RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

ECE-332:437
DIGITAL SYSTEMS DESIGN (DSD)

Fall 2016 – Lecture 8
Architecture (RISC - MIPS)

Nagi Naganathan
October 27, 2016

Topics to cover today – October 27, 2016

- How was Mid Term 1?
- How are the labs progressing?

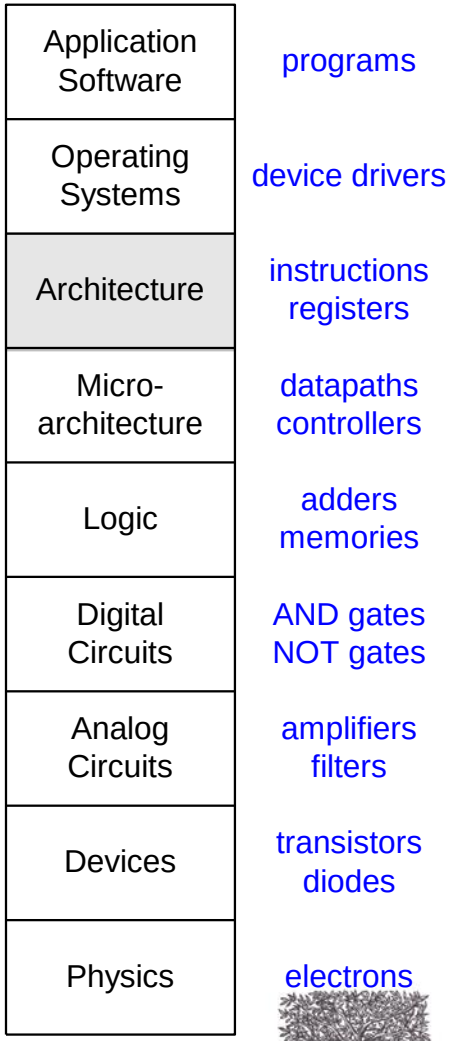
- Lecture 8 – Chapter 6 – Architecture

- Remaining Chapters
  - Lecture 9 – Chapter 7 – Microarchitecture
  - Lecture 10, 11 – ARM Architecture and Microarchitecture
  - Lecture 12 – Chapter 8 – Memory, Cache
  - Lecture 13 – ARM Cache

Part 1

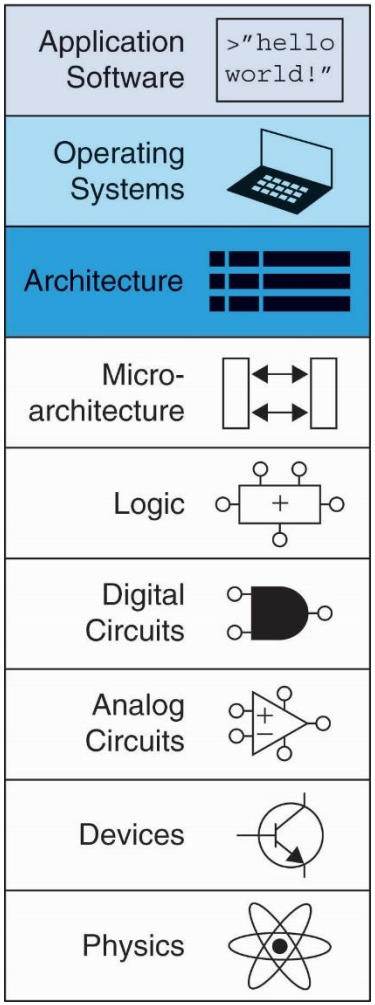
Introduction

- Jumping up a few levels of abstraction
- **Architecture:** programmer's view of computer
  - Defined by instructions & operand locations
- **Microarchitecture:** how to implement an architecture in hardware (covered in Chapter 7)



Chapter 6 :: Topics

- Introduction
- Assembly Language
- Machine Language
- Programming
- Addressing Modes
- Lights, Camera, Action: Compiling, Assembling, & Loading
- Odds and Ends



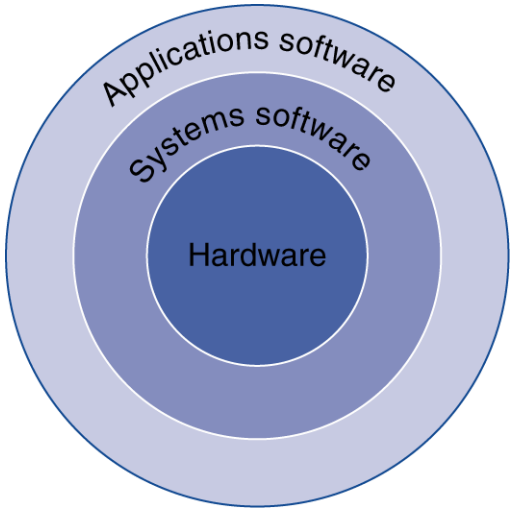
The Computer Revolution

- Progress in computer technology
  - Underpinned by Moore's Law
- Makes novel applications feasible
  - Computers in automobiles
  - Cell phones
  - Human genome project
  - World Wide Web
  - Search Engines
- Computers are pervasive

Classes of Computers

- Desktop computers
  - General purpose, variety of software
  - Subject to cost/performance tradeoff
- Server computers
  - Network based
  - High capacity, performance, reliability
  - Range from small servers to building sized
- Embedded computers
  - Hidden as components of systems
  - Stringent power/performance/cost constraints

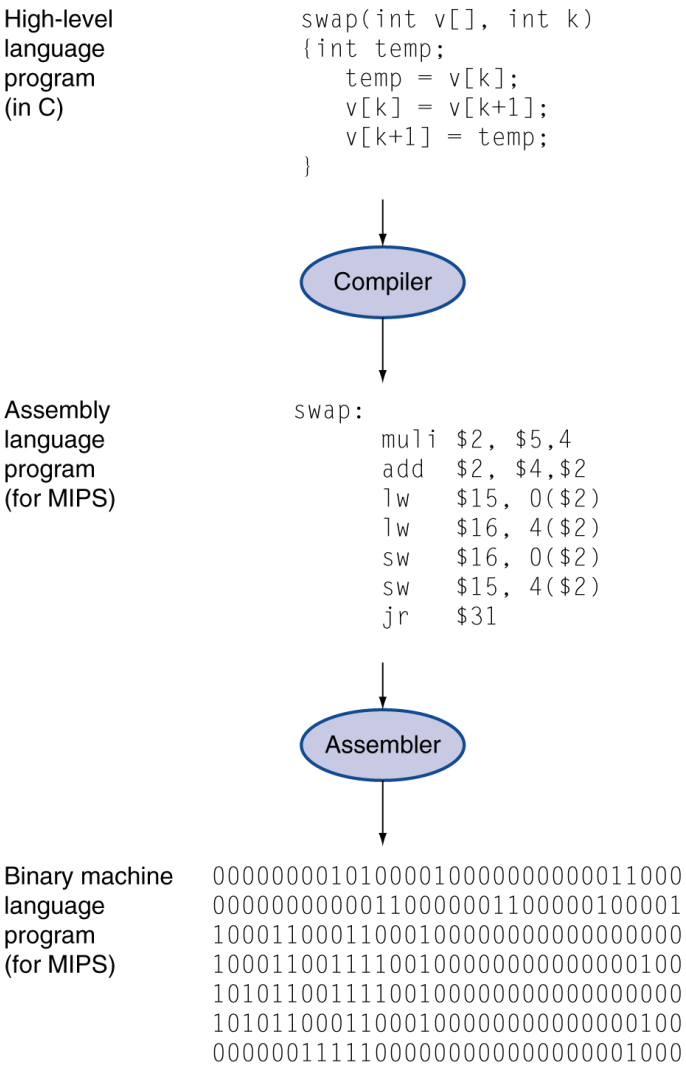
Below Your Program



- Application software
  - Written in high-level language
- System software
  - Compiler: translates HLL code to machine code
  - Operating System: service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks & sharing resources
- Hardware
  - Processor, memory, I/O controllers

Levels of Program Code

- High-level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- Assembly language
  - Textual representation of instructions
- Hardware representation
  - Binary digits (bits)
  - Encoded instructions and data



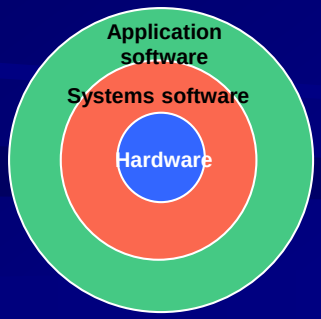
Software

Compiler

Assembler

Application software, a program in C:

```
swap (int  v[ ], int  k)
{int  temp;
        temp = v[k];
        v[k] = v[k+1];
        v[k+1] = temp;
}
```



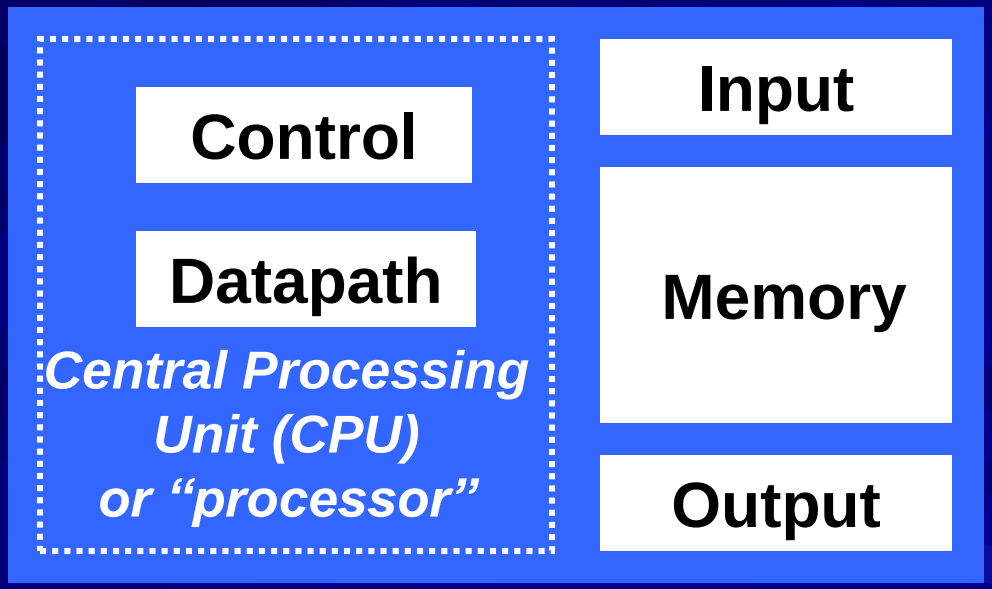
MIPS compiler output, assembly language program:

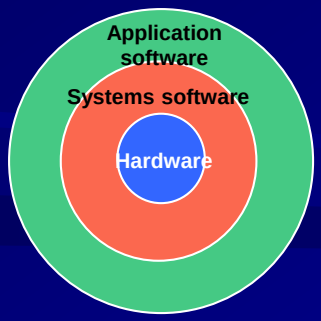
```
swap;
        muli    $2,     $5, 4
        add     $2,     $4, $2
        lw      $15,    0 ($2)
        lw      $16,    4 ($2)
        sw      $16,    0 ($2)
        sw      $15,    4 ($2)
        jr      $31
```

