PC <– PC + 4
SUBU	R[rd] <– R[rs] – R[rt];	PC <– PC + 4
ORi	R[rt] <– R[rs] | zero_ext(Imm16);	PC <– PC + 4
LOAD	R[rt] <– MEM[R[rs] + sign_ext(Imm16)];	PC <– PC + 4
STORE	MEM[R[rs] + sign_ext(Imm16)] <– R[rt];	PC <– PC + 4
BEQ	if (R[rs] == R[rt]) then PC <– PC + 4 + sign_ext(Imm16)] || 00 else PC <– PC + 4	

Step 2: Components of the Datapath

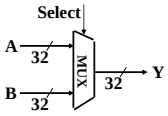
- Combinational Elements
- Storage Elements
  - Clocking methodology

Combinational Logic Elements (Basic Building Blocks)

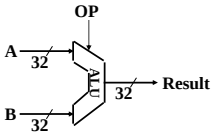
- **Adder**



- **MUX**

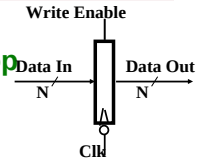


- **ALU**



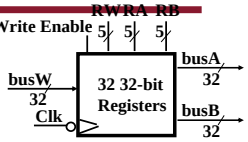
Storage Element: Register (Basic Building Block)

- **Register**
  - **Similar to the D Flip Flop except**
    - N-bit input and output
    - Write Enable input
  - **Write Enable:**
    - negated (0): Data Out will not change
    - asserted (1): Data Out will become Data In



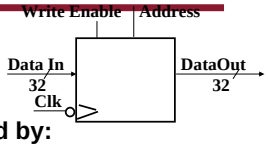
Storage Element: Register File

- 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 1
- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid => busA or busB valid after "access time."

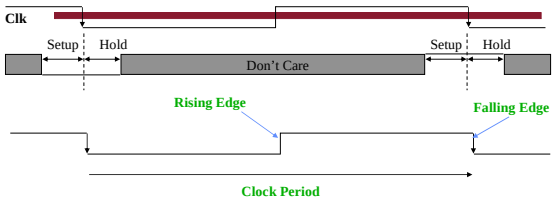


Storage Element: Idealized Memory

- **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."



Clocking Methodology



- Clocks needed in sequential logic to decide when an element that contains state should be updated.
- A clock is a free-running circuit with a fixed cycle time or clock period. The clock frequency is the inverse of the cycle time.
- The clock cycle time or clock period is divided into two portions: when the clock is high and when the clock is low.
- Edge-triggered clocking: all state changes occur on a clock edge.

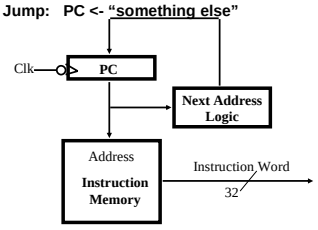
Step 3: Assemble DataPath meeting our requirements

- Register Transfer Requirements
  ⇒ Datapath Assembly
- Instruction Fetch
- Read Operands and Execute Operation

The common RTL operations for all instructions are:
(a) Fetch the instruction using the Program Counter (PC) at the beginning of an instruction's execution (PC -> Instruction Memory -> Instruction Word).
(b) Then at the end of the instruction's execution, you need to update the Program Counter (PC -> Next Address Logic -> PC).
More specifically, you need to increment the PC by 4 if you are executing sequential code. For Branch and Jump instructions, you need to update the program counter to "something else" other than plus 4.
The Next Address Logic block:
- Add 4 (number of bytes in an instruction) or
- Branch and Jump instructions

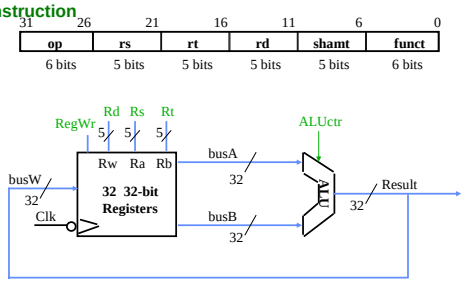
3a: Overview of the Instruction Fetch Unit

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

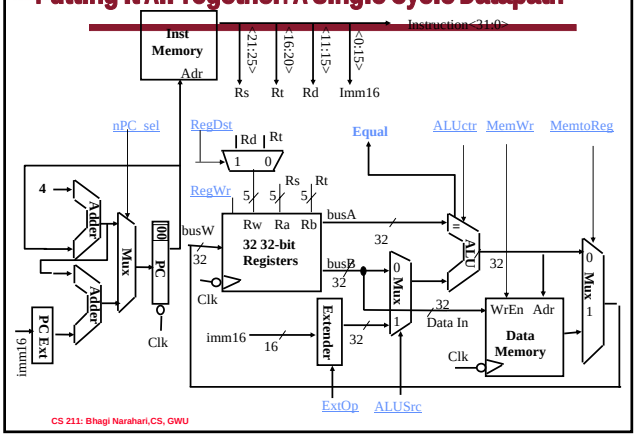


3b: Add & Subtract

- R[rd] <- R[rs] 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

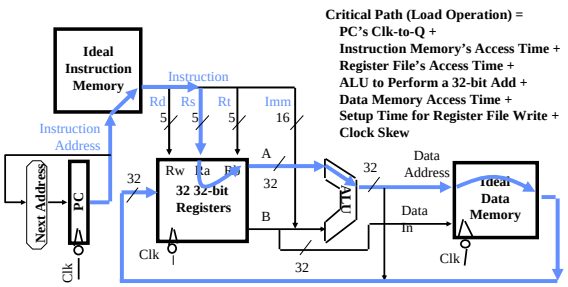


Putting it All Together: A Single Cycle Datapath



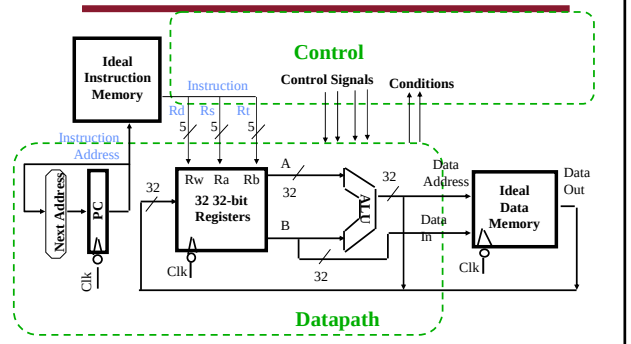
An Abstract View of the Critical Path

- **Register file and ideal memory:**
  - The CLK input is a factor ONLY during write operation
  - During read operation, behave as combinational logic:
    - Address valid => Output valid after "access time."

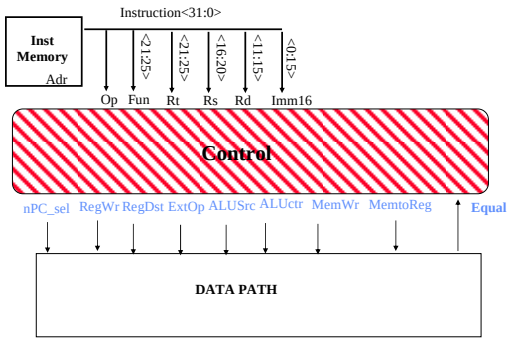


Critical Path (Load Operation) =
PC's Clk-to-Q +
Instruction Memory's Access Time +
Register File's Access Time +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew

An Abstract View of the Implementation



Step 4: Given Datapath: RTL -> Control



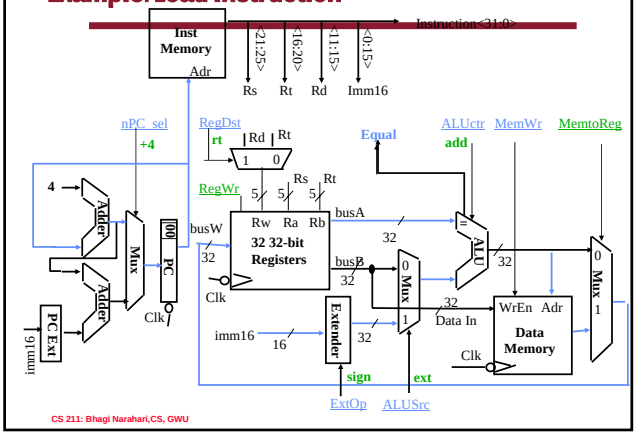
Control Signals

inst	Register Transfer	
ADD	R[rd] <– R[rs] + R[rt];	PC <– PC + 4
	ALUsrc = RegB, ALUctr = “add”, RegDst = rd, RegWr, nPC_sel = “+4”	
SUB	R[rd] <– R[rs] – R[rt];	PC <– PC + 4
	ALUsrc = RegB, ALUctr = “sub”, RegDst = rd, RegWr, nPC_sel = “+4”	
ORi	R[rt] <– R[rs] + zero_ext(Imm16);	PC <– PC + 4
	ALUsrc = Im, Extop = “Z”, ALUctr = “or”, RegDst = rt, RegWr, nPC_sel = “+4”	
LOAD	R[rt] <– MEM[R[rs] + sign_ext(Imm16)];	PC <– PC + 4
	ALUsrc = Im, Extop = “Sn”, ALUctr = “add”, MemtoReg, RegDst = rt, RegWr,	nPC_sel = “+4”
STORE	MEM[R[rs] + sign_ext(Imm16)] <– R[rs];	PC <– PC + 4
	ALUsrc = Im, Extop = “Sn”, ALUctr = “add”, MemWr, nPC_sel = “+4”	
BEQ	if (R[rs] == R[rt]) then PC <– PC + sign_ext(Imm16)] || 00 else PC <– PC + 4	
	nPC_sel = EQUAL, ALUctr = “sub”	

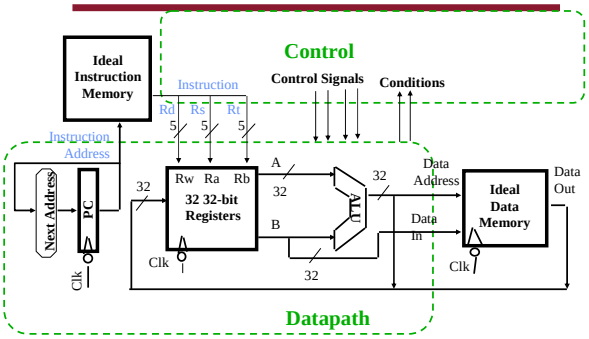
Step 5: Logic for each control signal

- nPC_sel <= if (OP == BEQ) then EQUAL else 0
- ALUsrc <= if (OP == “000000”) then “regB” else “immed”
- ALUctr <= if (OP == “000000”) then funct
  elseif (OP == ORi) then “OR”
  elseif (OP == BEQ) then “sub”
  else “add”
- ExtOp <= if (OP == ORi) then “zero” else “sign”
- MemWr <= (OP == Store)
- MemtoReg <= (OP == Load)
- RegWr: <= if ((OP == Store) || (OP == BEQ)) then 0 else 1
- RegDst: <= if ((OP == Load) || (OP == ORi)) then 0 else 1

Example: Load Instruction



An Abstract View of the Implementation

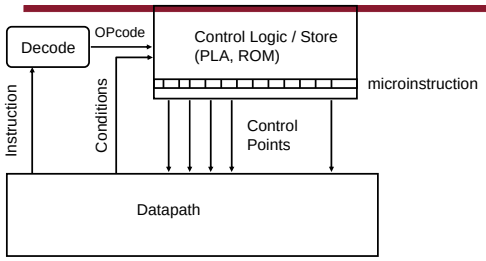


- Logical vs. Physical Structure

Summary

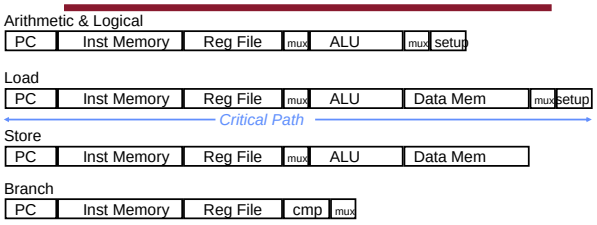
- 5 steps to design a processor
  - 1. Analyze instruction set => datapath requirements
  - 2. Select set of datapath components & establish clock methodology
  - 3. Assemble datapath meeting the requirements
  - 4. Analyze implementation of each instruction to determine setting of control points that effects 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

Systematic Generation of Control



- In a single-cycle processor, each instruction is realized by exactly one control command or "*microinstruction*"
  - in general, the controller is a finite state machine
  - microinstruction can also control sequencing (see later)