MIPS binary machine code:

```
00000000101000010000000000011000
00000000000011000000110000010000 1
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

The Hardware of a Computer



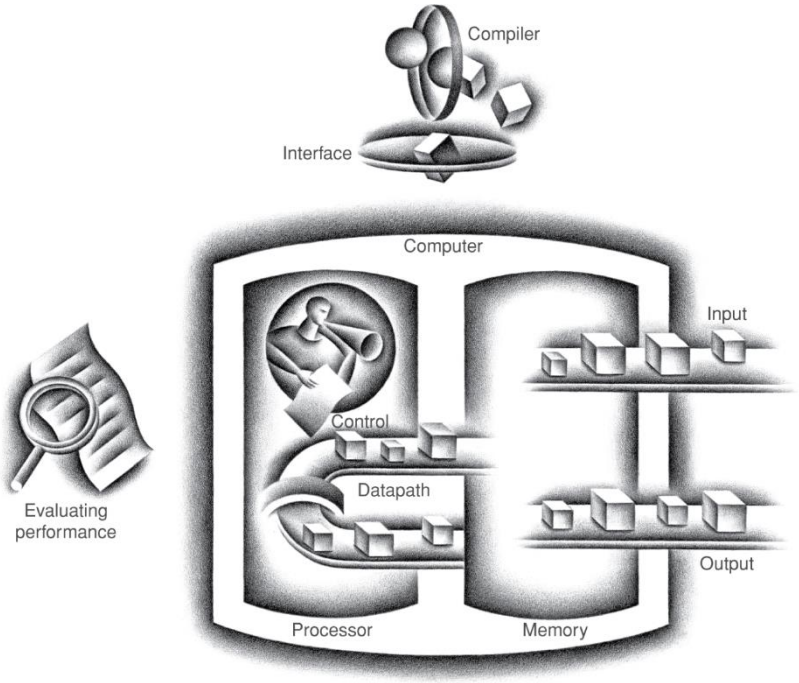


FIVE EASY PIECES

Anatomy of a Computer

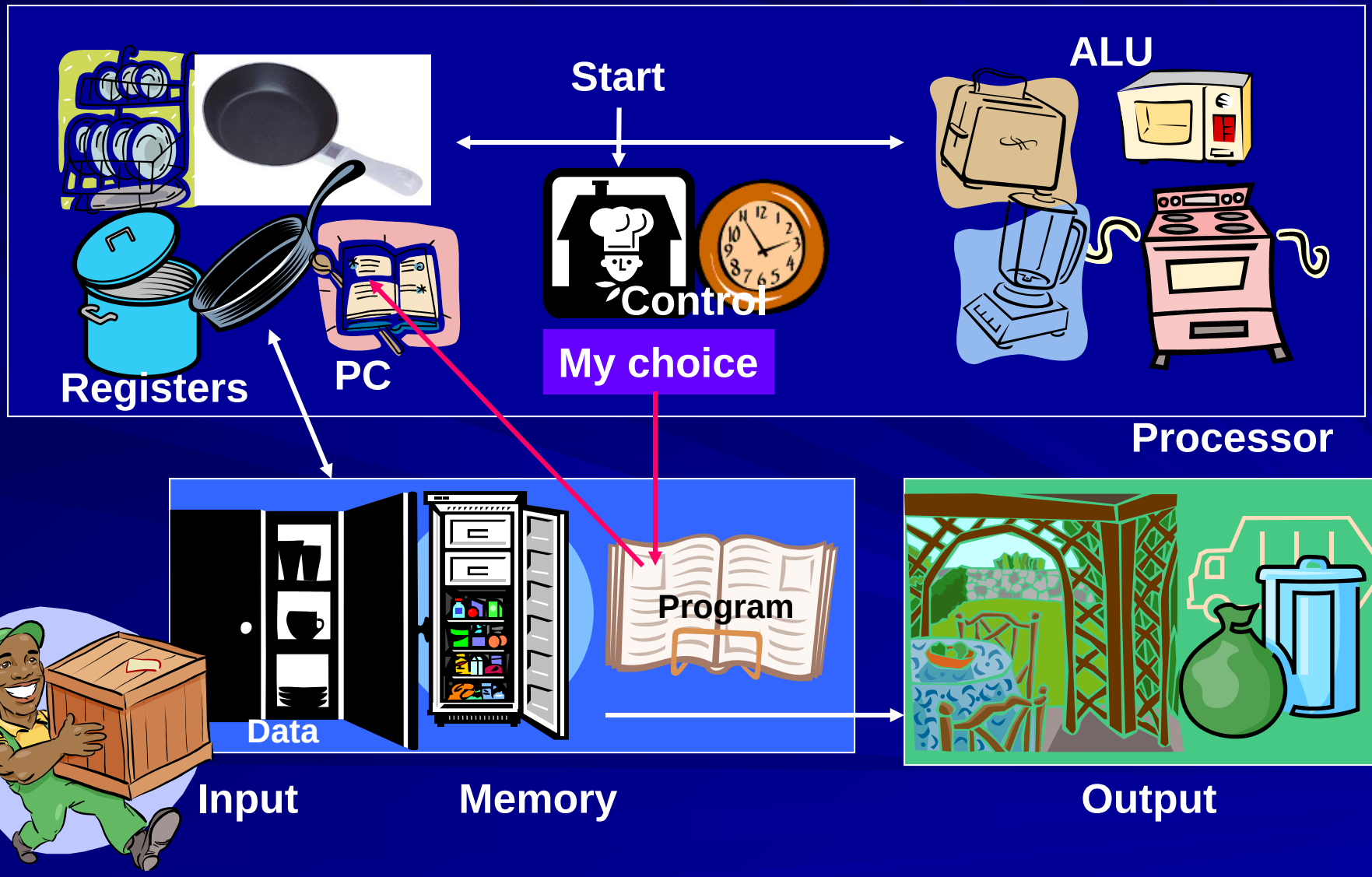
Components of a Computer

The BIG Picture



- Same components for all kinds of computer
  - Desktop, server, embedded
- Input/output includes
  - User-interface devices
    - Display, keyboard, mouse
  - Storage devices
    - Hard disk, CD/DVD, flash
  - Network adapters
    - For communicating with other computers

Von Neumann Kitchen



Datapath and Control

- Datapath: Memory, registers, adders, ALU, and communication buses. Each step (fetch, decode, execute) requires communication (data transfer) paths between memory, registers and ALU.
- Control: Datapath for each step is set up by control signals that set up dataflow directions on communication buses and select ALU and memory functions. Control signals are generated by a control unit consisting of one or more finite-state machines.

Inside the Processor (CPU)

- Datapath: performs operations on data
- Control: sequences datapath, memory, ...
- Cache memory
  - Small fast SRAM memory for immediate access to data

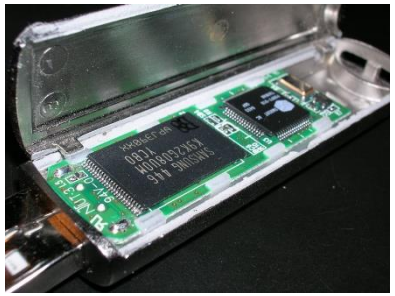
Abstractions

The BIG Picture

- Abstraction helps us deal with complexity
  - Hide lower-level detail
- Instruction set architecture (ISA)
  - The hardware/software interface
- Application binary interface
  - The ISA plus system software interface
- Implementation
  - The details underlying and interface

A Safe Place for Data

- Volatile main memory
  - Loses instructions and data when power off
- Non-volatile secondary memory
  - Magnetic disk
  - Flash memory
  - Optical disk (CDROM, DVD)



Computer *Architecture*

- Architecture: System attributes that have a direct impact on the logical execution of a program
- Architecture is visible to a programmer:
  - Instruction set
  - Data representation
  - I/O mechanisms
  - Memory addressing

Computer *Organization*

- Organization: Physical details that are transparent to a programmer, such as
  - Hardware implementation of an instruction
  - Control signals
  - Memory technology used
- Example: System/370 architecture has been used in many IBM computers, which widely differ in their organization.

First-Generation Computers

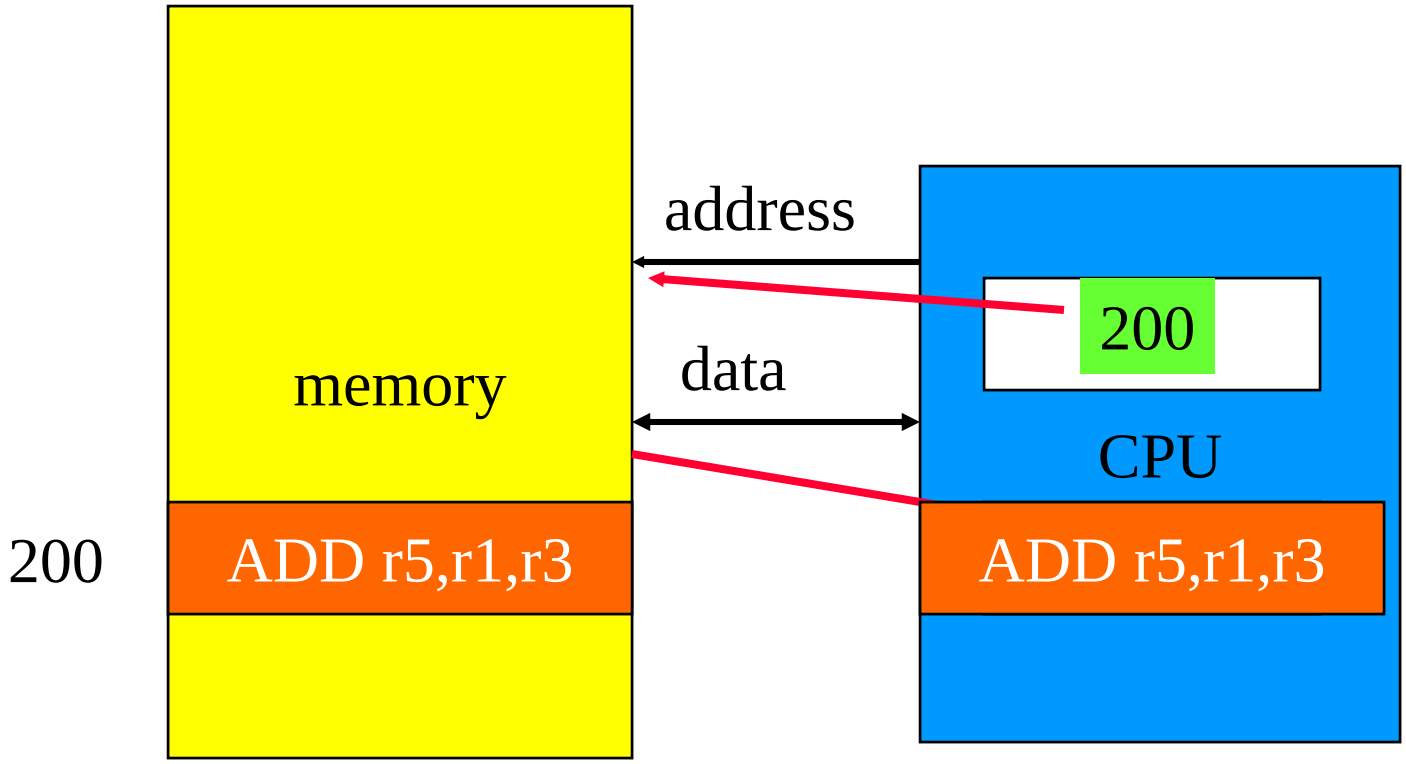
- Late 1940s and 1950s
- Stored-program computers
- Programmed in assembly language
- Used magnetic devices and earlier forms of memories
- Examples: IAS, ENIAC, EDVAC, UNIVAC, Mark I, IBM 701

Theory of Computing

- Alan Turing (1912-1954) gave a model of computing in 1936 – *Turing Machine*.
- Original paper: A. M. Turing, "On Computable Numbers with an Application to the *Entscheidungsproblem**," *Proc. Royal Math. Soc.*, ser. 2, vol. 42, pp. 230-265, 1936.
- Recent book: David Leavitt, *The Man Who Knew Too Much: Alan Turing and the Invention of the Computer (Great Discoveries)*, W. W. Norton & Co., 2005.

* *The question of decidability, posed by mathematician Hilbert.*

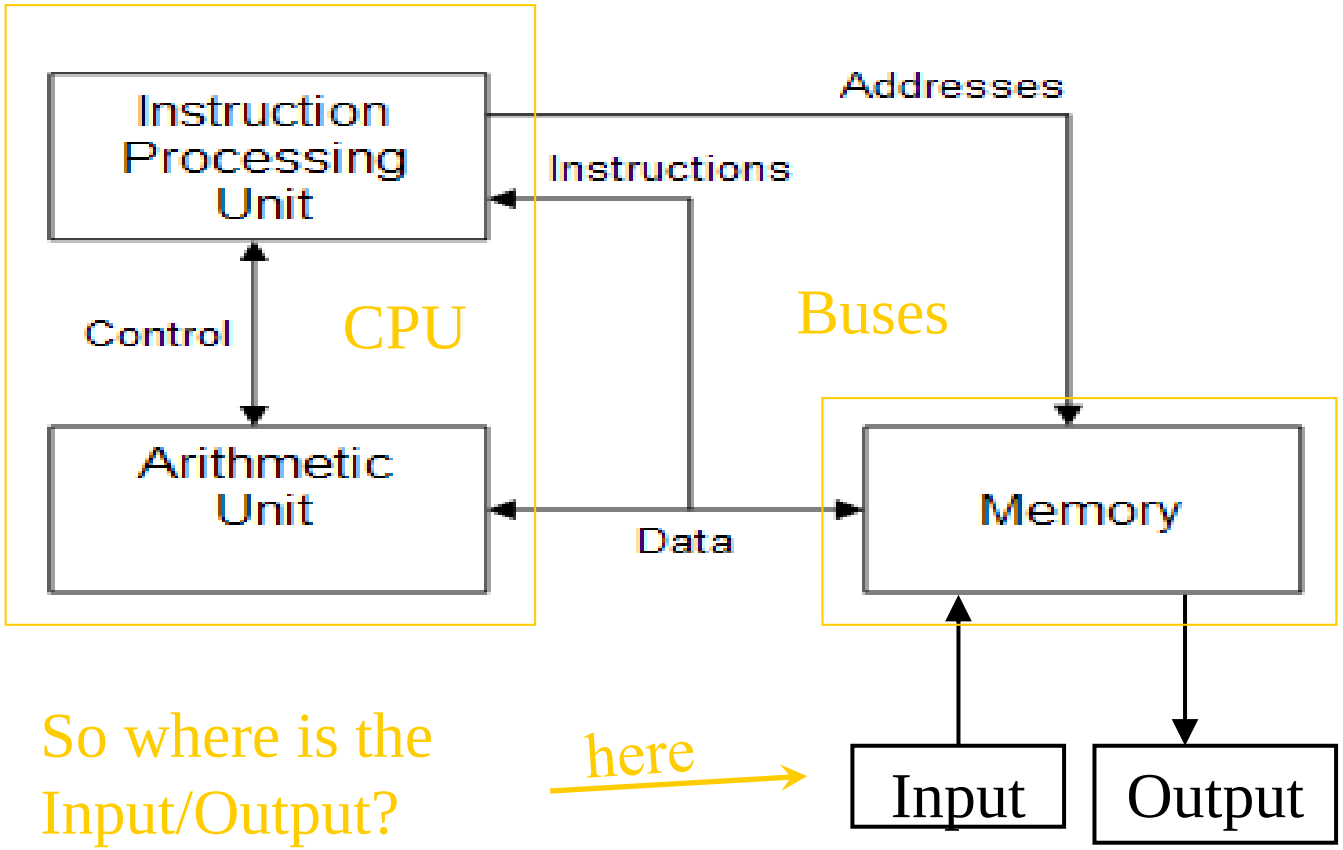
CPU + memory



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.

The von Neumann model



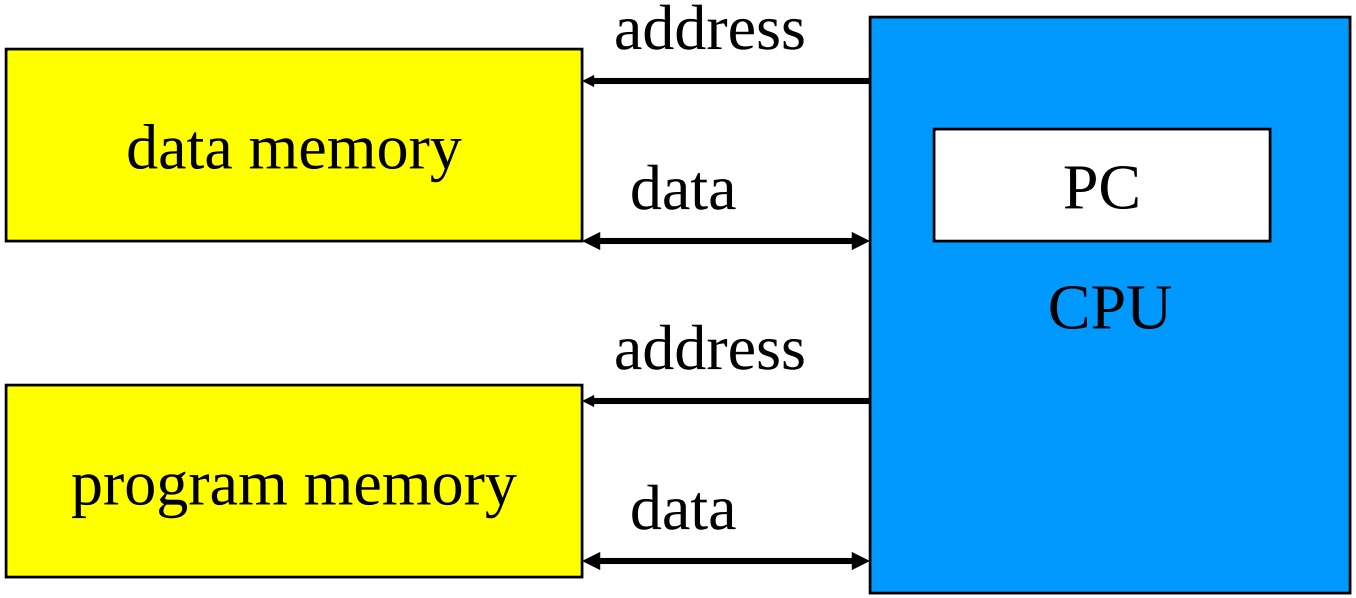
Second Generation Computers

- 1955 to 1964
- Transistor replaced vacuum tubes
- Magnetic core memories
- Floating-point arithmetic
- High-level languages used: ALGOL, COBOL and FORTRAN
- System software: compilers, subroutine libraries, batch processing
- Example: IBM 7094

Third Generation Computers

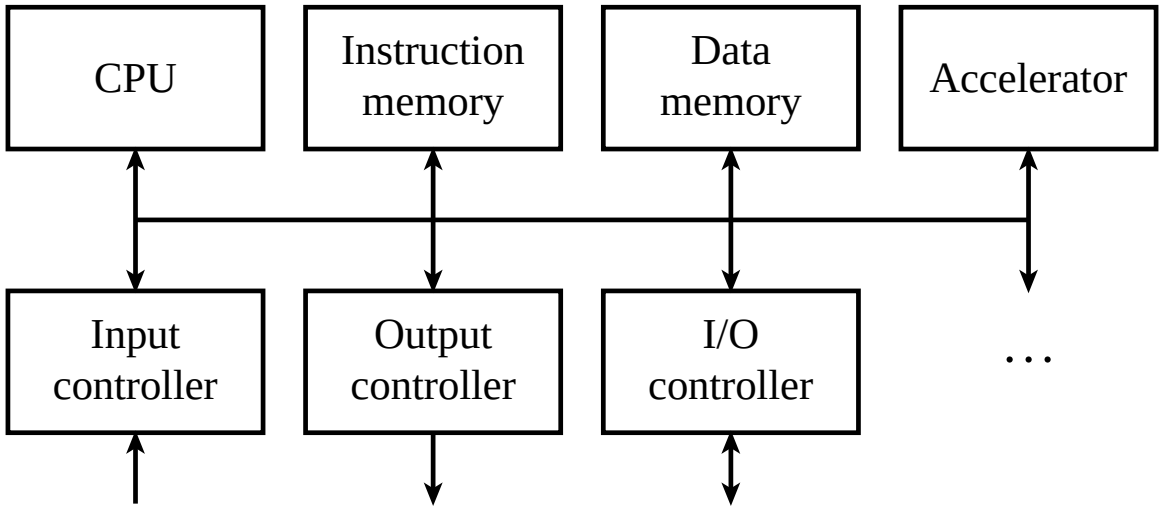
- Beyond 1965
- Integrated circuit (IC) technology
- Semiconductor memories
- Memory hierarchy, virtual memories and caches
- Time-sharing
- Parallel processing and pipelining
- Microprogramming
- Examples: IBM 360 and 370, CYBER, ILLIAC IV, DEC PDP and VAX, Amdahl 470

Harvard architecture



Memory Organization

- Von Neumann architecture
  - Single memory for instructions and data
- Harvard architecture
  - Separate instruction and data memories
  - Most common in embedded systems



The Harvard Architecture (1)

- **Harvard architecture** is a computer architecture with physically separate storage and signal pathways for instructions and data.
- The term originated from the Harvard Mark I relay-based computer, which stored instructions on punched tape (24 bits wide) and data in electro-mechanical counters (23 digits wide). These early machines had limited data storage, entirely contained within the data processing unit, and provided no access to the instruction storage as data, making loading and modifying programs an entirely offline process.

The Harvard Architecture (3)

- In a Harvard architecture, there is no need to make the two memories share characteristics. In particular, the word width, timing, implementation technology, and memory address structure can differ.
- In some systems, instructions can be stored in read-only memory while data memory generally requires read-write memory.
- Instruction memory is often wider than data memory.

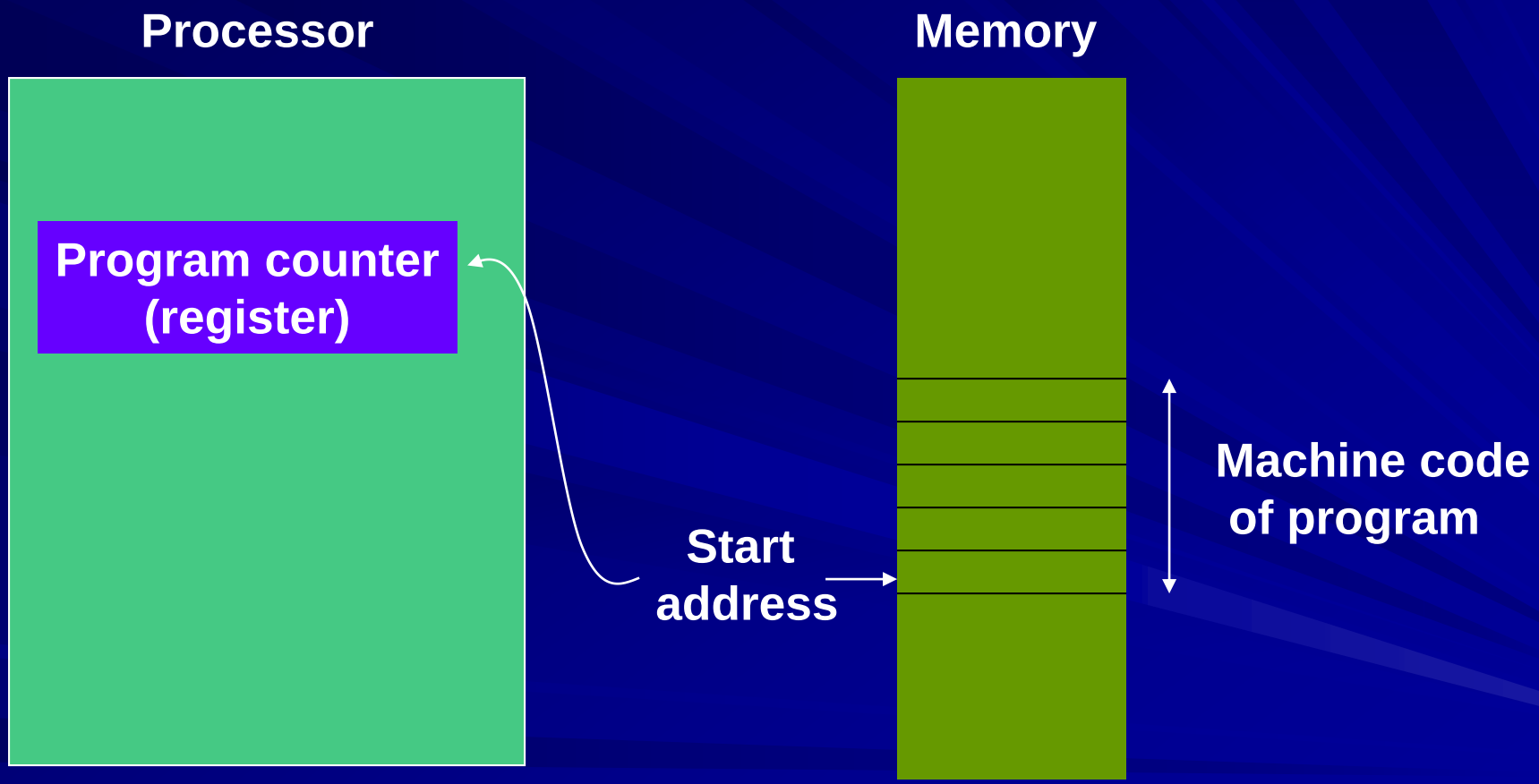
von Neumann vs. Harvard

- Harvard allows two simultaneous memory fetches.
- Most DSPs use Harvard architecture for streaming data:
  - greater memory bandwidth;
  - more predictable bandwidth.

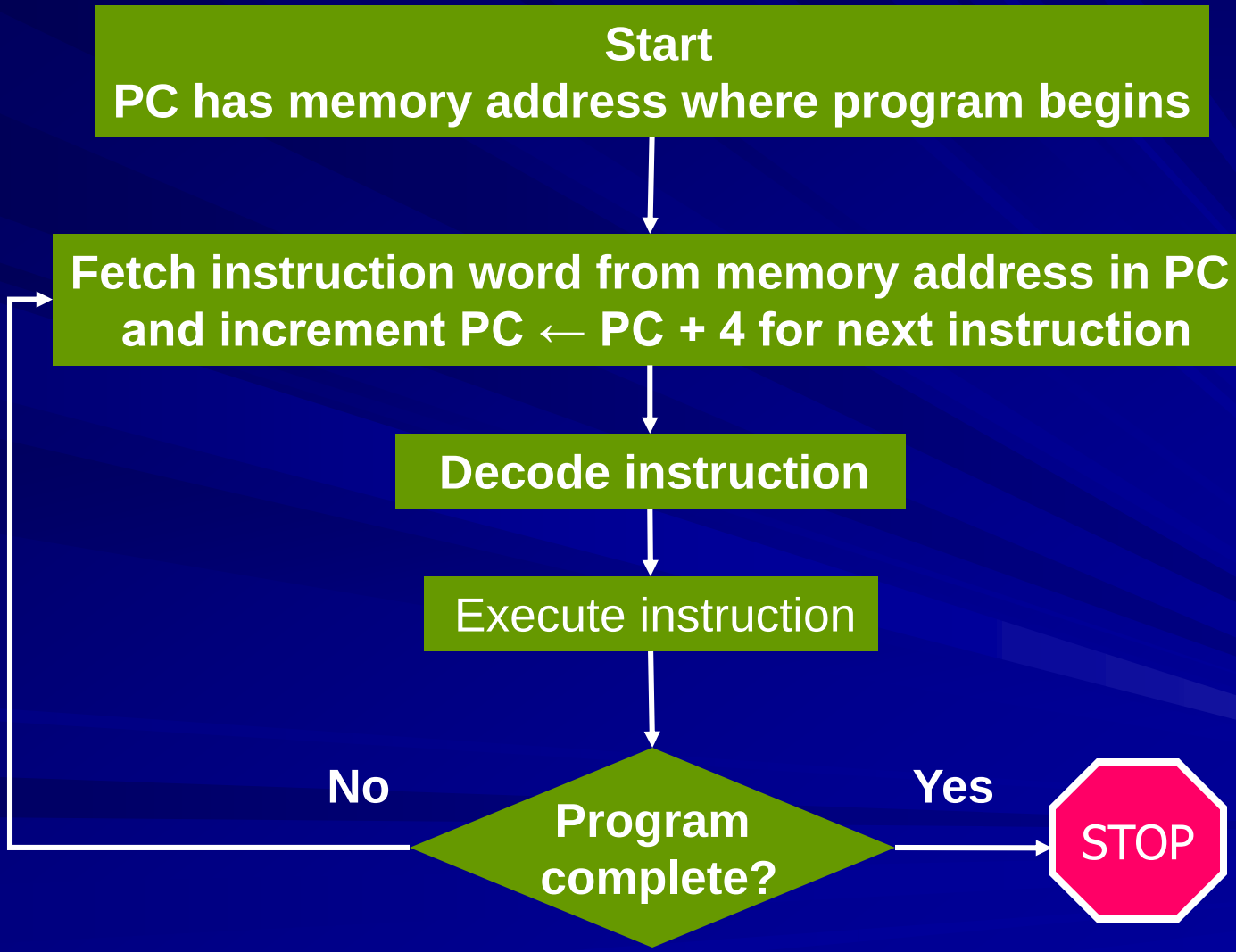
Where Does It All Begin?

- In a register called *program counter (PC)*.
- PC contains the memory address of the next instruction to be executed.
- In the beginning, PC contains the address of the memory location where the program begins.

Where is the Program?

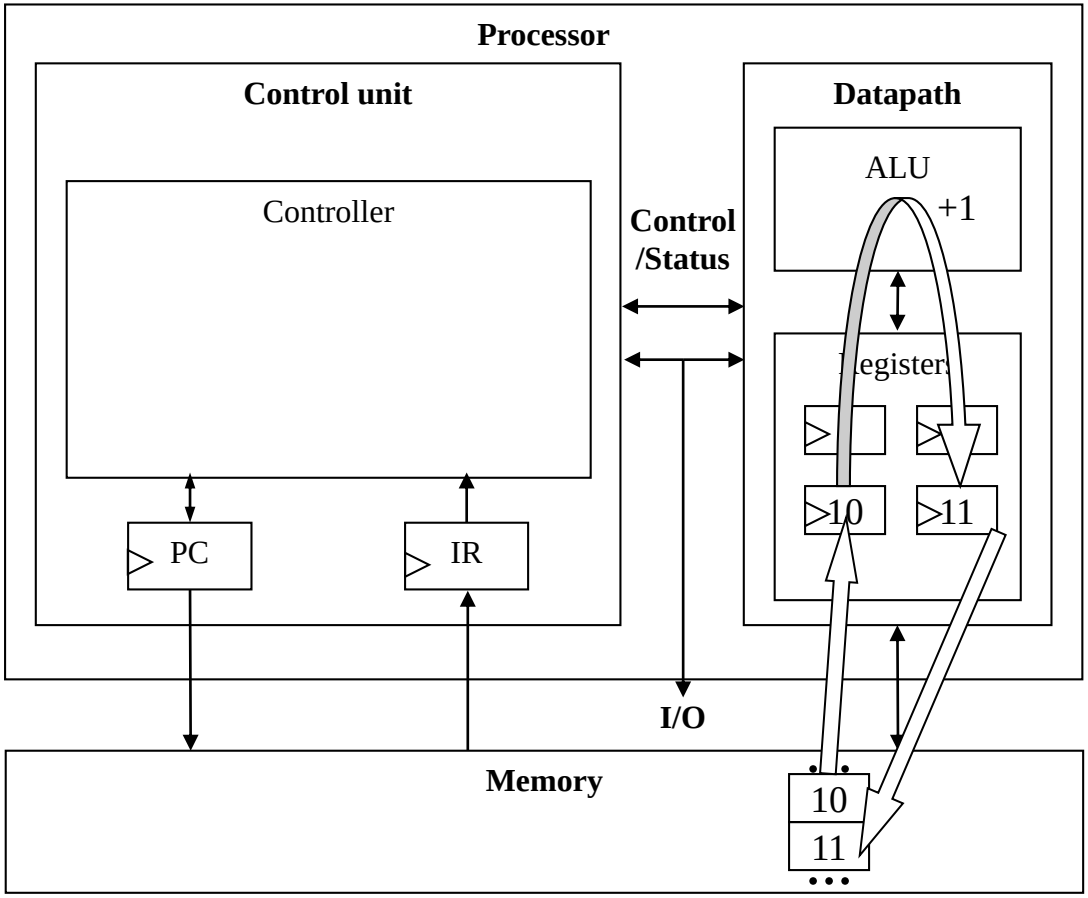


How Does It Run?

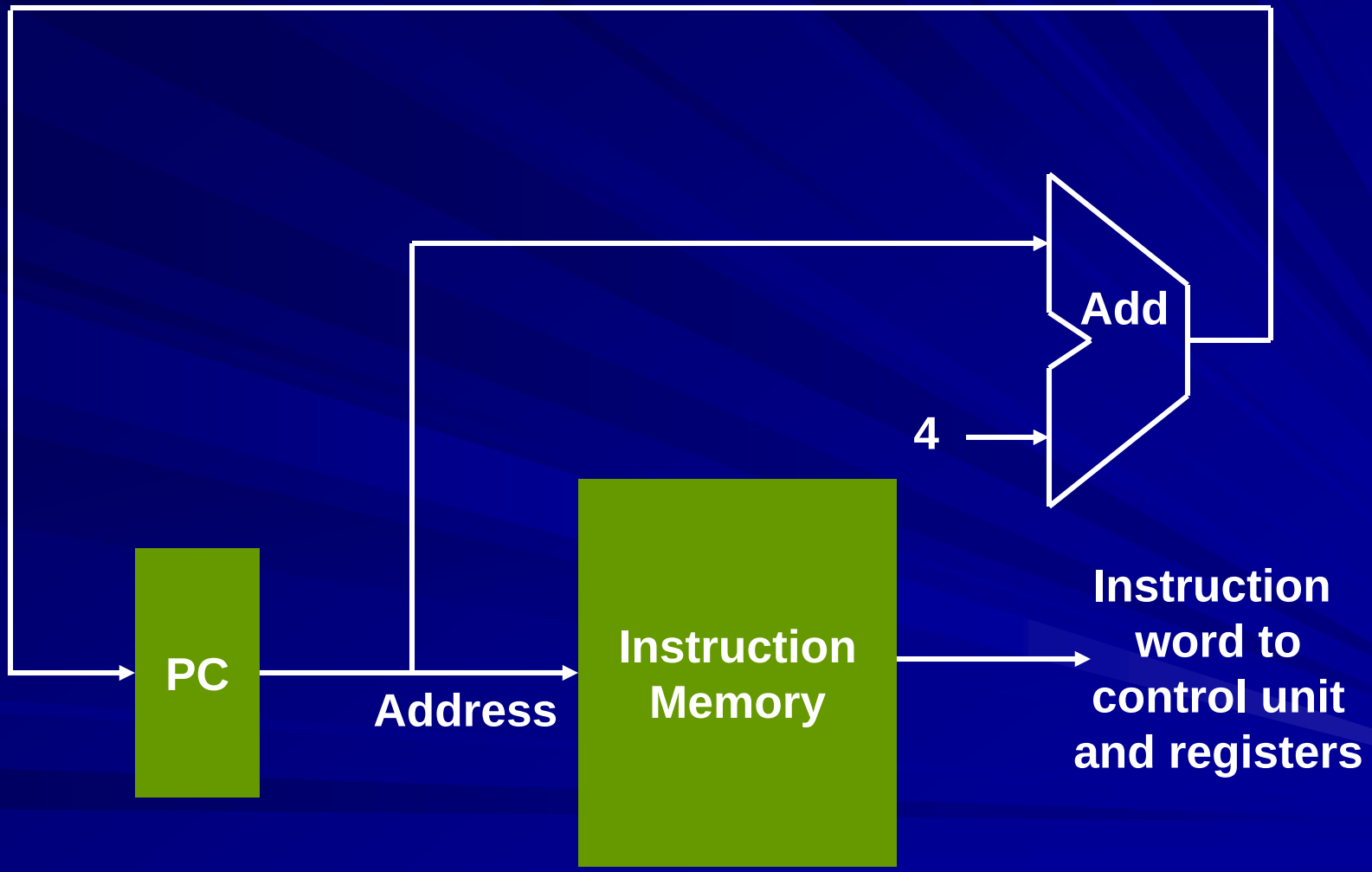


Datapath Operations

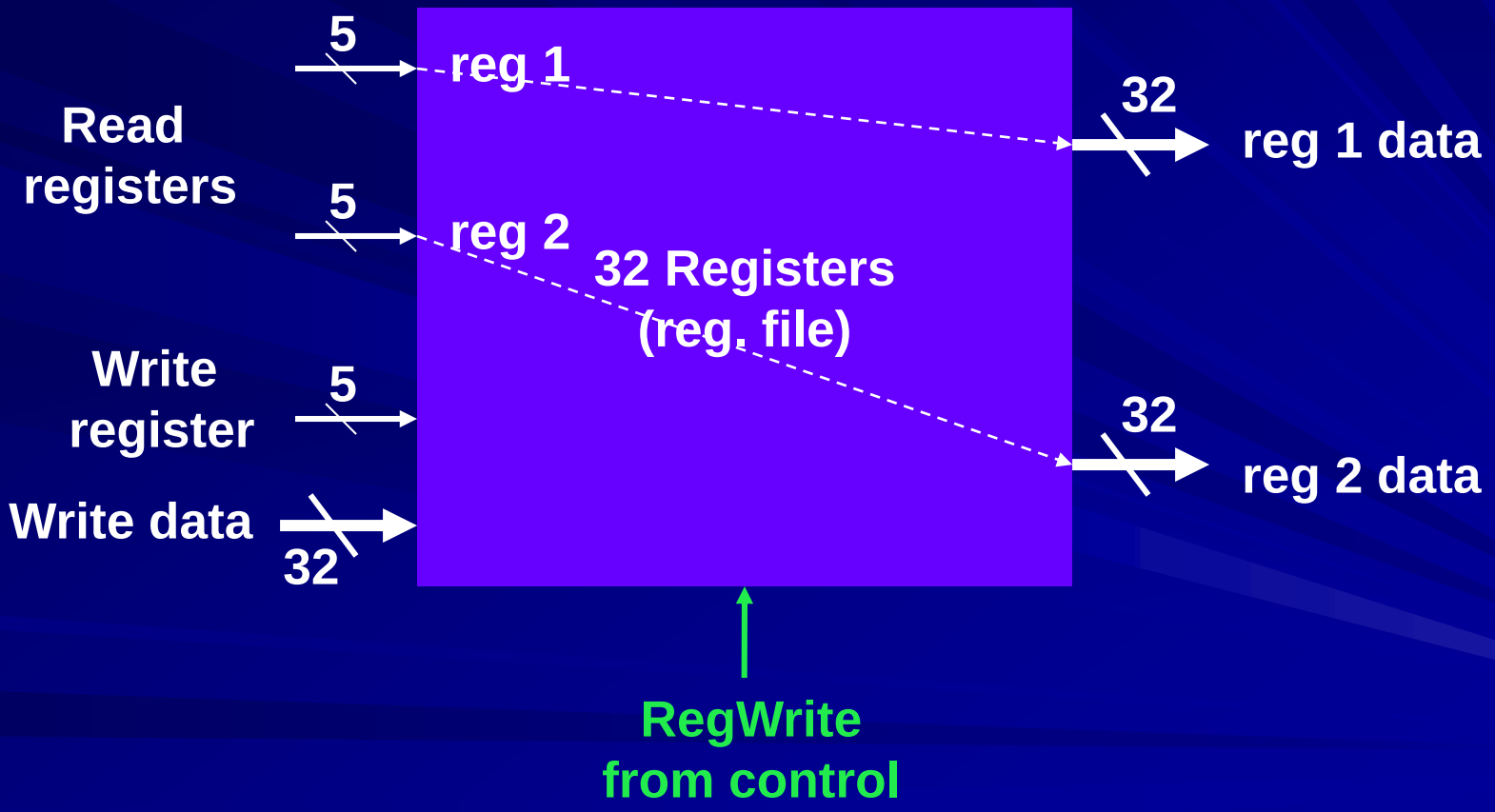
- Load
  - Read memory location into register
- ALU operation
  - Input certain registers through ALU, store back in register
- Store
  - Write register to memory location



Datapath for Instruction Fetch



Register File: A Datapath Component



Part 2

Assembly Language

- **Instructions:** commands in a computer's language
  - **Assembly language:** human-readable format of instructions
  - **Machine language:** computer-readable format (1's and 0's)
- **MIPS** architecture:
  - Developed by John Hennessy and his colleagues at Stanford and in the 1980's.
  - Used in many commercial systems, including Silicon Graphics, Nintendo, and Cisco

Once you've learned one architecture, it's easy to learn others

John Hennessy

- President of Stanford University
- Professor of Electrical Engineering and Computer Science at Stanford since 1977
- Co-invented the Reduced Instruction Set Computer (RISC) with David Patterson
- Developed the MIPS architecture at Stanford in 1984 and cofounded MIPS Computer Systems
- As of 2004, over 300 million MIPS microprocessors have been sold

The MIPS ISA

- MIPS (*Microprocessor without Interlocked Pipeline Stages*) is a reduced instruction set architecture.

Architecture Design Principles

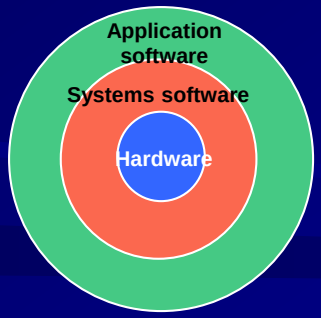
Underlying design principles, as articulated by Hennessy and Patterson:

1. **Simplicity favors regularity**
2. **Make the common case fast**
3. **Smaller is faster**
4. **Good design demands good compromises**

- Instruction Set Architecture
  - the machine behavior as observable and controllable by the programmer
- Instruction Set
  - the set of commands understood by the computer
- Machine Code
  - a collection of instructions encoded in binary format
  - directly consumable by the hardware
- Assembly Code
  - a collection of instructions expressed in "textual" format e.g. Add r1, r2, r3
  - converted to machine code by an assembler
  - one-to-one correspondence with machine code (mostly true: compound instructions, address labels ....)

Instruction Set Architecture (ISA)

- A set of assembly language instructions (ISA) provides a link between software and hardware.
- Given an instruction set, software programmers and hardware engineers work more or less independently.
- ISA is designed to extract the most performance out of the available hardware technology.



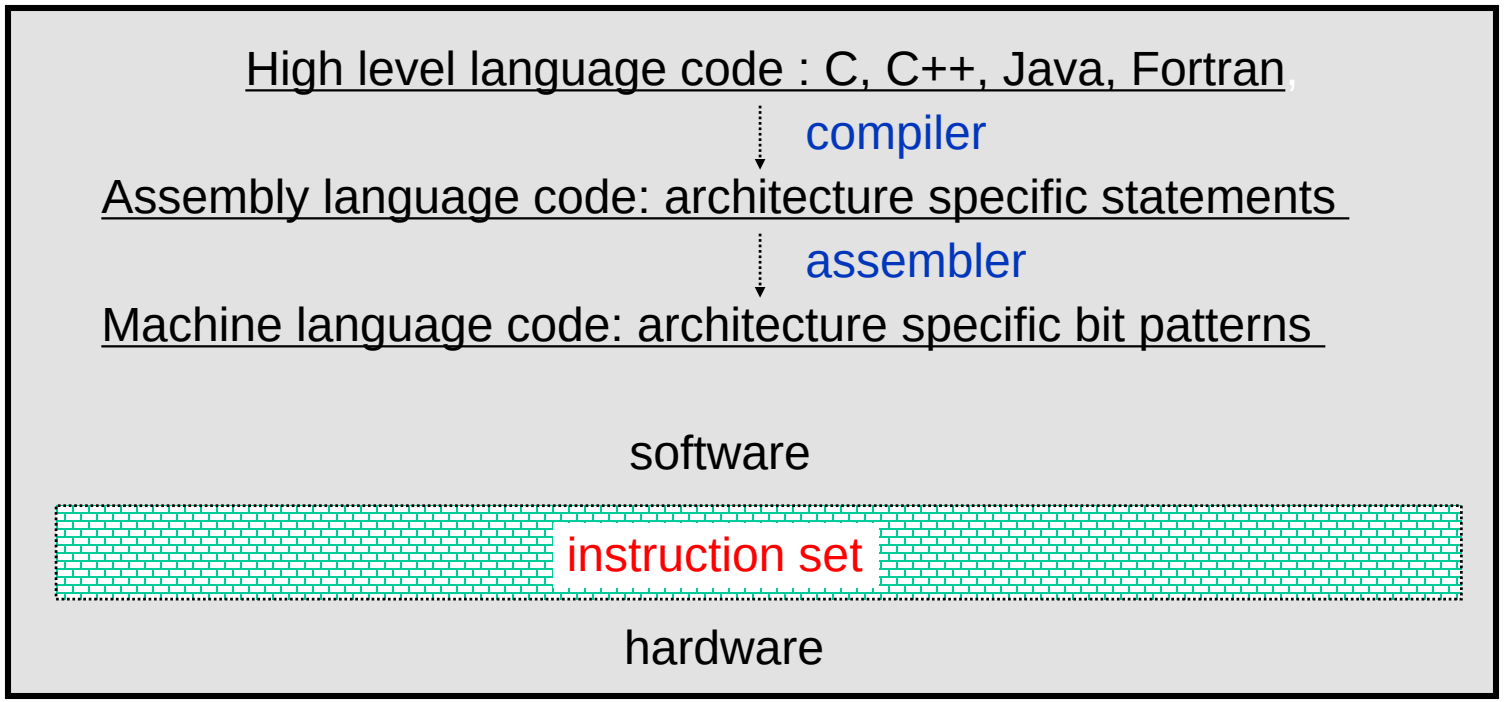
Software

Instruction set

Hardware

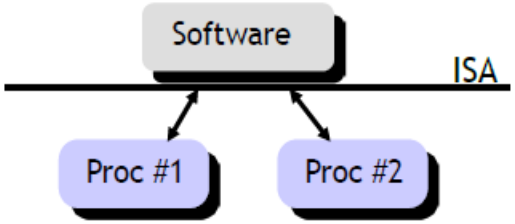
Instruction Set Architecture (ISA)

- Serves as an interface between software and hardware.
- Provides a mechanism by which the software tells the hardware what should be done.



What's an ISA?

- The ISA is the interface between hardware and software.
- The ISA serves as an abstraction layer between the HW and SW
  - Software doesn't need to know how the processor is implemented
  - Any processor that implements the ISA appears equivalent



- An ISA enables processor innovation without changing software
  - This is how Intel has made billions of dollars.
- Before ISAs, software was re-written for each new machine.

ISA

- Defines registers
- Defines data transfer modes between registers, memory and I/O
- Types of ISA: RISC, CISC, VLIW, Superscalar
- Examples:
  - IBM370/X86/Pentium/K6 (CISC)
  - PowerPC (Superscalar)
  - Alpha (Superscalar)
  - MIPS (RISC and Superscalar)
  - Sparc (RISC), UltraSparc (Superscalar)

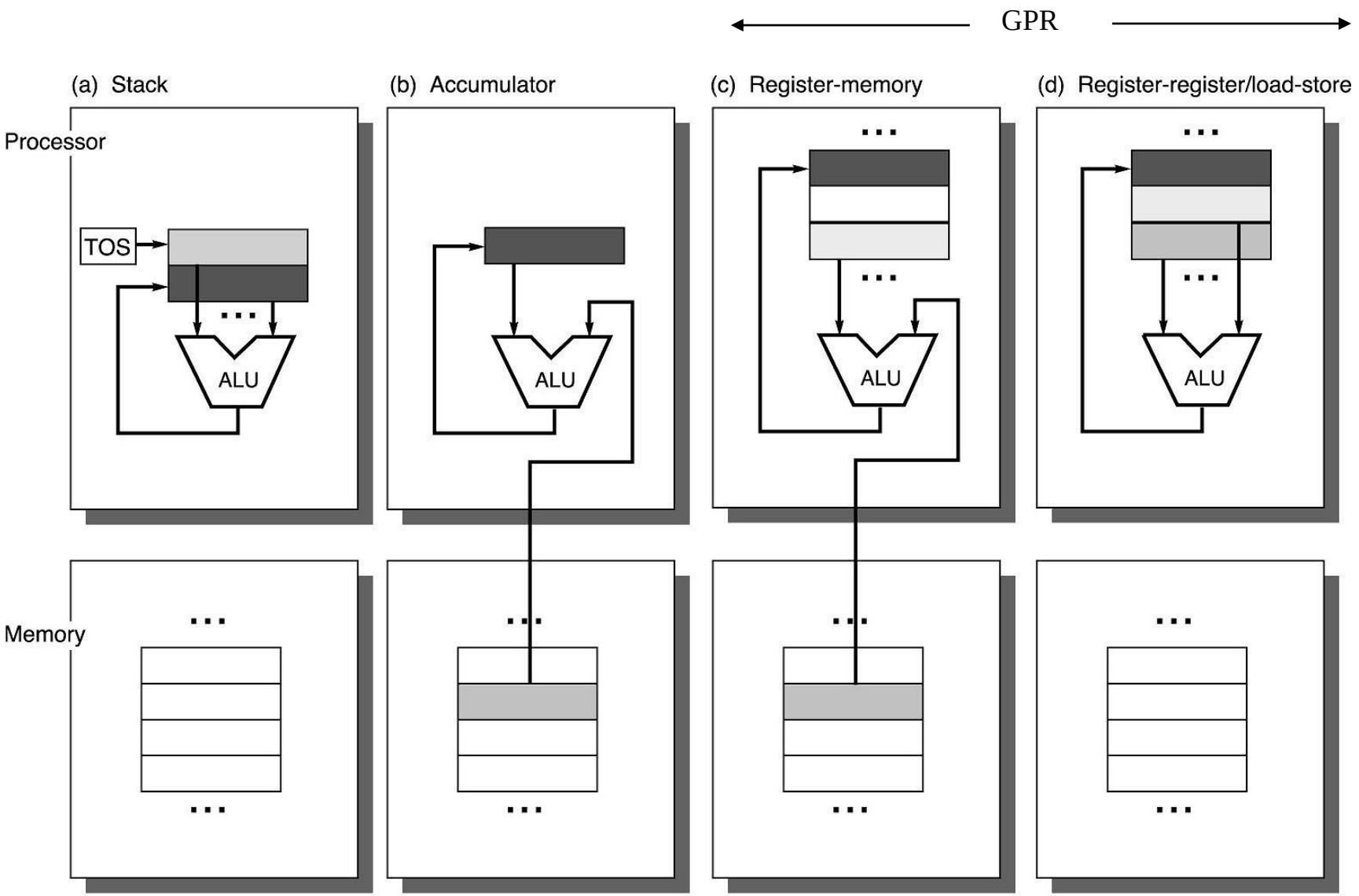
Why ISA?

- An ISA provides binary compatibility across machines that share the ISA
- Any machine that implements the ISA X can execute a program encoded using ISA X.
- You typically see families of machines, all with the same ISA, but with different power, performance and cost characteristics.
  - e.g., the MIPS family: MIPS 2000, 3000, 4400, 10000

Terminologies

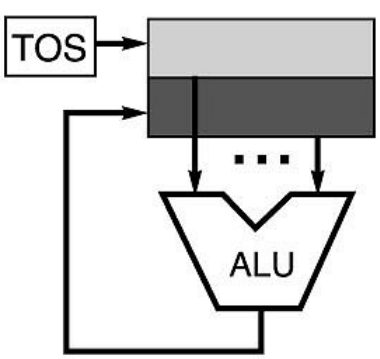
- Instruction Set Architecture
  - the machine behavior as observable and controllable by the programmer
- Instruction Set
  - the set of commands understood by the computer
- Machine Code
  - a collection of instructions encoded in binary format
  - directly consumable by the hardware
- Assembly Code
  - a collection of instructions expressed in "textual" format e.g. Add r1, r2, r3
  - converted to machine code by an assembler
  - one-to-one correspondence with machine code (mostly true: compound instructions, address labels

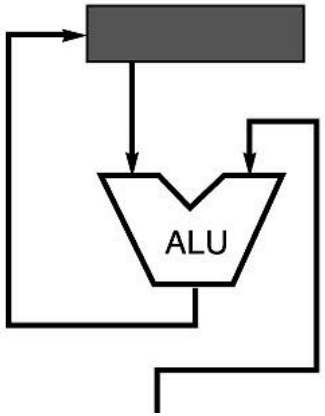
Operand Locations in Four ISA Classes



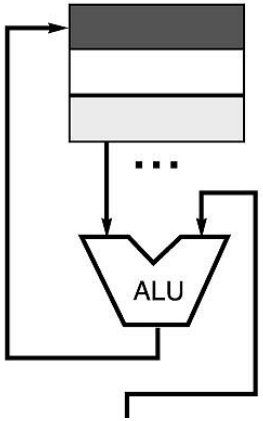
Code Sequence C = A + B for Four Instruction Sets

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R1, B Store C, R1	Load R1,A Load R2, B Add R3, R1, R2 Store C, R3

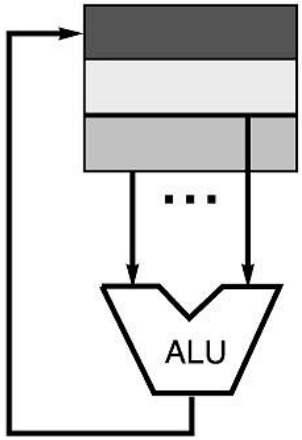




acc = acc + mem[C]



R1 = R1 + mem[C]



R3 = R1 + R2

Types of ISA

- Complex instruction set computer (CISC)
  - Many instructions (several hundreds)
  - An instruction takes many cycles to execute
  - Example: Intel Pentium
- Reduced instruction set computer (RISC)
  - Small set of instructions (typically 32)
  - Simple instructions, each executes in one clock cycle – ***REALLY? Well, almost.***
  - Effective use of pipelining
  - Example: ARM

ISA

- The architecture is similar to that of other RISC architectures, including Sun's SPARC, IBM and Motorola's PowerPC, and ARM-based processors.
- Older processors used complex instruction sets computing, or CISC architectures.