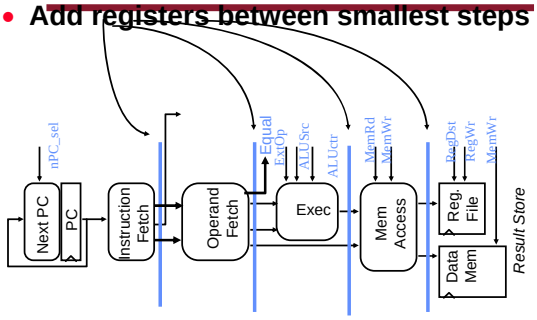
What's wrong with our CPI=1 processor?



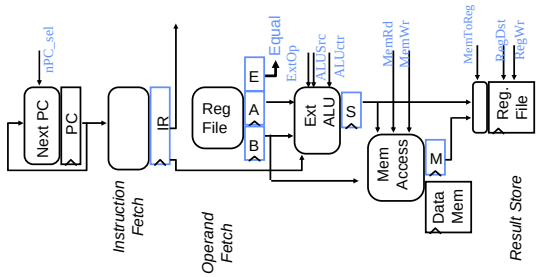
- Long Cycle Time
- All instructions take as much time as the slowest
- Real memory is not as nice as our idealized memory
  - cannot always get the job done in one (short) cycle

Partitioning the CPI=1 Datapath

- Add registers between smallest steps



Example Multicycle Datapath

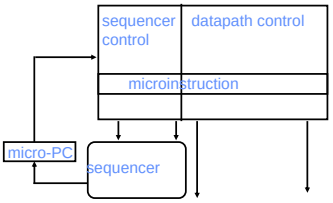


- **Critical Path ?**

Controller Design

- **The state digrams that arise define the controller for an instruction set processor are highly structured**
- **Use this structure to construct a simple "microsequencer"**
- **Control reduces to programming this very simple device**

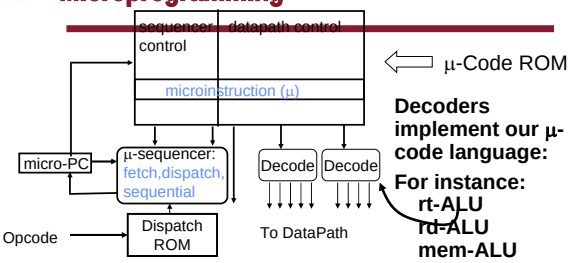
  ⇒ **microprogramming**



Microprogramming

- **Microprogramming is a convenient method for implementing *structured* control state diagrams:**
  - **Random logic replaced by microPC sequencer and ROM**
  - **Each line of ROM called a μinstruction: contains sequencer control + values for control points**
  - **limited state transitions: branch to zero, next sequential, branch to μinstruction address from displatch ROM**
- **Horizontal μCode: one control bit in μinstruction for every control line in datapath**
- **Vertical μCode: groups of control-lines coded together in μinstruction (e.g. possible ALU dest)**
- **Control design reduces to Microprogramming**
  - **Part of the design process is to develop a "language" that describes control and is easy for humans to understand**

Microprogramming



- **Microprogramming is a fundamental concept**
  - **implement an instruction set by building a very simple processor and interpreting the instructions**
  - **essential for very complex instructions and when few register transfers are possible**
  - **overkill when ISA matches datapath 1-1**

Microprogramming one inspiration for RISC

- If simple instruction could execute at very high clock rate…
  - you could even write compilers to produce microinstructions…
- If most programs use simple instructions and addressing modes…
- If microcode is kept in RAM instead of ROM so as to fix bugs …
- Then why not skip instruction interpretation by a microprogram and simply compile directly into lowest language of machine? (microprogramming is overkill when ISA matches datapath 1-1)

How to improve performance?

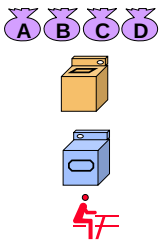
- Recall performance is function of
  - CPI: cycles per instruction
  - Clock cycle
  - Instruction count
- Reducing any of the 3 factors will lead to improved performance

How to improve performance?