  - Many powerful instructions were supported, making the assembly language programmer's job much easier.
  - But this meant that the processor was more complex, which made the hardware designer's life harder.
- Many new processors use reduced instruction sets computing, or RISC architectures.
  - Only relatively simple instructions are available. But with high-level languages and compilers, the impact on programmers is minimal.
  - On the other hand, the hardware is much easier to design, optimize, and teach in classes.
- Even most current CISC processors, such as Intel 8086-based chips, are now implemented using a lot of RISC techniques.

ISA

- An ISA consists of:
  - a set of operations (instructions) ← arithmetic, logical, conditional, branch, etc.
  - data units (sized, addressing modes, etc.) ← 32-bit data word
  - processor state (registers) ← 32, 32-bit registers
  - input and output control (memory operations) ← load and store
  - execution model (program counter) ← 32-bit program counter

(for MIPS)

RISC vs. CISC

- Characteristics of ISAs

CISC	RISC
Variable length instruction	Single word instruction
Variable format	Fixed-field decoding
Memory operands	Load/store architecture
Complex operations	Simple operations

MIPS Instruction Set (RISC)

- Instructions execute simple functions.
- Maintain regularity of format – each instruction is one word, contains *opcode* and *arguments*.
- Minimize memory accesses – whenever possible use registers as arguments.
- Three types of instructions:
  - Register (R)-type – only registers as arguments.
  - Immediate (I)-type – arguments are registers and numbers (constants or memory addresses).
  - Jump (J)-type – argument is an address.

Registers v. Memory

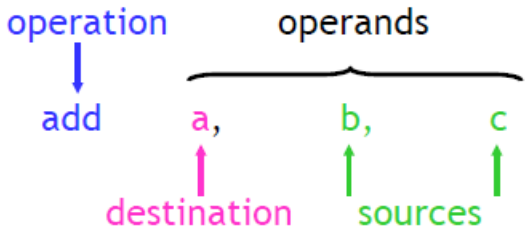
- Registers are faster to access than memory
- Operating on data in memory requires loads and stores
  - (More instructions to be executed)
- Compiler should use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important for performance

MIPS Arch

- MIPS = Microprocessor without Interlocked Pipeline Stages
- MIPS architecture developed at Stanford in 1984, spun out into MIPS Computer Systems
- As of 2004, over 300 million MIPS microprocessors had been sold
- Used in many commercial systems, including Silicon Graphics, Nintendo, and Cisco
- Once you've learned one architecture, it's easy to learn others.

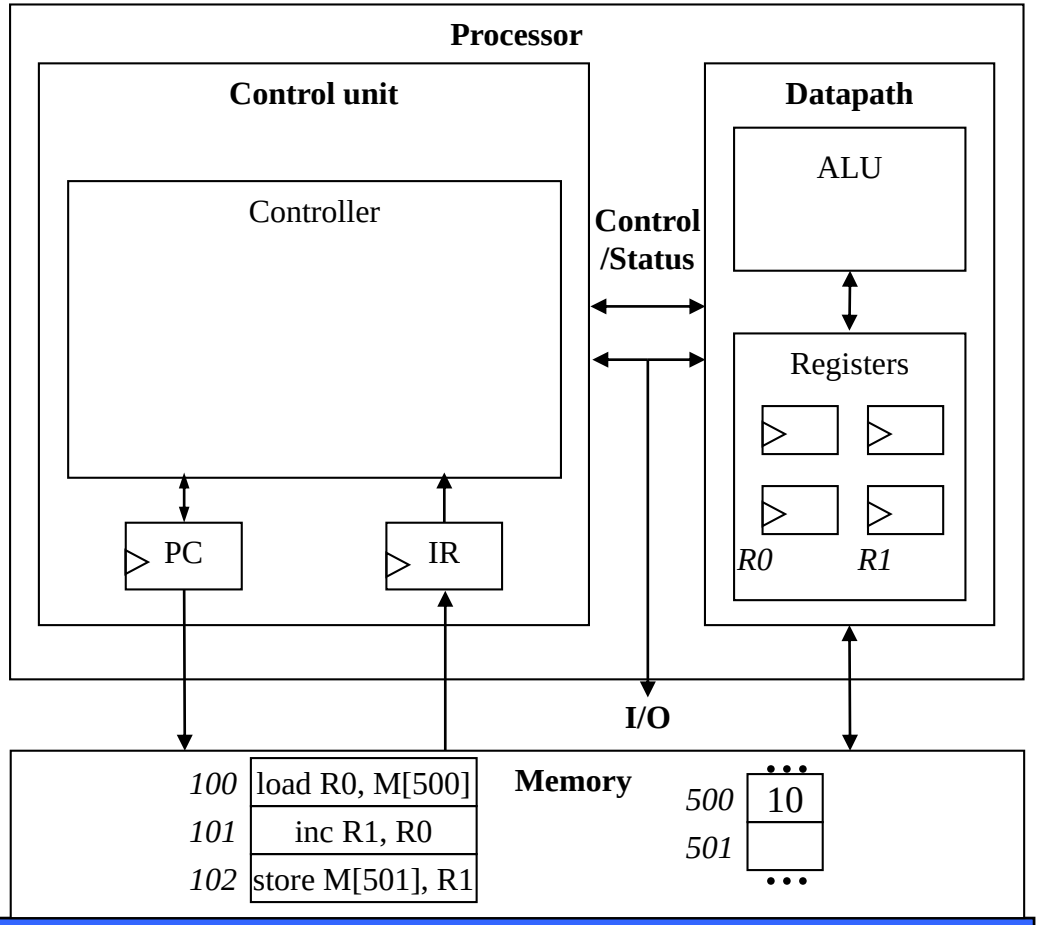
MIPS

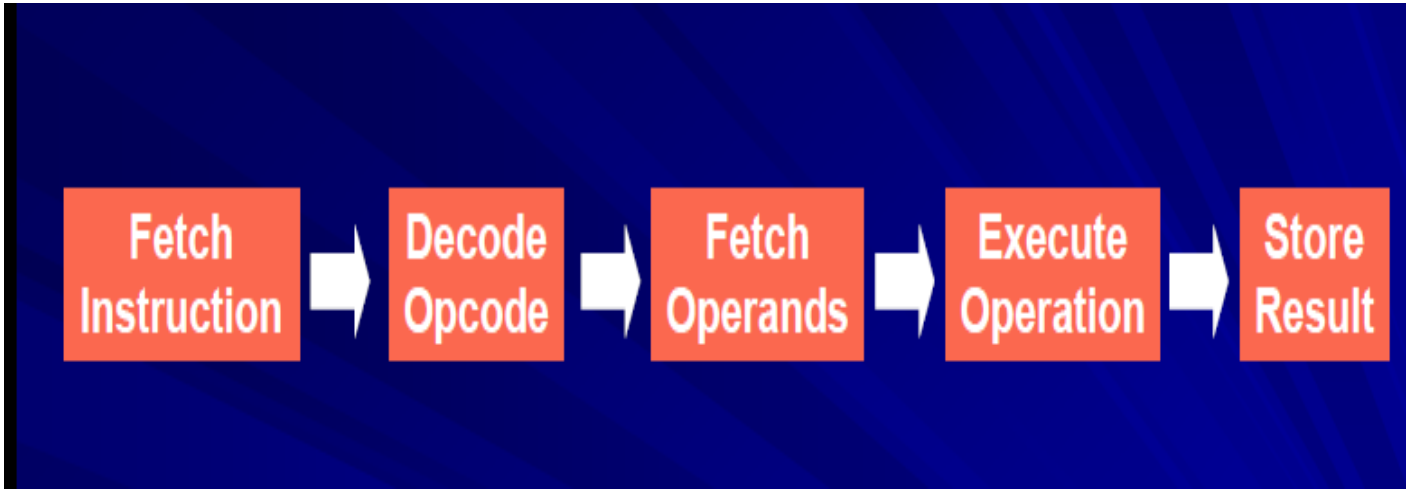
- MIPS is a register-to-register, or load/store, architecture.
  - The destination and sources must all be registers.
  - Special instructions, which we'll see later, are needed to access main memory.
- MIPS uses three-address instructions for data manipulation.
  - Each ALU instruction contains a destination and two sources.
  - For example, an addition instruction (a = b + c) has the form:



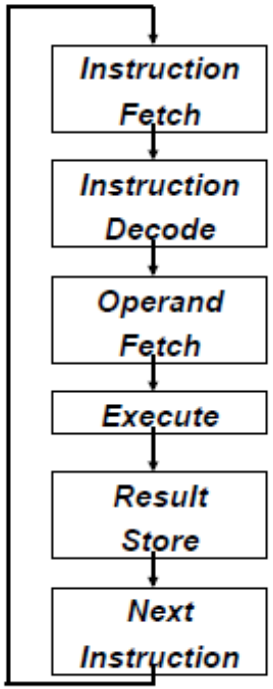
Control Unit

- Control unit: configures the datapath operations
  - Sequence of desired operations (“instructions”) stored in memory – “program”
- Instruction cycle – broken into several sub-operations, each one clock cycle, e.g.:
  - Fetch: Get next instruction into IR
  - Decode: Determine what the instruction means
  - Fetch operands: Move data from memory to datapath register
  - Execute: Move data through the ALU
  - Store results: Write data from register to memory




Fetch Instruction
Decode Opcode
Fetch Operands
Execute Operation
Store Result

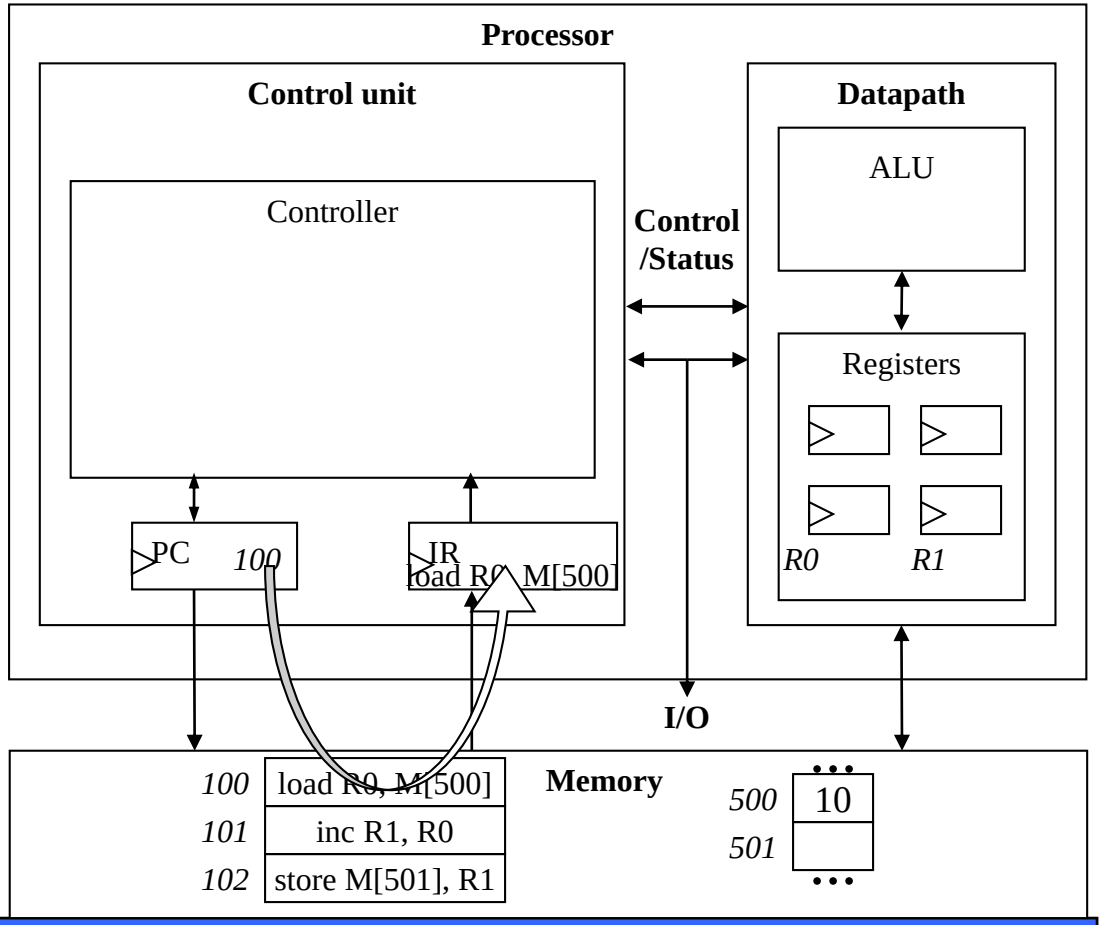
How are Instructions Executed?



- Instruction Fetch:
  Read instruction bits from memory
- Decode:
  Figure out what those bits mean
- Operand Fetch:
  Read registers (+ mem to get sources)
- Execute:
  Do the actual operation (e.g., add the #s)
- Result Store:
  Write result to register or memory
- Next Instruction:
  Figure out mem addr of next insn, repeat

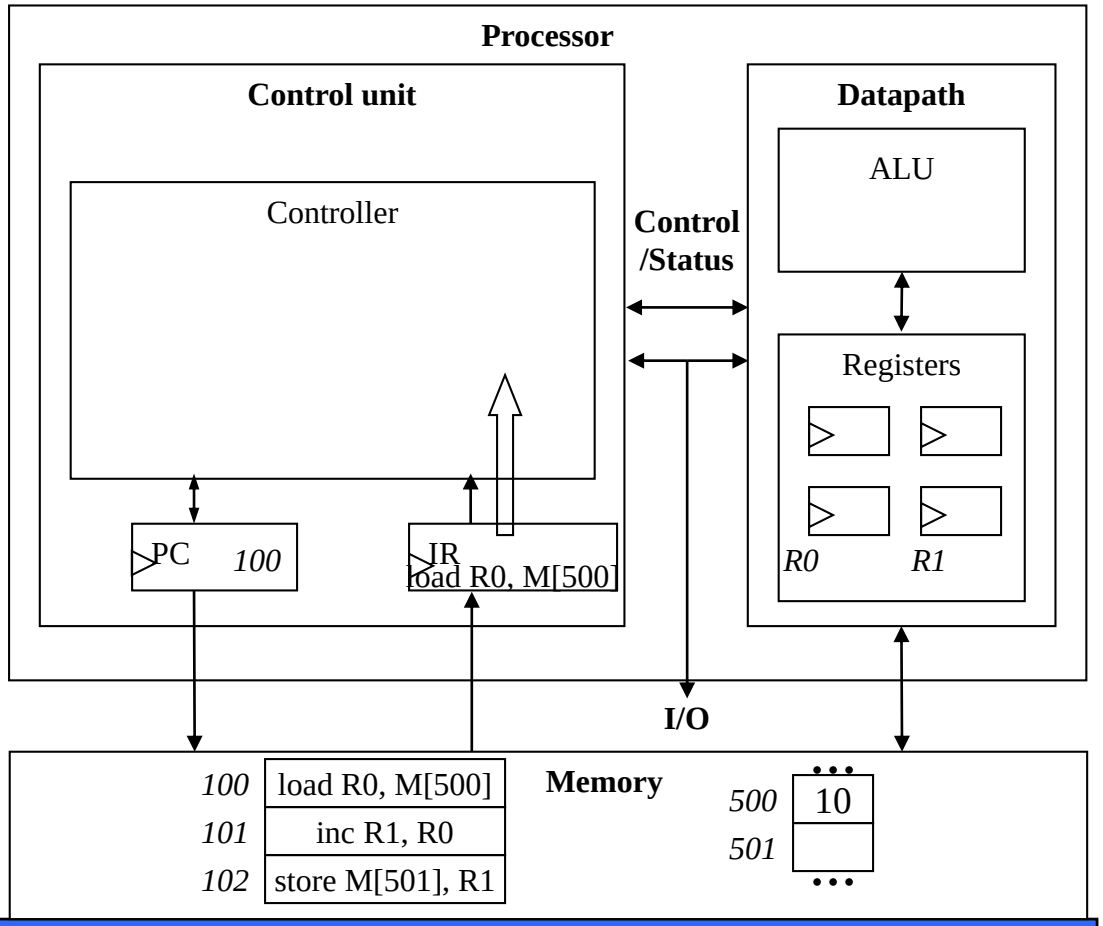
Control Unit Sub-Operations

- Fetch
  - Get next instruction into IR
  - PC: program counter, always points to next instruction
  - IR: holds the fetched instruction



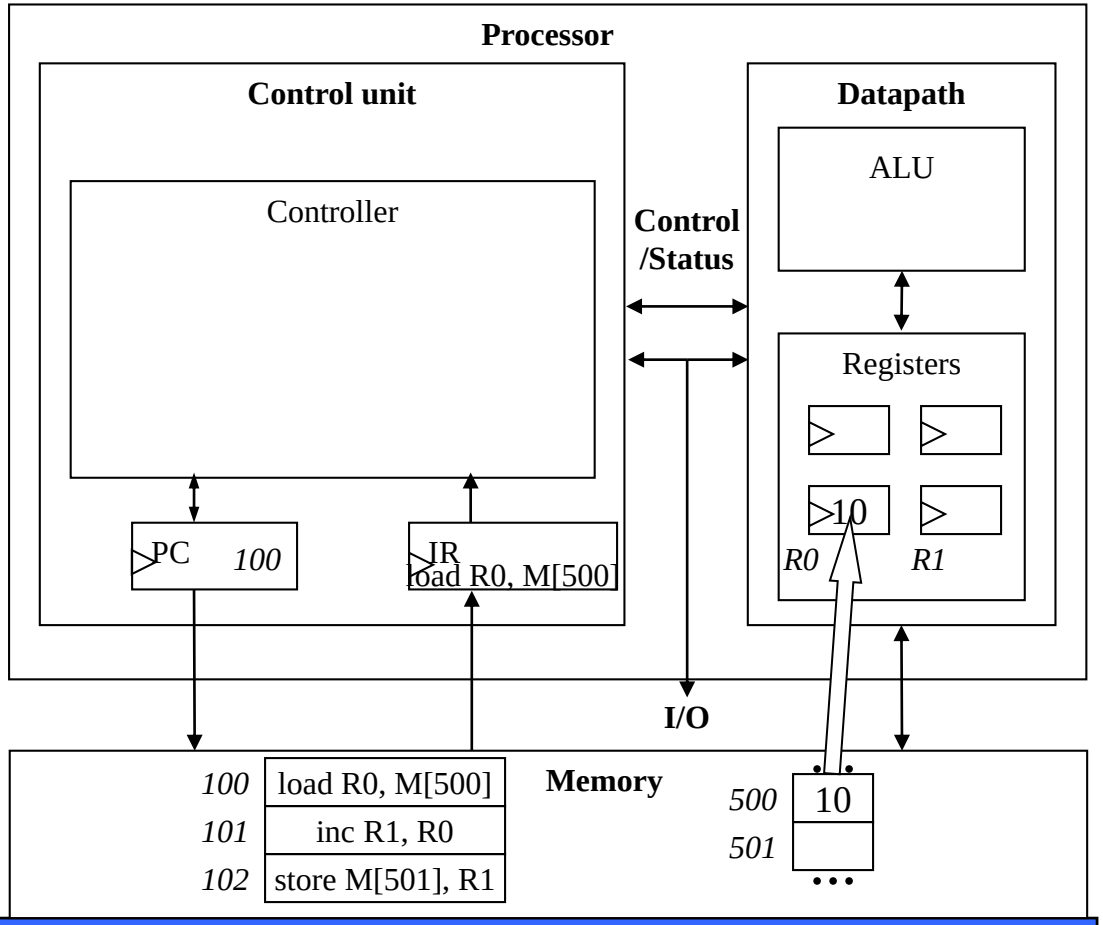
Control Unit Sub-Operations

- Decode
  - Determine what the instruction means



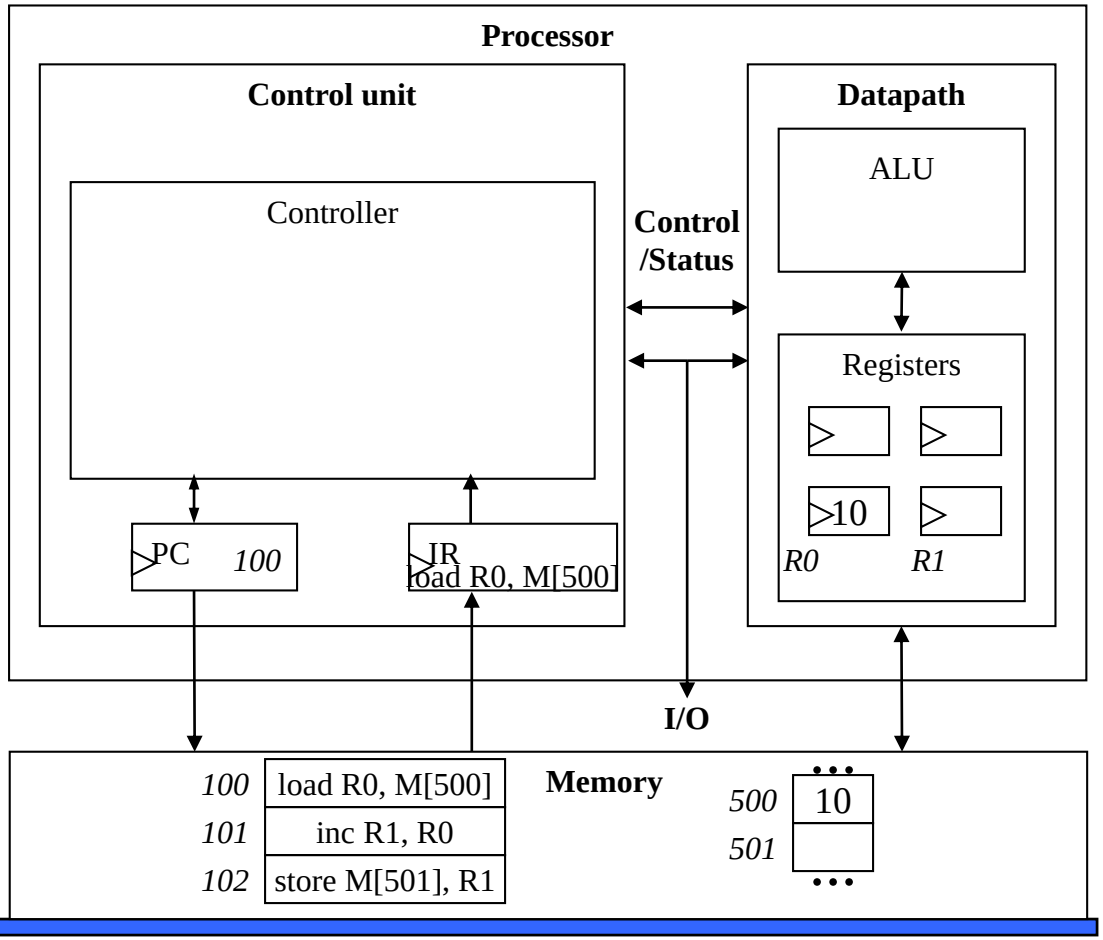
Control Unit Sub-Operations

- Fetch operands
  - Move data from memory to datapath register



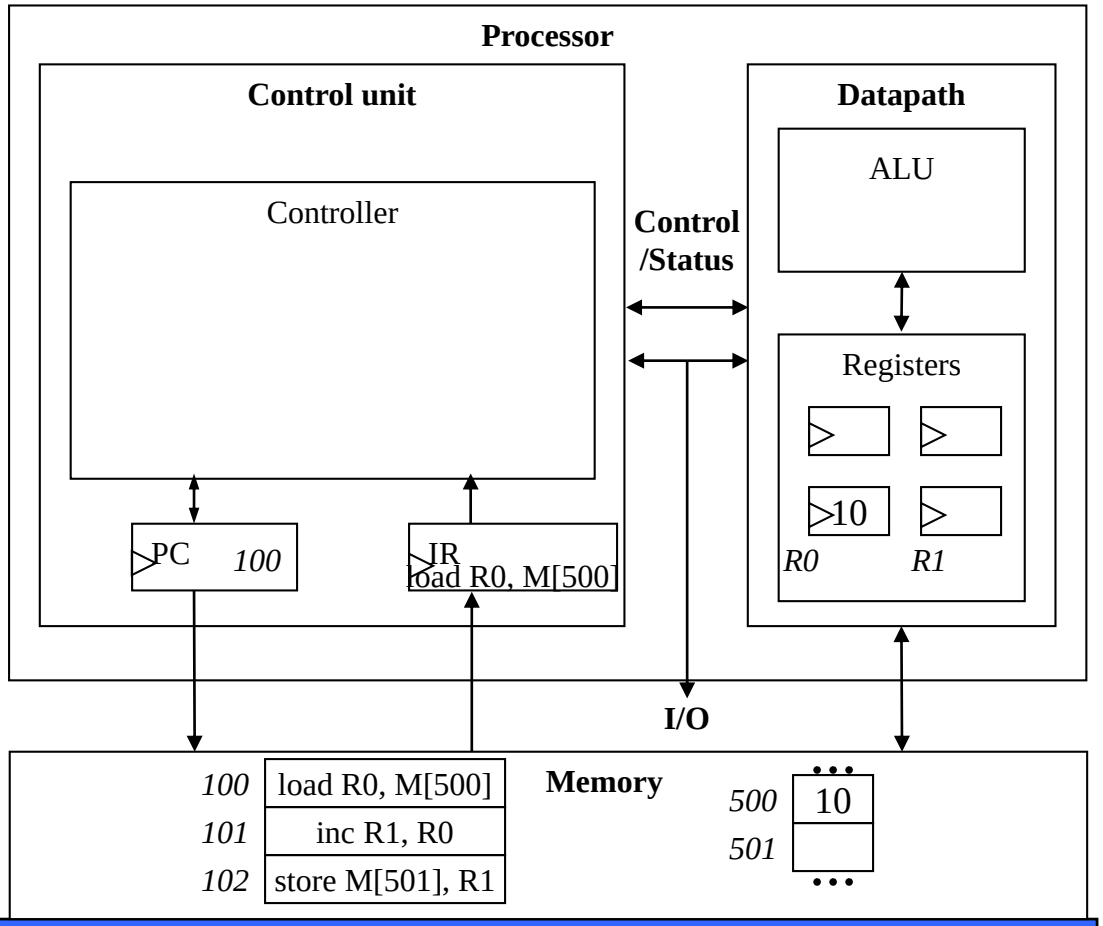
Control Unit Sub-Operations

- Execute
  - Move data through the ALU



Control Unit Sub-Operations

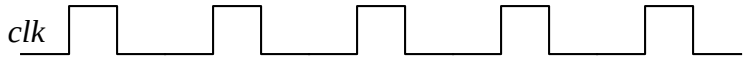
- Store results
  - Write data from register to memory

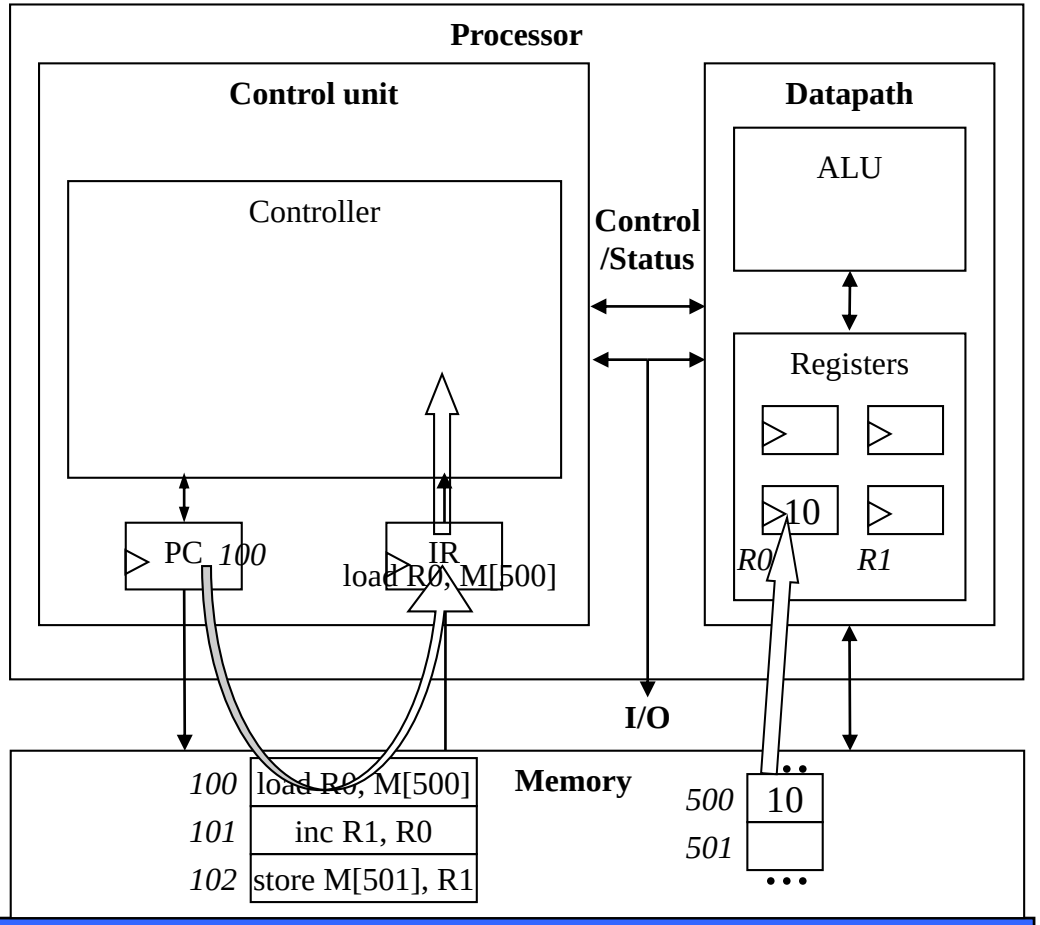


Instruction Cycles

PC=100

Fetch Decode Fetch ops Exec. Store results





Instruction Cycles

PC=100

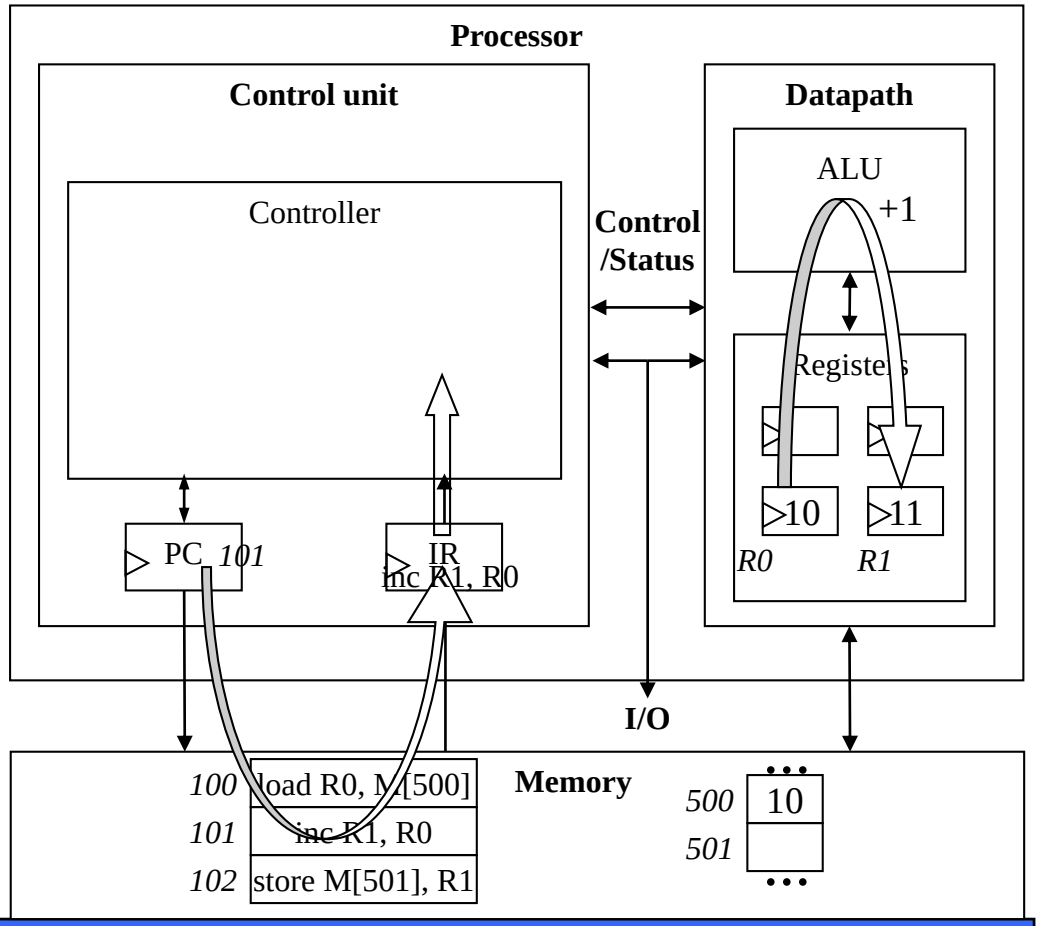
Fetch Decode Fetch ops Exec. Store results

clk

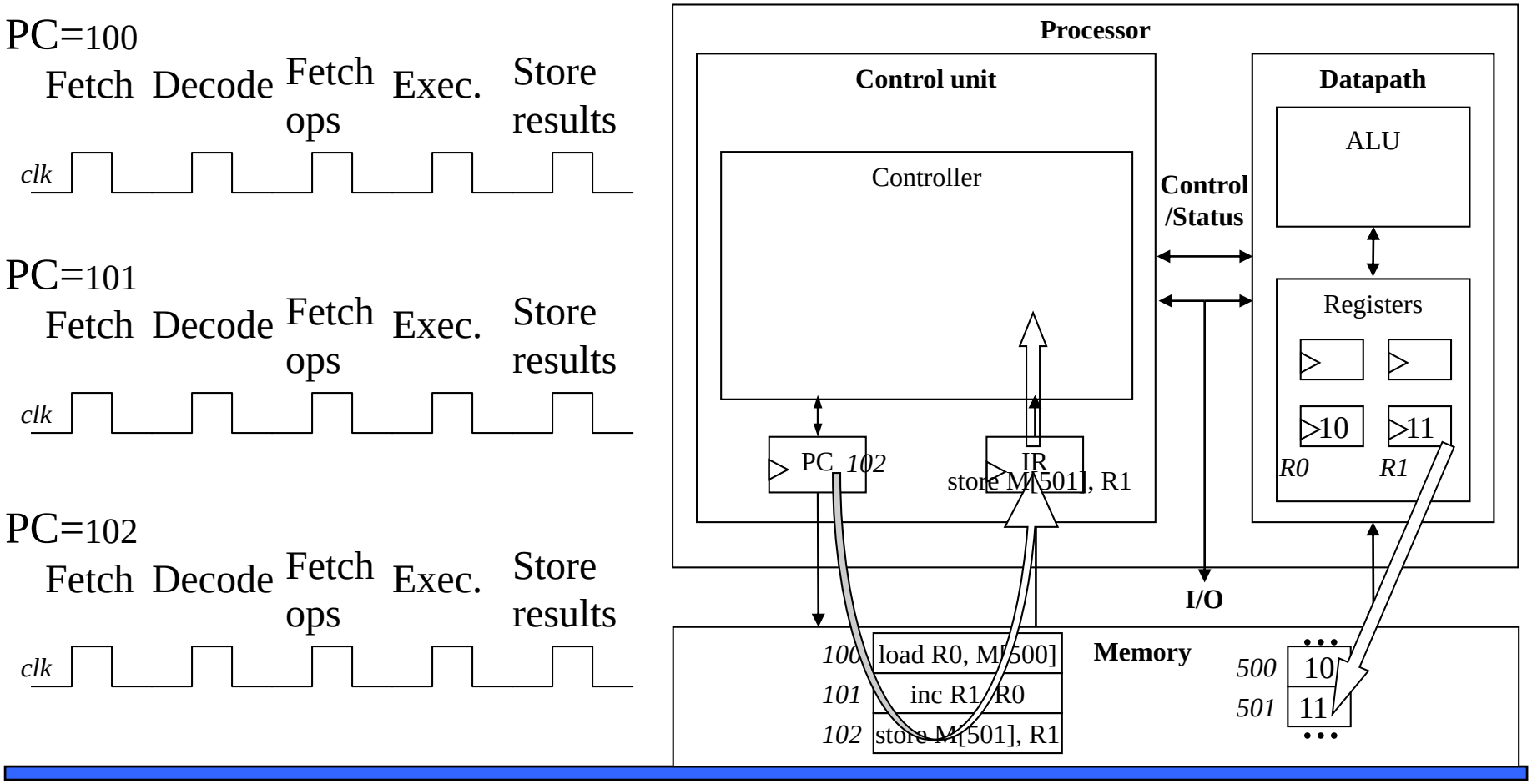
PC=101

Fetch Decode Fetch ops Exec. Store results

clk

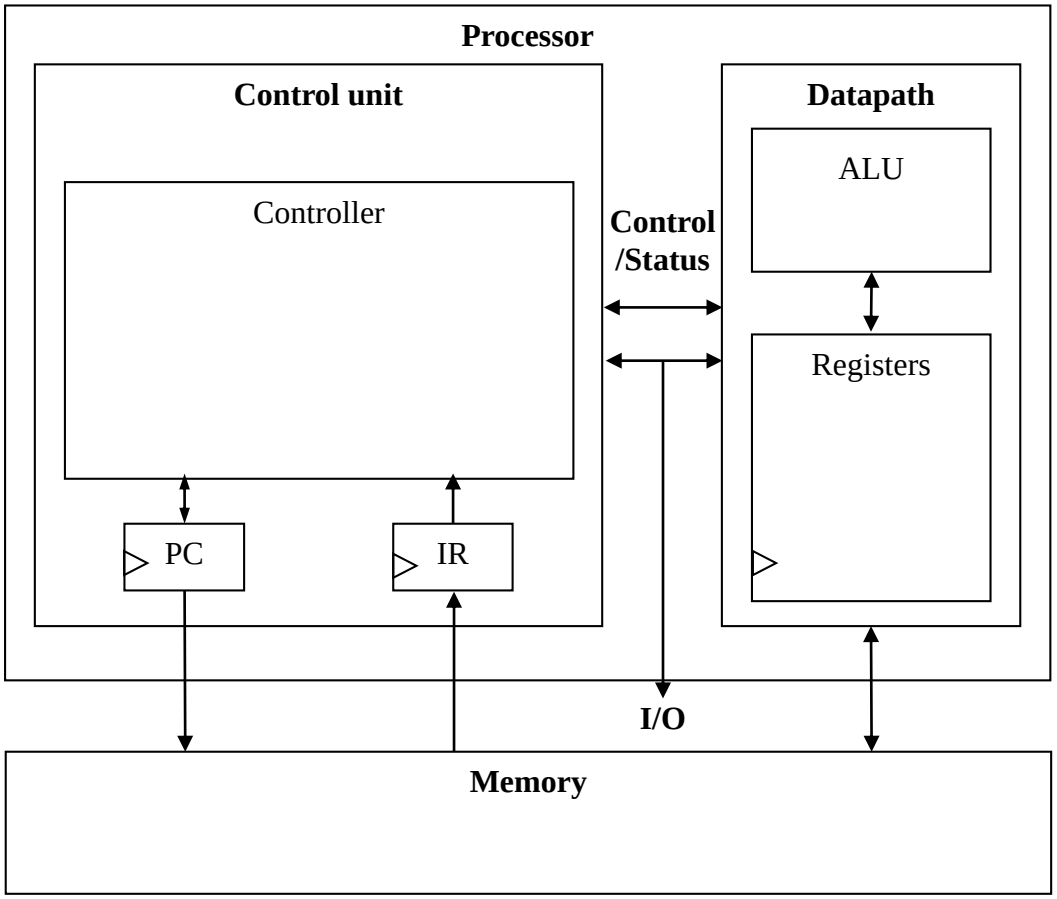


Instruction Cycles



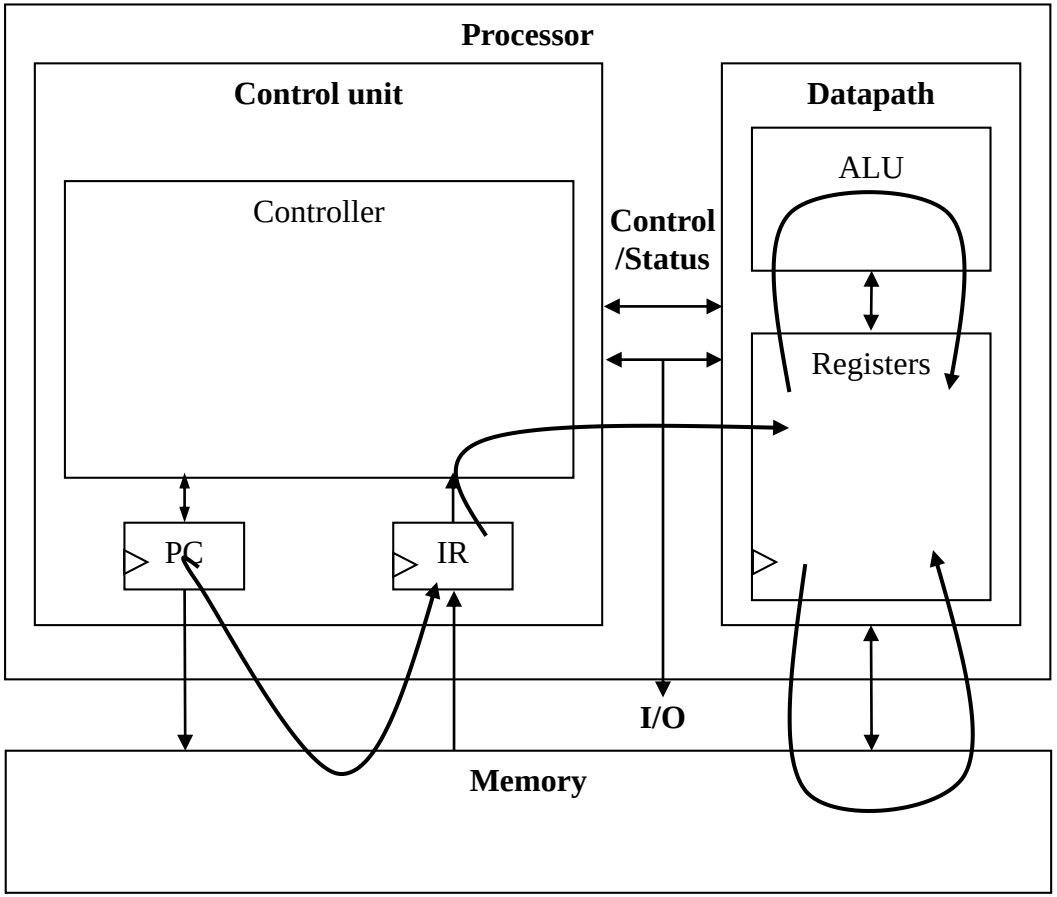
Architectural Considerations

- *N-bit* processor
  - N-bit ALU, registers, buses, memory data interface
  - Embedded: 8-bit, 16-bit, 32-bit common
  - Desktop/servers: 32-bit, even 64
- PC size determines address space



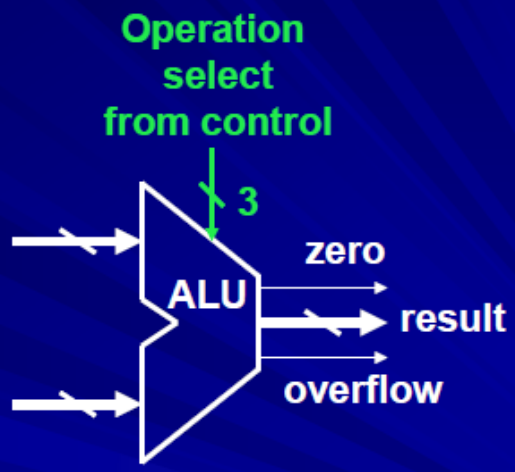
Architectural Considerations

- Clock frequency
  - Inverse of clock period
  - Must be longer than longest register to register delay in entire processor
  - Memory access is often the longest



Multi-Operation ALU

Operation select	ALU function
000	AND
001	OR
010	Add
110	Subtract
111	Set on less than



zero = 1, when all bits of result are 0

Instructions: Addition

C Code

```
a = b + c;
```

MIPS assembly code

```
add a, b, c
```

- **add:** mnemonic indicates operation to perform
- **b, c:** source operands (on which the operation is performed)
- **a:** destination operand (to which the result is written)

MIPS Arithmetic Instructions

- All instructions have 3 operands
- Operand order is fixed (destination first)

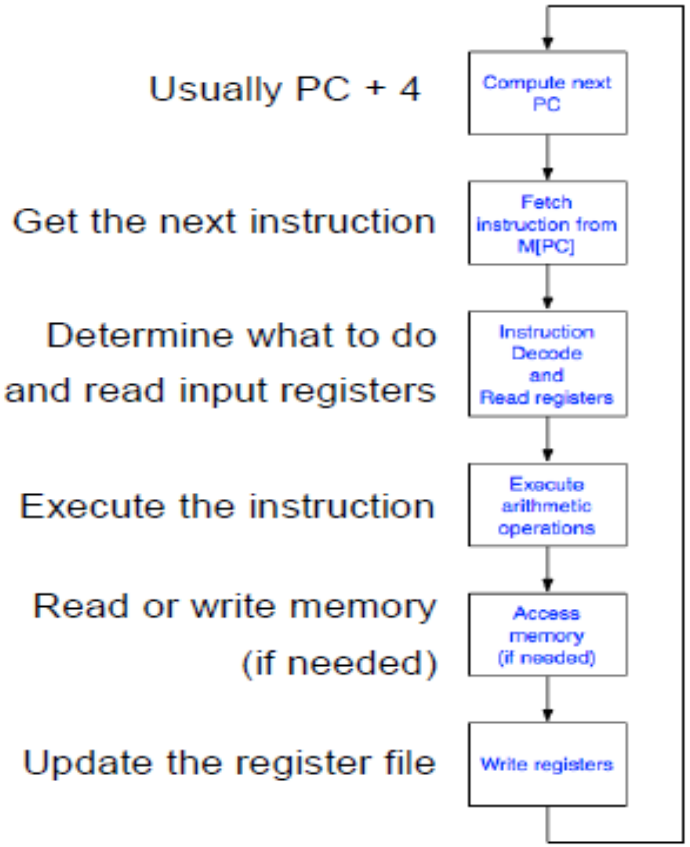
Example:

C code: `a = b + c;`

MIPS 'code': `add a, b, c`

"The natural number of operands for an operation like addition is three… requiring every instruction to have exactly three operands conforms to the philosophy of keeping the hardware simple"

Executing a MIPS program



- All instructions have
  - <= 1 arithmetic op
  - <= 1 memory access
  - <= 2 register reads
  - <= 1 register write
  - <= 1 branch
- All instructions go through all the steps
- As a result
  - Implementing MIPS is (sort of) easy!
  - The resulting HW is (relatively) simple!

Instructions: Subtraction

- Similar to addition - only mnemonic changes

C Code

```
a = b - c;
```

MIPS assembly code

```
sub a, b, c
```

- **sub:** mnemonic
- **b, c:** source operands
- **a:** destination operand

Design Principle 1

Simplicity favors regularity

- Consistent instruction format
- Same number of operands (two sources and one destination)
- Easier to encode and handle in hardware

Multiple Instructions

- More complex code is handled by multiple MIPS instructions.

C Code