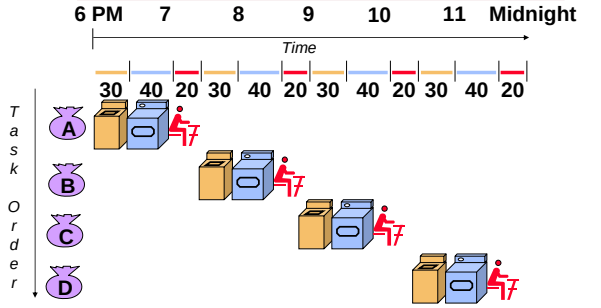
- First step is to apply concept of pipelining to the instruction execution process
  - Overlap computations
- What does this do?
  - Decrease clock cycle
  - Decrease effective CPI compared to original clock cycle

Pipelining: Its 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

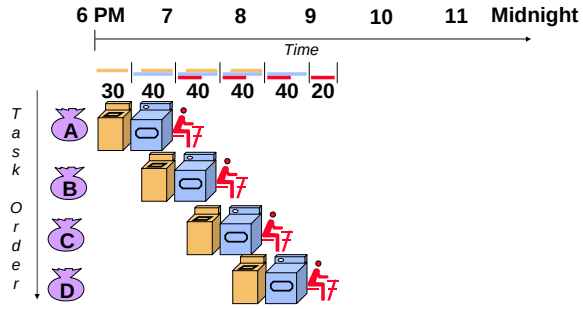


Sequential Laundry



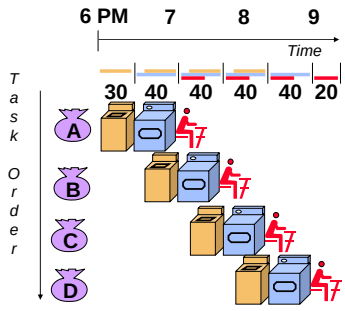
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

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 limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup

Instruction Pipeline

- Instruction execution process lends itself naturally to pipelining
  - overlap the subtasks of instruction fetch, decode and execute

How to improve performance?

- Recall performance is function of
  - CPI: cycles per instruction
  - Clock cycle
  - Instruction count
- Reducing any of the 3 factors will lead to improved performance

How to improve performance?

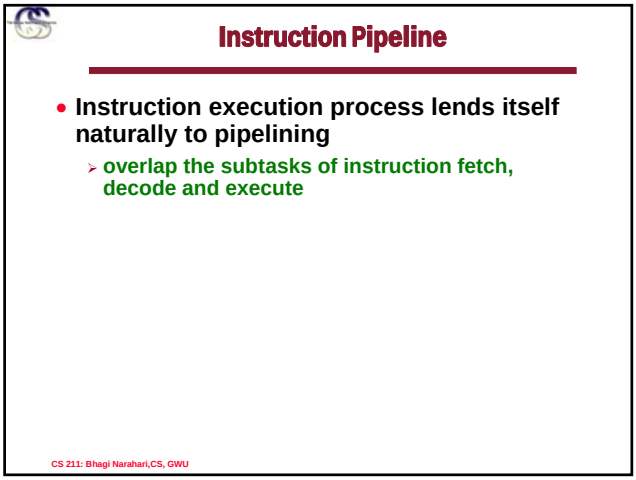
- First step is to apply concept of pipelining to the instruction execution process
  - Overlap computations
- What does this do?
  - Decrease clock cycle
  - Decrease effective CPU time compared to original clock cycle

Pipeline Approach to Improve System Performance

- Analogous to fluid flow in pipelines and assembly line in factories
- Divide process into "stages" and send tasks into a pipeline
  - Overlap computations of different tasks by operating on them concurrently in different stages

Instruction Level Parallel Processors (ILP)

- early ILP - one of two orthogonal concepts:
  - pipelining - vertical approach
  - multiple (non-pipelined) units - horizontal approach
- progression to multiple pipelined units
- instruction issue became bottleneck, led to
  - superscalar ILP processors
  - Very Large Instruction Word (VLIW)
- Note: key performance metric in all ILP processor classes is IPC (instructions per cycle)
  - this is the degree of parallelism achieved


Instruction Pipeline
Instruction execution process lends itself naturally to pipelining
overlap the subtasks of instruction fetch, decode and execute
CS 211: Bhagi Narahari,CS, GWU