```
a = b + c - d;
```

MIPS assembly code

```
add t, b, c  # t = b + c
sub a, t, d  # a = t - d
```

Arithmetic Instr. (Continued)

- Design Principle: simplicity favors regularity.
- Of course this complicates some things...

C code: `a = b + c + d;`

MIPS code:
```
add a, b, c
add a, a, d
```

- Operands must be registers (why?) *Remember von Neumann bottleneck.*
- 32 registers provided
- Each register contains 32 bits

Design Principle 2

Make the common case fast

- MIPS includes only simple, commonly used instructions
- Hardware to decode and execute instructions can be simple, small, and fast
- More complex instructions (that are less common) performed using multiple simple instructions
- MIPS is a ***reduced instruction set computer*** **(RISC)**, with a small number of simple instructions
- Other architectures, such as Intel's x86, are ***complex instruction set computers*** **(CISC)**

Operands

- Operand location: physical location in computer
  - Registers
  - Memory
  - Constants (also called *immediates*)

Operands: Registers

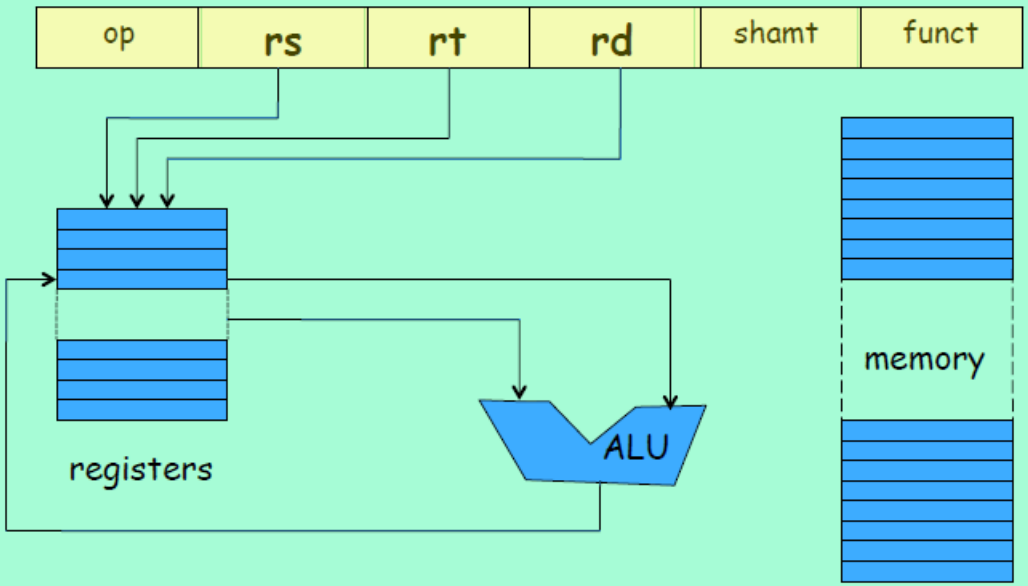
- MIPS has 32 32-bit registers
- Registers are faster than memory
- MIPS called “32-bit architecture” because it operates on 32-bit data

Design Principle 3

Smaller is Faster

- MIPS includes only a small number of registers

Register addressing



MIPS register names

- MIPS register names begin with a $. There are two naming conventions:
  - By number:

    $0 $1 $2 … $31
  - By (mostly) two-character names, such as:

    $a0-$a3 $s0-$s7 $t0-$t9 $sp $ra

- Not all of the registers are equivalent:
  - E.g., register $0 or $zero always contains the value 0
    - (go ahead, try to change it)

- Other registers have special uses, by convention:
  - E.g., register $sp is used to hold the "stack pointer"

Operands: Registers

- Registers:
  - `$` before name
  - Example: `$0`, “register zero”, “dollar zero”
- Registers used for specific purposes:
  - `$0` always holds the constant value 0.
  - the *saved registers*, `$s0`–`$s7`, used to hold variables
  - the *temporary registers*, `$t0` - `$t9`, used to hold intermediate values during a larger computation
  - Discuss others later

MIPS Register Set

Name	Register Number	Usage
`$0`	0	the constant value 0
`$at`	1	assembler temporary
`$v0-$v1`	2-3	Function return values
`$a0-$a3`	4-7	Function arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved variables
`$t8-$t9`	24-25	more temporaries
`$k0-$k1`	26-27	OS temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	Function return address

Instruction Format

- **R-Type** : instructions use opcode **000000**.
  - This group Contains all instructions that do not require an immediate value, target offset, memory address displacement, or memory address to specify an operand.
  - Includes arithmetic and logic with all **operands in registers**, shift instructions, and register direct jump instructions (jalr and jr).
- **I-Type** : Opcodes **except 000000, 00001x, and 0100xx**
  - This group includes instructions with **an immediate operand**, branch instructions, and load and store instructions. In the MIPS architecture, all memory accesses are handled by the main processor, so coprocessor load and store instructions are included in this group.
- **J-Type** : instructions use opcode **00001x**.
  - This group consists of the two **direct jump instructions** (j and jal).
  - These instructions require a memory address to specify their operand.

12

Instructions with Registers

- Revisit add instruction

C Code

```
a = b + c
```

MIPS assembly code

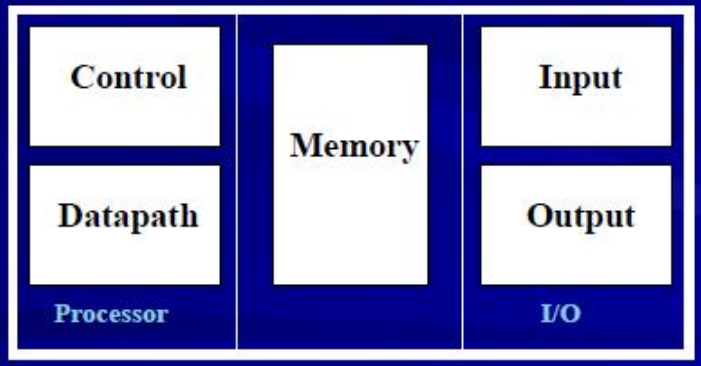
```
# $s0 = a, $s1 = b, $s2 = c
add $s0, $s1, $s2
```

Operands: Memory

- Too much data to fit in only 32 registers
- Store more data in memory
- Memory is large, but slow
- Commonly used variables kept in registers

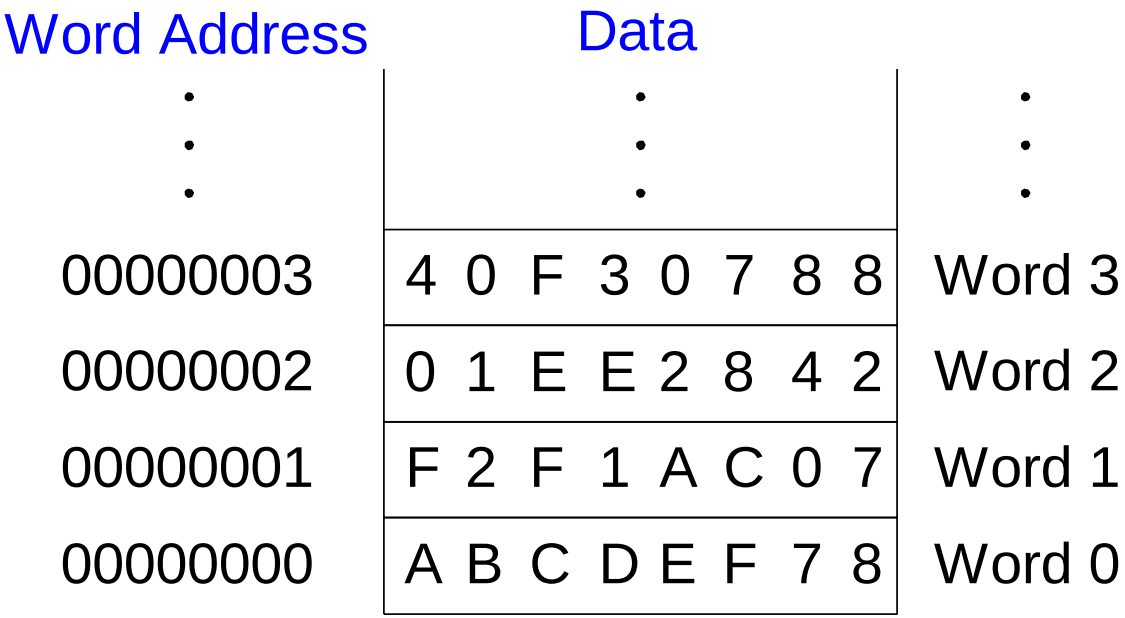
Registers vs. Memory

- Arithmetic instructions operands must be registers
  - **32 registers provided**
- Compiler associates variables with registers.
- What about programs with lots of variables? *Must use memory.*



Word-Addressable Memory

- Each 32-bit data word has a unique address



Note: MIPS uses byte-addressable memory, which we'll talk about next.

Reading Word-Addressable Memory

- Memory read called ***load***
- **Mnemonic:** *load word* (`lw`)
- **Format:**

```
lw $s0, 5($t1)
```

- **Address calculation:**
  - add *base address* (`$t1`) to the *offset* (5)
  - address = (`$t1` + 5)
- **Result:**
  - `$s0` holds the value at address (`$t1` + 5)

Any register may be used as base address

Reading Word-Addressable Memory

- **Example:** read a word of data at memory address 1 into `$s3`
  - address = (`$0` + 1) = 1
  - `$s3` = 0xF2F1AC07 after load

Assembly code

```
lw $s3, 1($0)   # read memory word 1 into $s3
```

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Writing Word-Addressable Memory

- Memory write are called ***store***
- **Mnemonic:** *store word* (`sw`)

Writing Word-Addressable Memory

- **Example:** Write (store) the value in `$t4` into memory address 7
  - add the base address (`$0`) to the offset (0x7)
  - address: ($0 + 0x7) = 7

Offset can be written in decimal (default) or hexadecimal

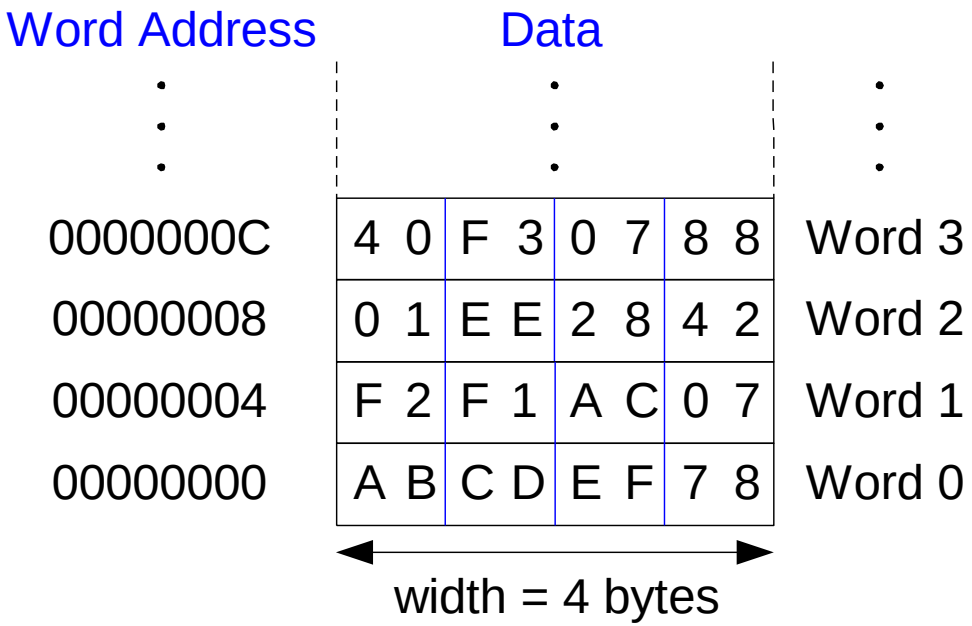
Assembly code

```
sw $t4, 0x7($0)   # write the value in $t4
                  # to memory word 7
```

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Byte-Addressable Memory

- Each data byte has unique address
- Load/store words or single bytes: load byte (`lb`) and store byte (`sb`)
- 32-bit word = 4 bytes, so word address increments by 4



Reading Byte-Addressable Memory

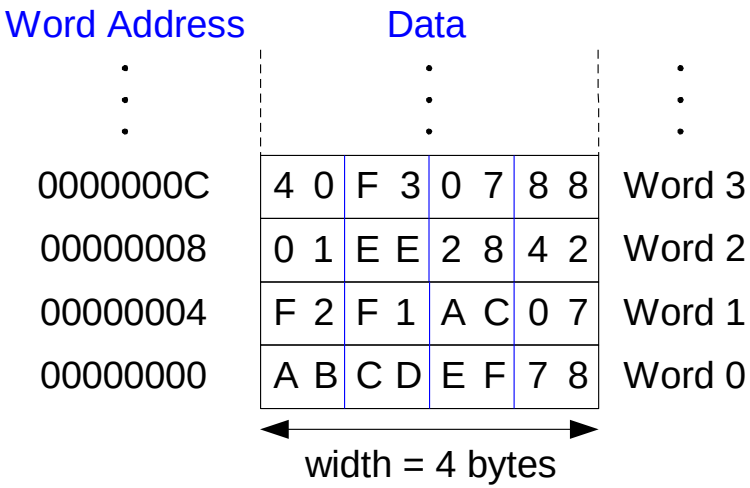
- The address of a memory word must now be multiplied by 4. For example,
  - the address of memory word 2 is $2 \times 4 = 8$
  - the address of memory word 10 is $10 \times 4 = 40$ (0x28)
- **MIPS is byte-addressed, not word-addressed**

Reading Byte-Addressable Memory

- **Example:** Load a word of data at memory address 4 into `$s3`.
- `$s3` holds the value 0xF2F1AC07 after load

MIPS assembly code

```
lw $s3, 4($0)   # read word at address 4 into $s3
```

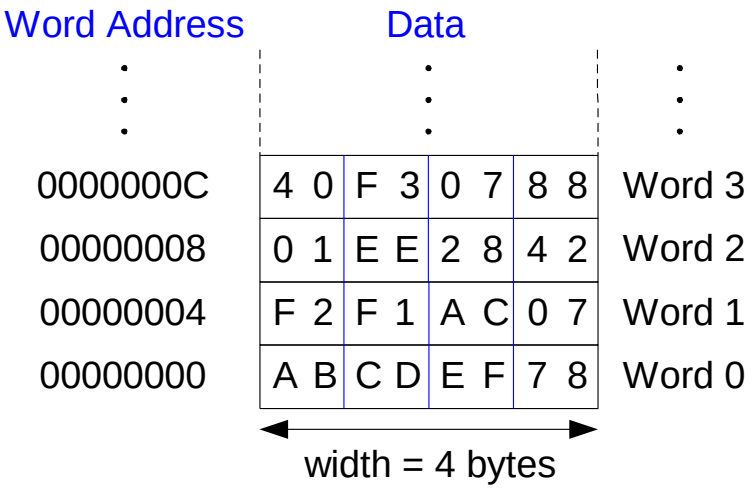


Writing Byte-Addressable Memory

- **Example:** stores the value held in `$t7` into memory address 0x2C (44)

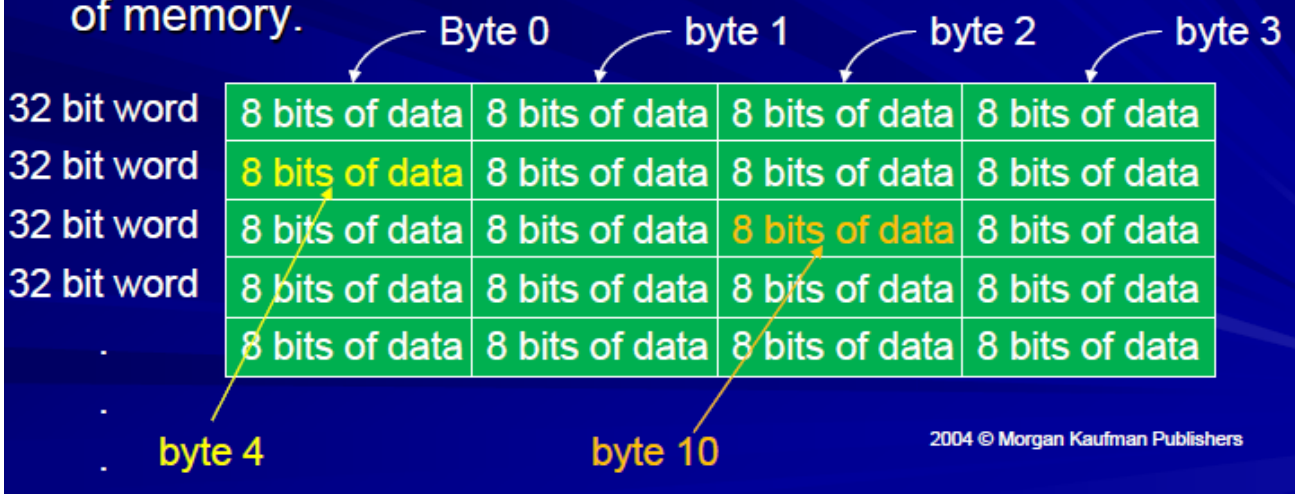
MIPS assembly code

```
sw $t7, 44($0)   # write $t7 into address 44
```



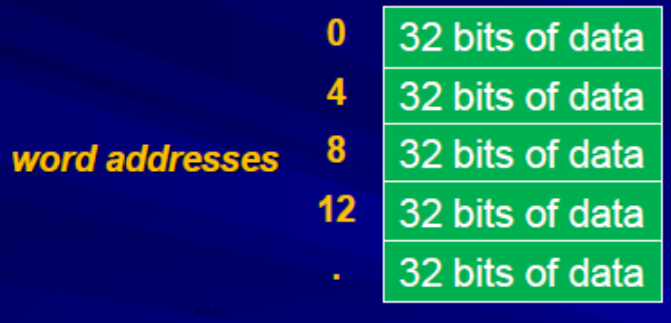
Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array.
- "Byte addressing" means that the index points to a byte of memory.



Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word contains 32 bits or 4 bytes.



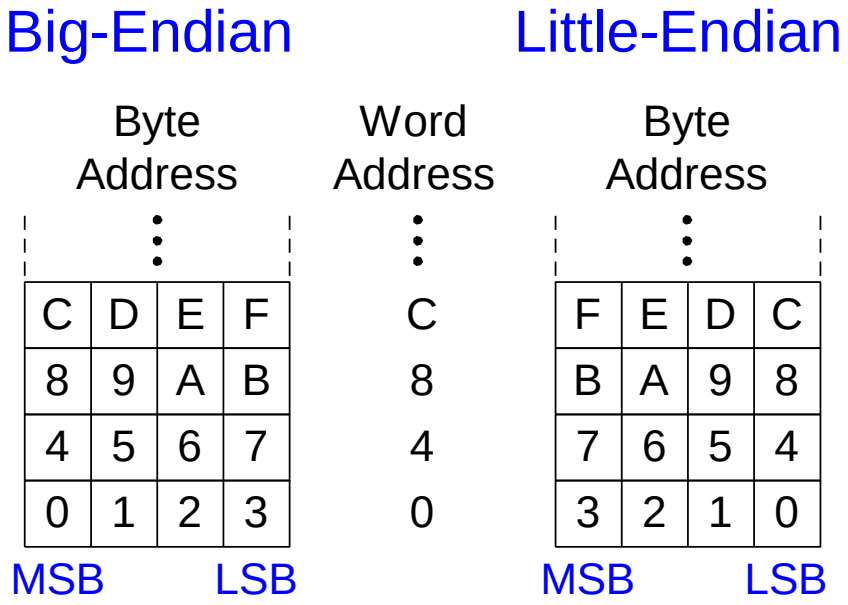
Registers hold 32 bits of data

Use 32 bit address

- 2^{32} bytes with addresses from 0 to $2^{32} - 1$
- 2^{30} words with addresses 0, 4, 8, ... $2^{32} - 4$
- Words are aligned
  i.e., what are the least 2 significant bits of a word address?

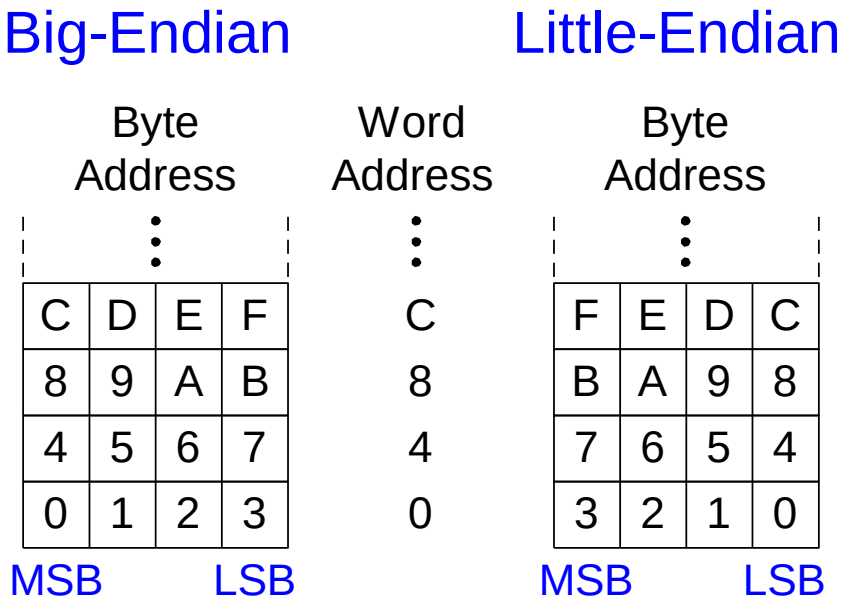
Big-Endian & Little-Endian Memory

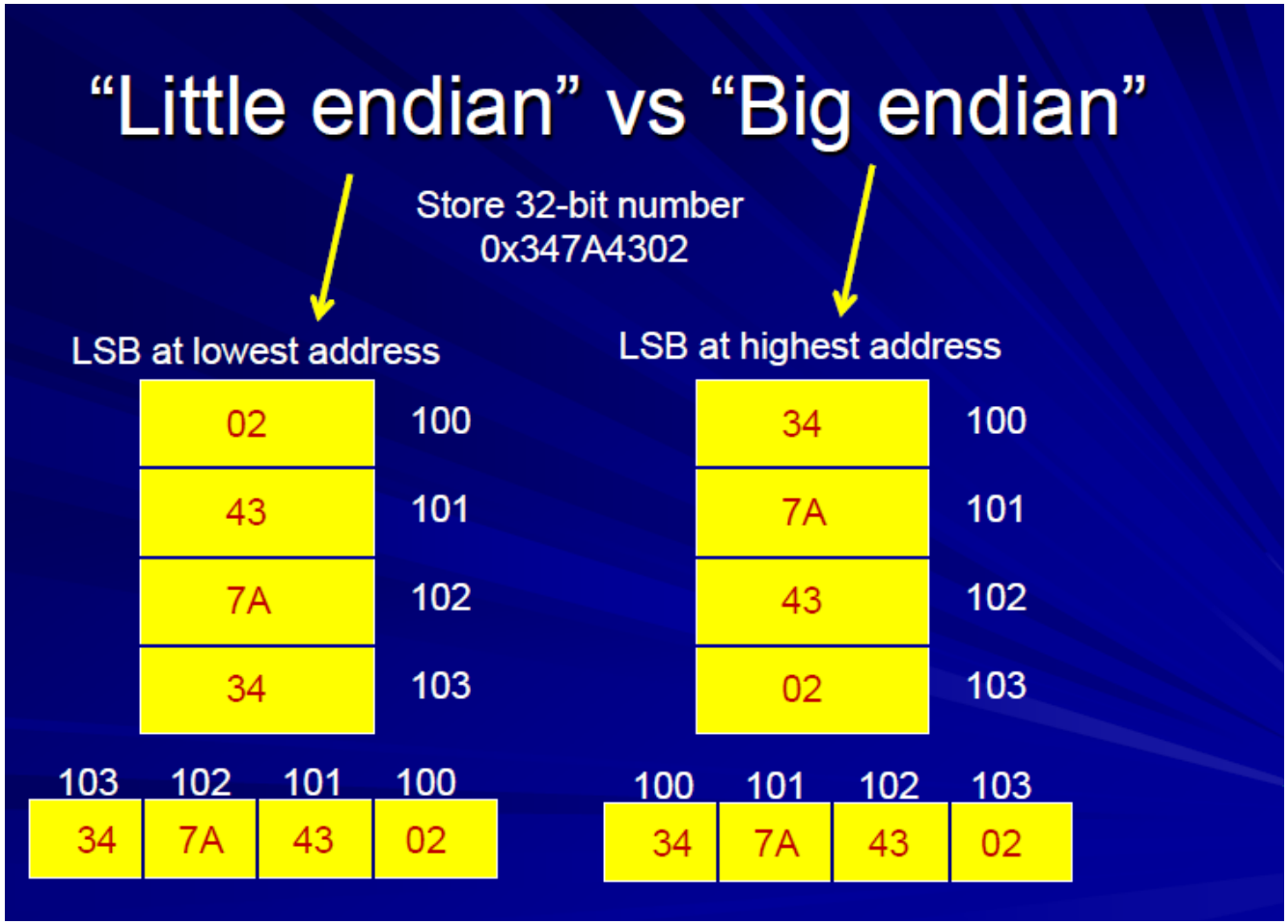
- How to number bytes within a word?
- **Little-endian:** byte numbers start at the little (least significant) end
- **Big-endian:** byte numbers start at the big (most significant) end
- **Word address** is the **same** for big- or little-endian



Big-Endian & Little-Endian Memory

- Jonathan Swift's *Gulliver's Travels*: the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end
- It doesn't really matter which addressing type used – except when the two systems need to share data!



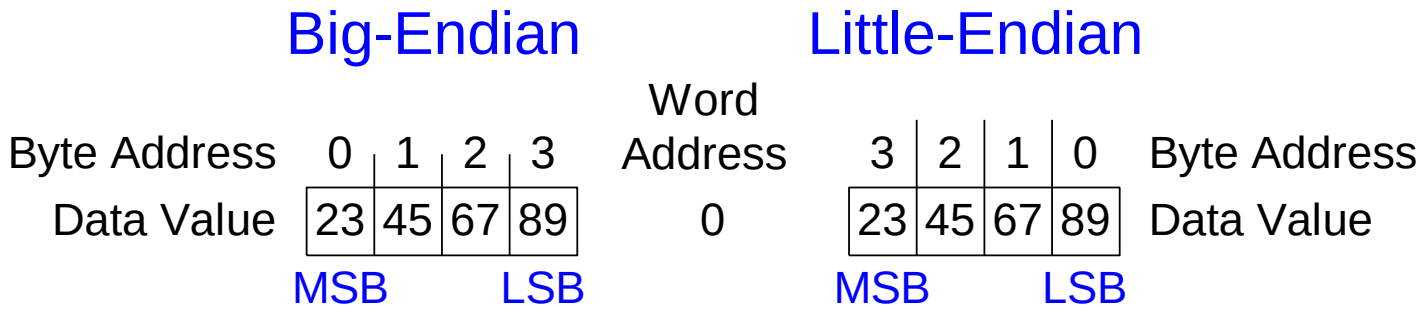

"Little endian" vs "Big endian"
Store 32-bit number
0x347A4302
LSB at lowest address
02
43
7A
34
100
101
102
103
103 102 101 100
34 7A 43 02
LSB at highest address
34
7A
43
02
100
101
102
103
100 101 102 103
34 7A 43 02

Big-Endian & Little-Endian Example

- Suppose `$t0` initially contains 0x23456789
- After following code runs on big-endian system, what value is `$s0`?
- In a little-endian system?

```
sw $t0, 0($0)
lb $s0, 1($0)
```

- Big-endian: 0x00000045
- Little-endian: 0x00000067



Design Principle 4

Good design demands good compromises

- Multiple instruction formats allow flexibility
  - `add`, `sub`: use 3 register operands
  - `lw`, `sw`: use 2 register operands and a constant
- Number of instruction formats kept small
  - to adhere to design principles 1 and 3 (simplicity favors regularity and smaller is faster).

Operands: Constants/Immediates

- `lw` and `sw` use constants or *immediates*
- *immediate*ly available from instruction
- 16-bit two's complement number
- `addi`: add immediate
- Subtract immediate (`subi`) necessary?

C Code

```
a = a + 4;
b = a - 12;
```

MIPS assembly code

```
# $s0 = a, $s1 = b
addi $s0, $s0, 4
addi $s1, $s0, -12
```

Machine Language

- Binary representation of instructions
- Computers only understand 1's and 0's
- 32-bit instructions
  - Simplicity favors regularity: 32-bit data & instructions
- 3 instruction formats:
  - **R-Type:** register operands
  - **I-Type:** immediate operand
  - **J-Type:** for jumping (discuss later)

R-Type

- *Register-type*
- 3 register operands:
  - `rs`, `rt`: source registers
  - `rd`: destination register
- Other fields:
  - `op`: the *operation code* or *opcode* (0 for R-type instructions)
  - `funct`: the *function* with opcode, tells computer what operation to perform
  - `shamt`: the *shift amount* for shift instructions, otherwise it's 0

R-Type

R-Type Examples

Assembly Code

```
add $s0, $s1, $s2
sub $t0, $t3, $t5
```

Field Values

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Note the order of registers in the assembly code:

```
add rd, rs, rt
```

I-Type

- *Immediate-type*
- 3 operands:
  - `rs`, `rt`: register operands
  - `imm`: 16-bit two's complement immediate
- Other fields:
  - `op`: the opcode
  - Simplicity favors regularity: all instructions have opcode
  - Operation is completely determined by opcode

I-Type

I-Type Examples

Assembly Code

```
addi $s0, $s1, 5
addi $t0, $s3, -12
lw   $t2, 32($0)
sw   $s1,  4($t1)
```

Field Values

op	rs	rt	imm
8	17	16	5
8	19	8	-12
35	0	10	32
43	9	17	4
6 bits	5 bits	5 bits	16 bits

Note the differing order of registers in assembly and machine codes:

```
addi rt, rs, imm
lw   rt, imm(rs)
sw   rt, imm(rs)
```

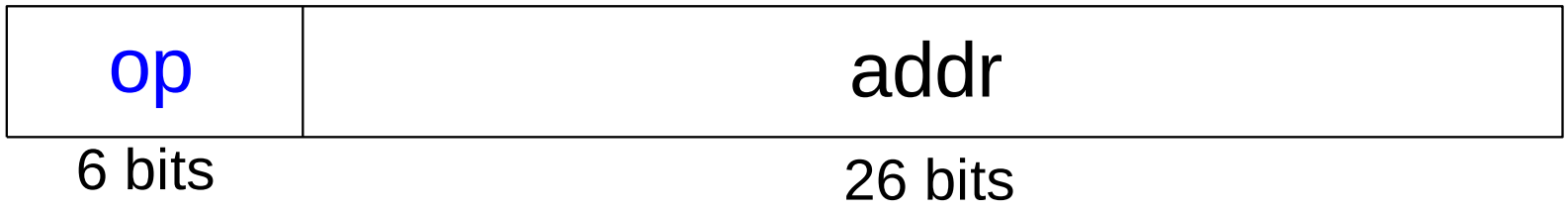
Machine Code

op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
6 bits	5 bits	5 bits	16 bits	

Machine Language: J-Type

- *Jump-type*
- 26-bit address operand (`addr`)
- Used for jump instructions (`j`)

J-Type

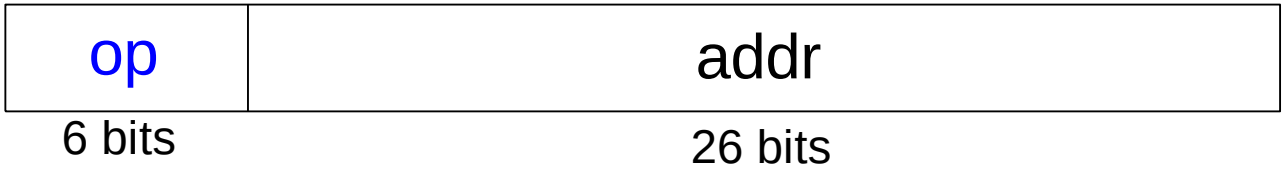


Review: Instruction Formats

R-Type

I-Type

J-Type



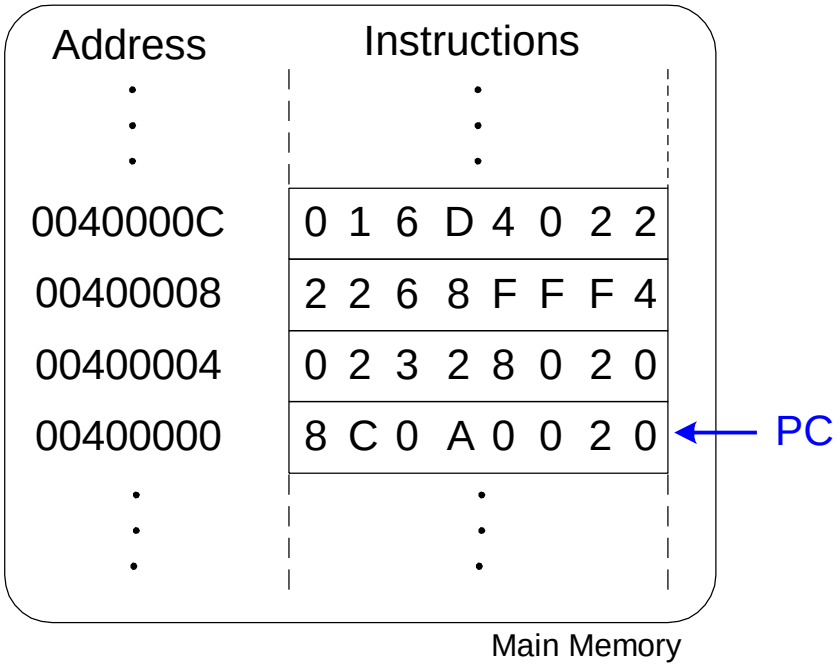
Power of the Stored Program

- 32-bit instructions & data stored in memory
- Sequence of instructions: only difference between two applications
- To run a new program:
  - No rewiring required
  - Simply store new program in memory
- Program Execution:
  - Processor *fetches* (reads) instructions from memory in sequence
  - Processor performs the specified operation

The Stored Program

Assembly Code	Machine Code
`lw $t2, 32($0)`	`0x8C0A0020`
`add $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub $t0, $t3, $t5`	`0x016D4022`

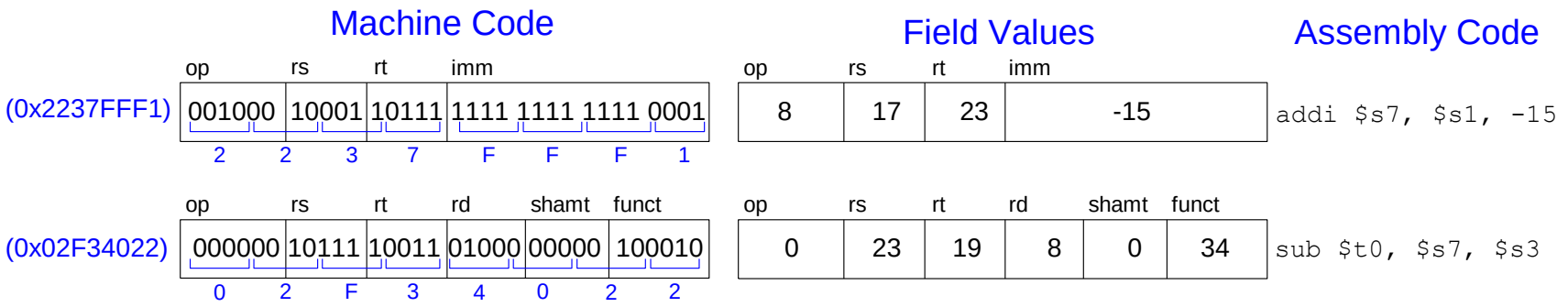
Stored Program



Program Counter (PC): keeps track of current instruction

Interpreting Machine Code

- Start with opcode: tells how to parse rest
- If opcode all 0's
  - R-type instruction
  - Function bits tell operation
- Otherwise
  - opcode tells operation



Programming

- High-level languages:
  - e.g., C, Java, Python
  - Written at higher level of abstraction
- Common high-level software constructs:
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

Ada Lovelace, 1815-1852

- Wrote the first computer program
- Her program calculated the Bernoulli numbers on Charles Babbage's Analytical Engine
- She was the daughter of the poet Lord Byron

Logical Instructions

- **`and`, `or`, `xor`, `nor`**
  - `and`: useful for **masking** bits
    - Masking all but the least significant byte of a value:
      0xF234012F AND 0x000000FF = 0x0000002F
  - `or`: useful for **combining** bit fields
    - Combine 0xF2340000 with 0x000012BC:
      0xF2340000 OR 0x000012BC = 0xF23412BC
  - `nor`: useful for **inverting** bits:
    - A NOR $0 = NOT A
- **`andi`, `ori`, `xori`**
  - 16-bit immediate is zero-extended (*not* sign-extended)
  - `nori` not needed

Logical Instructions Example 1

Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code

```
and $s3, $s1, $s2
or  $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

Result

$s3								
$s4								
$s5								
$s6								

Logical Instructions Example 1

Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

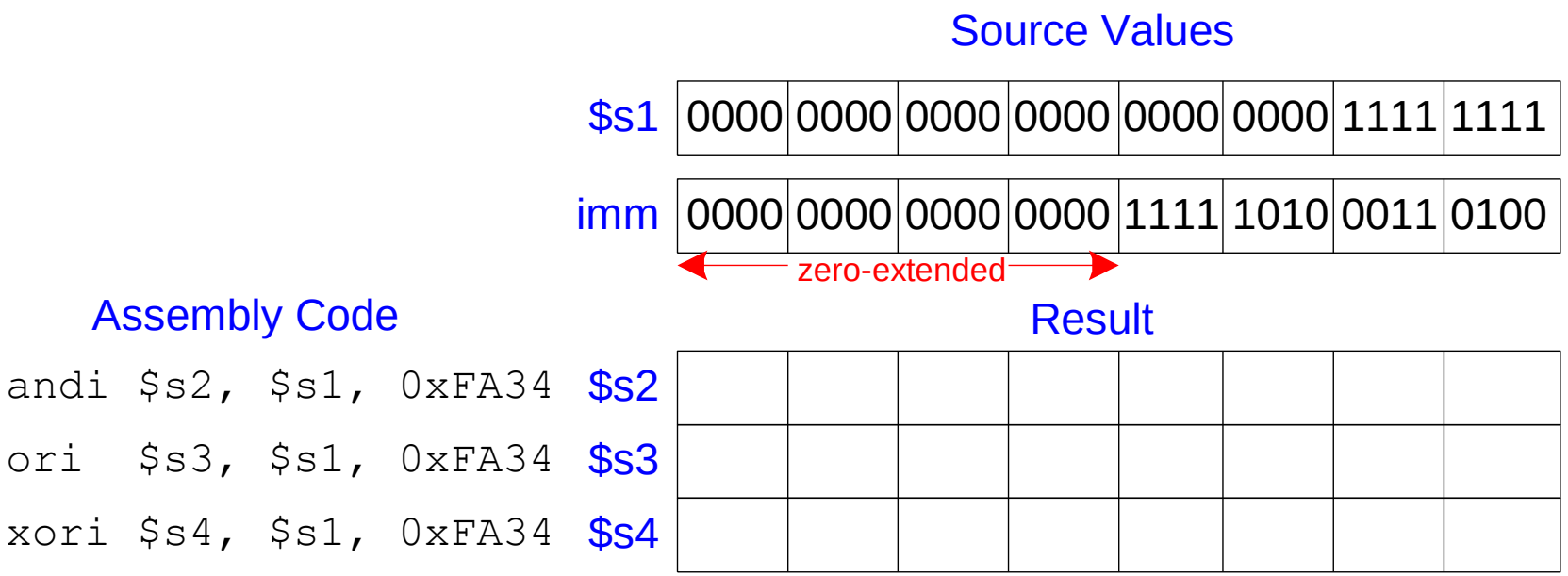
Assembly Code

```
and $s3, $s1, $s2
or  $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

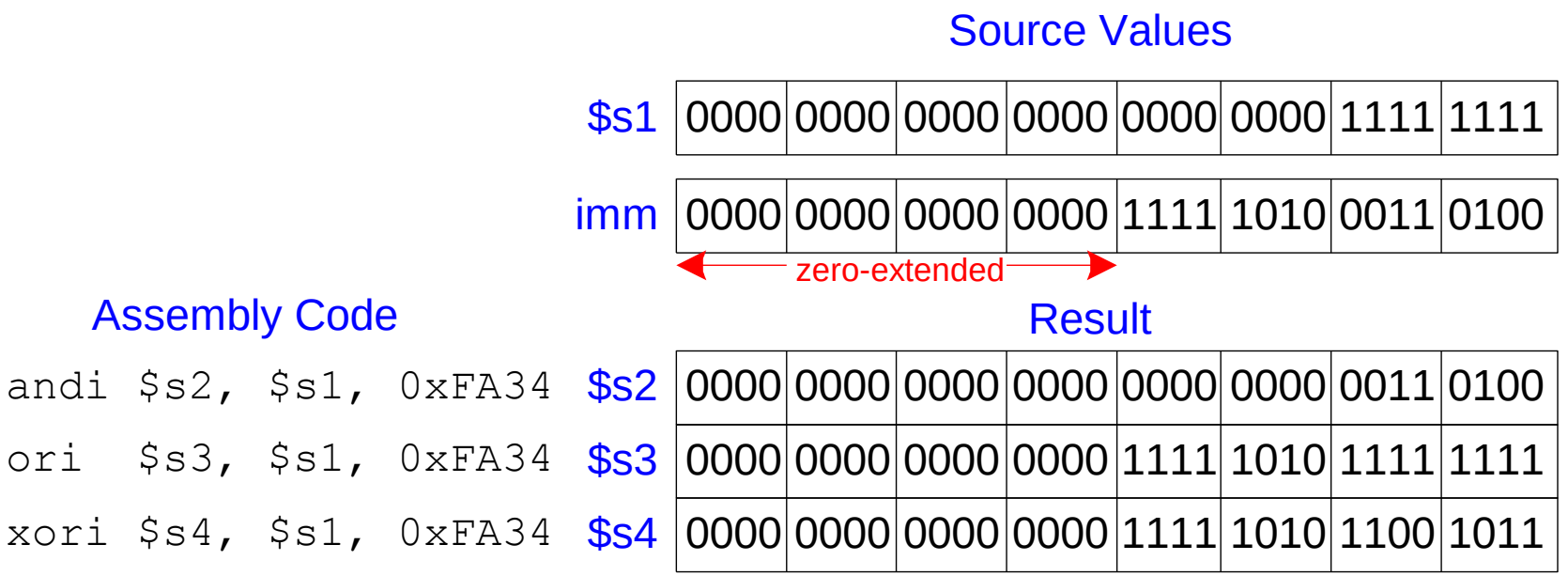
Result

$s3	0100	0110	1010	0001	0000	0000	0000	0000
$s4	1111	1111	1111	1111	1111	0000	1011	0111
$s5	1011	1001	0101	1110	1111	0000	1011	0111
$s6	0000	0000	0000	0000	0000	1111	0100	1000

Logical Instructions Example 2



Logical Instructions Example 2



Shift Instructions

- `sll`: shift left logical
  - **Example:** `sll $t0, $t1, 5  # $t0 <= $t1 << 5`
- `srl`: shift right logical
  - **Example:** `srl $t0, $t1, 5  # $t0 <= $t1 >> 5`
- `sra`: shift right arithmetic
  - **Example:** `sra $t0, $t1, 5  # $t0 <= $t1 >>> 5`

Variable Shift Instructions

- `sllv`: shift left logical variable
  - **Example:** `sllv $t0, $t1, $t2 # $t0 <= $t1 << $t2`
- `srlv`: shift right logical variable
  - **Example:** `srlv $t0, $t1, $t2 # $t0 <= $t1 >> $t2`
- `srav`: shift right arithmetic variable
  - **Example**: `srav $t0, $t1, $t2 # $t0 <= $t1 >>> $t2`

Shift Instructions

Assembly Code

```
sll $t0, $s1, 2
srl $s2, $s1, 2
sra $s3, $s1, 2
```

Field Values

	op	rs	rt	rd	shamt	funct
sll $t0, $s1, 2	0	0	17	8	2	0
srl $s2, $s1, 2	0	0	17	18	2	2
sra $s3, $s1, 2	0	0	17	19	2	3
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	00000	10001	01000	00010	000000	(0x00114080)
000000	00000	10001	10010	00010	000010	(0x00119082)
000000	00000	10001	10011	00010	000011	(0x00119883)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Generating Constants

- 16-bit constants using `addi`:

C Code

```
// int is a 32-bit signed word
int a = 0x4f3c;
```

MIPS assembly code

```
# $s0 = a
addi $s0, $0, 0x4f3c
```

- 32-bit constants using load upper immediate (`lui`) and `ori`:

C Code

```
int a = 0xFEDC8765;
```

MIPS assembly code

```
# $s0 = a
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
```

Multiplication, Division

- Special registers: `lo`, `hi`
- 32 × 32 multiplication, 64 bit result
  - `mult $s0, $s1`
  - Result in {`hi`, `lo`}
- 32-bit division, 32-bit quotient, remainder
  - `div $s0, $s1`
  - Quotient in `lo`
  - Remainder in `hi`
- Moves from `lo`/`hi` special registers
  - `mflo $s2`
  - `mfhi $s3`

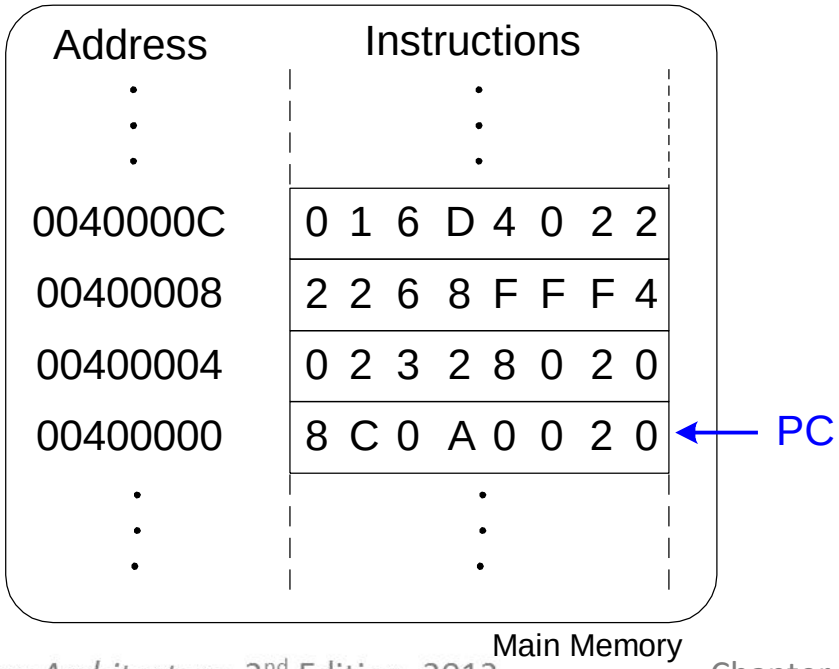
Branching

- Execute instructions out of sequence
- Types of branches:
  - **Conditional**
    - branch if equal (`beq`)
    - branch if not equal (`bne`)
  - **Unconditional**
    - jump (`j`)
    - jump register (`jr`)
    - jump and link (`jal`)

Review: The Stored Program

Assembly Code	Machine Code
`lw $t2, 32($0)`	`0x8C0A0020`
`add $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub $t0, $t3, $t5`	`0x016D4022`

Stored Program



Conditional Branching (beq)

```
# MIPS assembly
 addi  $s0, $0, 4          # $s0 = 0 + 4 = 4
 addi  $s1, $0, 1          # $s1 = 0 + 1 = 1
 sll   $s1, $s1, 2         # $s1 = 1 << 2 = 4
 beq   $s0, $s1, target    # branch is taken
 addi  $s1, $s1, 1         # not executed
 sub   $s1, $s1, $s0       # not executed

target:                    # label
 add   $s1, $s1, $s0       # $s1 = 4 + 4 = 8
```

Labels indicate instruction location. They can't be reserved words and must be followed by colon (:)

The Branch Not Taken (bne)

```
# MIPS assembly
  addi    $s0, $0, 4          # $s0 = 0 + 4 = 4
  addi    $s1, $0, 1          # $s1 = 0 + 1 = 1
  sll     $s1, $s1, 2         # $s1 = 1 << 2 = 4
  bne     $s0, $s1, target    # branch not taken
  addi    $s1, $s1, 1         # $s1 = 4 + 1 = 5
  sub     $s1, $s1, $s0       # $s1 = 5 - 4 = 1

target:
  add     $s1, $s1, $s0       # $s1 = 1 + 4 = 5
```

Unconditional Branching (j)

```
# MIPS assembly
  addi $s0, $0, 4              # $s0 = 4
  addi $s1, $0, 1              # $s1 = 1
  j        target              # jump to target
  sra      $s1, $s1, 2         # not executed
  addi     $s1, $s1, 1         # not executed
  sub      $s1, $s1, $s0       # not executed

  target:
  add      $s1, $s1, $s0       # $s1 = 1 + 4 = 5
```

Unconditional Branching (jr)

```
# MIPS assembly
0x00002000          addi $s0, $0, 0x2010
0x00002004          jr   $s0
0x00002008          addi $s1, $0, 1
0x0000200C          sra  $s1, $s1, 2
0x00002010          lw   $s3, 44($s1)
```

jr is an **R-type** instruction.

High-Level Code Constructs

- `if` statements
- `if/else` statements
- `while` loops
- `for` loops

If Statement

C Code

```
if (i == j)
   f = g + h;

f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
```

If Statement

C Code

```
if (i == j)
  f = g + h;

f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
    bne $s3, $s4, L1
    add $s0, $s1, $s2

L1: sub $s0, $s0, $s3
```

Assembly tests opposite case (`i != j`) of high-level code (`i == j`)

If/Else Statement

C Code

```
if (i == j)
  f = g + h;
else
  f = f - i;
```

MIPS assembly code

If/Else Statement

C Code

```
if (i == j)
  f = g + h;
else
  f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
      bne $s3, $s4, L1
      add $s0, $s1, $s2
      j   done
L1:   sub $s0, $s0, $s3
done:
```

While Loops

C Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

MIPS assembly code

Assembly tests for the opposite case (`pow == 128`) of the C code (`pow != 128`).

While Loops

C Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

MIPS assembly code

```
# $s0 = pow, $s1 = x

        addi $s0, $0, 1
        add  $s1, $0, $0
        addi $t0, $0, 128
while:  beq  $s0, $t0, done
        sll  $s0, $s0, 1
        addi $s1, $s1, 1
        j    while
done:
```

Assembly tests for the opposite case (`pow == 128`) of the C code (`pow != 128`).

For Loops

```
for (initialization; condition; loop operation)
  statement
```

- `initialization:` executes before the loop begins
- `condition:` is tested at the beginning of each iteration
- `loop operation:` executes at the end of each iteration
- `statement:` executes each time the condition is met

For Loops

High-level code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
  sum = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
```

For Loops

C Code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
  sum = sum + i;
}
```

MIPS assembly code

For Loops

C Code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
  sum = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
        addi $s1, $0, 0
        add  $s0, $0, $0
        addi $t0, $0, 10
for:    beq  $s0, $t0, done
        add  $s1, $s1, $s0
        addi $s0, $s0, 1
        j    for
done:
```

Less Than Comparison

C Code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
  sum = sum + i;
}
```

MIPS assembly code

Less Than Comparison

C Code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
  sum = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
        addi $s1, $0, 0
        addi $s0, $0, 1
        addi $t0, $0, 101
loop:   slt  $t1, $s0, $t0
        beq  $t1, $0, done
        add  $s1, $s1, $s0
        sll  $s0, $s0, 1
        j    loop
done:
```

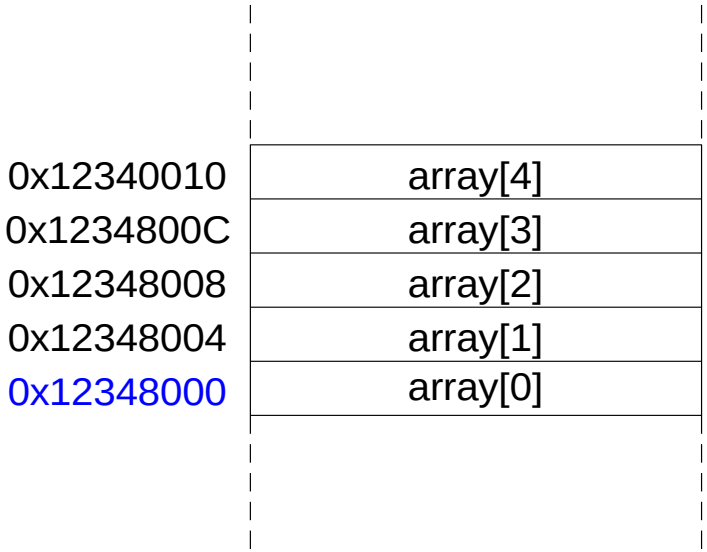
$t1 = 1 if i < 101

Arrays

- Access large amounts of similar data
- **Index**: access each element
- **Size**: number of elements

Arrays

- 5-element array
- **Base address** = 0x12348000 (address of first element, `array[0]`)
- First step in accessing an array: load base address into a register



Accessing Arrays

```
// C Code
   int array[5];
   array[0] = array[0] * 2;
   array[1] = array[1] * 2;
```

Accessing Arrays

```
// C Code
   int array[5];
   array[0] = array[0] * 2;
   array[1] = array[1] * 2;

# MIPS assembly code
# $s0 = array base address
  lui  $s0, 0x1234          # 0x1234 in upper half of $s0
  ori  $s0, $s0, 0x8000     # 0x8000 in lower half of $s0

  lw   $t1, 0($s0)          # $t1 = array[0]
  sll  $t1, $t1, 1          # $t1 = $t1 * 2
  sw   $t1, 0($s0)          # array[0] = $t1

  lw   $t1, 4($s0)          # $t1 = array[1]
  sll  $t1, $t1, 1          # $t1 = $t1 * 2
  sw   $t1, 4($s0)          # array[1] = $t1
```

Arrays using For Loops

```
// C Code
   int array[1000];
   int i;

   for (i=0; i < 1000; i = i + 1)
        array[i] = array[i] * 8;

# MIPS assembly code
# $s0 = array base address, $s1 = i
```

Arrays Using For Loops

```
# MIPS assembly code
# $s0 = array base address, $s1 = i
# initialization code
  lui  $s0, 0x23B8         # $s0 = 0x23B80000
  ori  $s0, $s0, 0xF000    # $s0 = 0x23B8F000
  addi $s1, $0, 0          # i = 0
  addi $t2, $0, 1000       # $t2 = 1000

loop:
  slt  $t0, $s1, $t2       # i < 1000?
  beq  $t0, $0, done       # if not then done
  sll  $t0, $s1, 2         # $t0 = i * 4 (byte offset)
  add  $t0, $t0, $s0       # address of array[i]
  lw   $t1, 0($t0)         # $t1 = array[i]
  sll  $t1, $t1, 3         # $t1 = array[i] * 8
  sw   $t1, 0($t0)         # array[i] = array[i] * 8
  addi $s1, $s1, 1         # i = i + 1
  j    loop                # repeat
done:
```

ASCII Code

- *American Standard Code for Information Interchange*
- Each text character has unique byte value
  - For example, S = 0x53, a = 0x61, A = 0x41
  - Lower-case and upper-case differ by 0x20 (32)

Cast of Characters

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

Function Calls

- **Caller:** calling function (in this case, `main`)
- **Callee:** called function (in this case, `sum`)

C Code

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

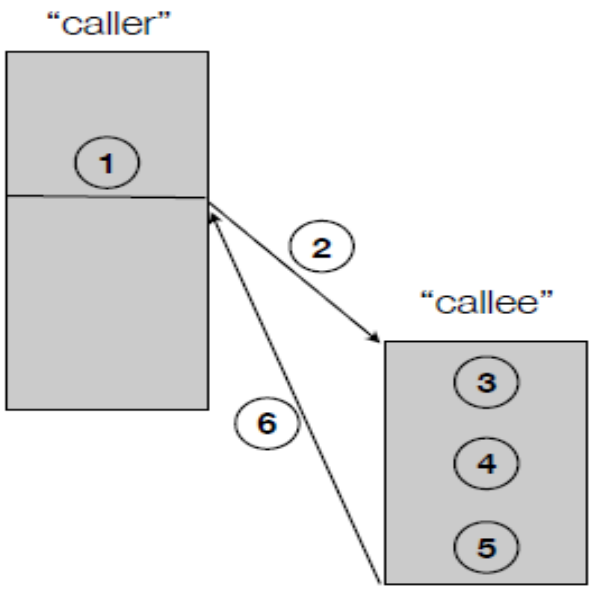
Function Conventions

- **Caller:**
  - passes **arguments** to callee
  - jumps to callee
- **Callee:**
  - **performs** the function
  - **returns** result to caller
  - **returns** to point of call
  - **must not overwrite** registers or memory needed by caller

Procedure Calling

- Steps required:

1. Place parameters in registers
2. Transfer control to procedure
3. Acquire storage for procedure
4. Perform procedure's operations
5. Place result in register for caller
6. Return to place of call



MIPS Function Conventions

- **Call Function:** jump and link (`jal`)
- **Return** from function: jump register (`jr`)
- **Arguments**: `$a0 – $a3`
- **Return value**: `$v0`

Function Calls

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

MIPS assembly code

```
0x00400200 main: jal  simple
0x00400204       add  $s0, $s1, $s2
...

0x00401020 simple: jr $ra
```

`void` means that `simple` doesn't return a value

Function Calls

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

MIPS assembly code

```
0x00400200 main: jal  simple
0x00400204       add  $s0, $s1, $s2
...

0x00401020 simple: jr $ra
```

jal: jumps to `simple`
`$ra` = PC + 4 = 0x00400204

jr $ra: jumps to address in `$ra` (0x00400204)

Input Arguments & Return Value

MIPS conventions:

- Argument values: `$a0` - `$a3`
- Return value: `$v0`

Input Arguments & Return Value

C Code

```
int main()
{
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);  // 4 arguments
  ...
}

int diffofsums(int f, int g, int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;                // return value
}
```

Input Arguments & Return Value

MIPS assembly code

```
# $s0 = y

main:
  ...
  addi $a0, $0, 2     # argument 0 = 2
  addi $a1, $0, 3     # argument 1 = 3
  addi $a2, $0, 4     # argument 2 = 4
  addi $a3, $0, 5     # argument 3 = 5
  jal  diffofsums     # call Function
  add  $s0, $v0, $0   # y = returned value
  ...

# $s0 = result
diffofsums:
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  jr  $ra             # return to caller
```

Input Arguments & Return Value

MIPS assembly code

```
# $s0 = result
diffofsums:
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  jr  $ra             # return to caller
```

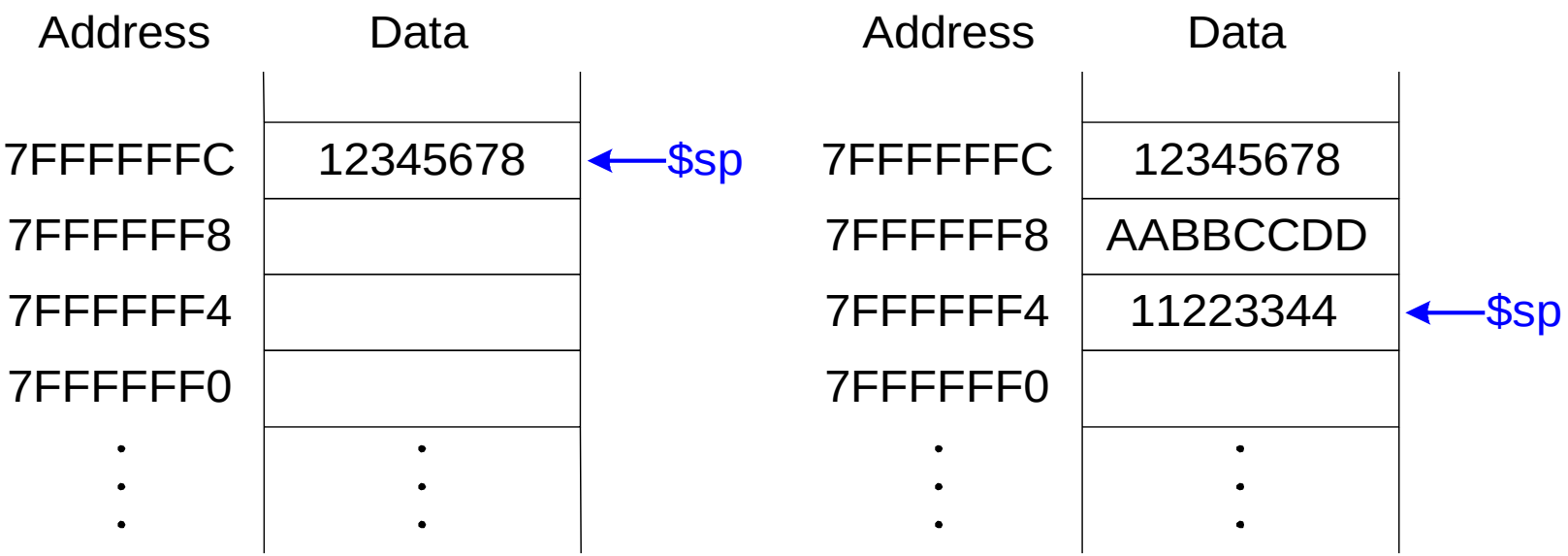
- `diffofsums` overwrote 3 registers: `$t0`, `$t1`, `$s0`
- `diffofsums` can use *stack* to temporarily store registers

The Stack

- Memory used to temporarily save variables
- Like stack of dishes, last-in-first-out (LIFO) queue
- ***Expands*:** uses more memory when more space needed
- ***Contracts*:** uses less memory when the space is no longer needed

The Stack

- Grows down (from higher to lower memory addresses)
- Stack pointer: `$sp` points to top of the stack



How Functions use the Stack

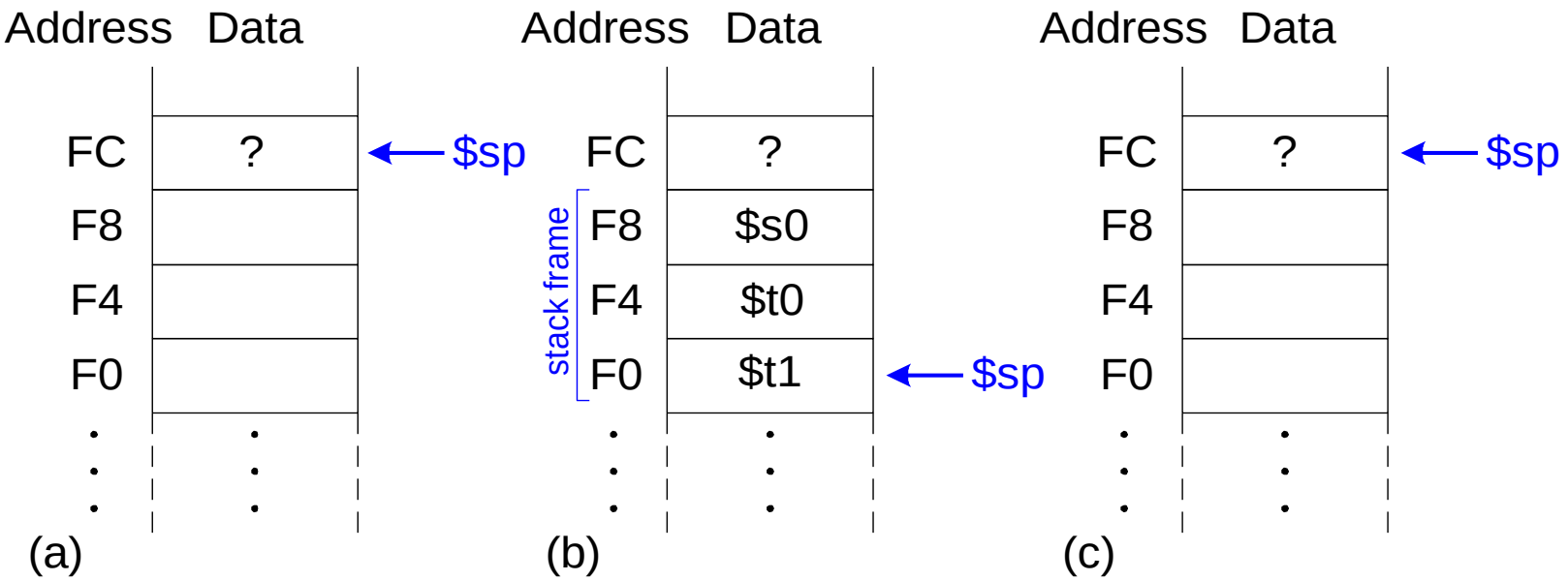
- Called functions must have no unintended side effects
- But `diffofsums` overwrites 3 registers: `$t0`, `$t1`, `$s0`

```
# MIPS assembly
# $s0 = result
diffofsums:
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  jr  $ra             # return to caller
```

Storing Register Values on the Stack

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -12   # make space on stack
                       # to store 3 registers
  sw   $s0, 8($sp)     # save $s0 on stack
  sw   $t0, 4($sp)     # save $t0 on stack
  sw   $t1, 0($sp)     # save $t1 on stack
  add  $t0, $a0, $a1   # $t0 = f + g
  add  $t1, $a2, $a3   # $t1 = h + i
  sub  $s0, $t0, $t1   # result = (f + g) - (h + i)
  add  $v0, $s0, $0    # put return value in $v0
  lw   $t1, 0($sp)     # restore $t1 from stack
  lw   $t0, 4($sp)     # restore $t0 from stack
  lw   $s0, 8($sp)     # restore $s0 from stack
  addi $sp, $sp, 12    # deallocate stack space
  jr   $ra             # return to caller
```

The stack during `diffofsums` Call



Registers

Preserved *Callee-Saved*	Nonpreserved *Caller-Saved*
`$s0-$s7`	`$t0-$t9`
`$ra`	`$a0-$a3`
`$sp`	`$v0-$v1`
stack above `$sp`	stack below `$sp`

Multiple Function Calls

```
proc1:
   addi $sp, $sp, -4     # make space on stack
   sw   $ra, 0($sp)      # save $ra on stack
   jal  proc2
   ...
   lw   $ra, 0($sp)      # restore $s0 from stack
   addi $sp, $sp, 4      # deallocate stack space
   jr  $ra               # return to caller
```

Storing Saved Registers on the Stack

```
# $s0 = result
diffofsums:
   addi $sp, $sp, -4   # make space on stack to
                       # store one register
   sw  $s0, 0($sp)     # save $s0 on stack
                       # no need to save $t0 or $t1
   add $t0, $a0, $a1   # $t0 = f + g
   add $t1, $a2, $a3   # $t1 = h + i
   sub $s0, $t0, $t1   # result = (f + g) - (h + i)
   add $v0, $s0, $0    # put return value in $v0
   lw  $s0, 0($sp)     # restore $s0 from stack
   addi $sp, $sp, 4    # deallocate stack space
   jr  $ra             # return to caller
```

Recursive Function Call

High-level code

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n-1));
}
```

Recursive Function Call

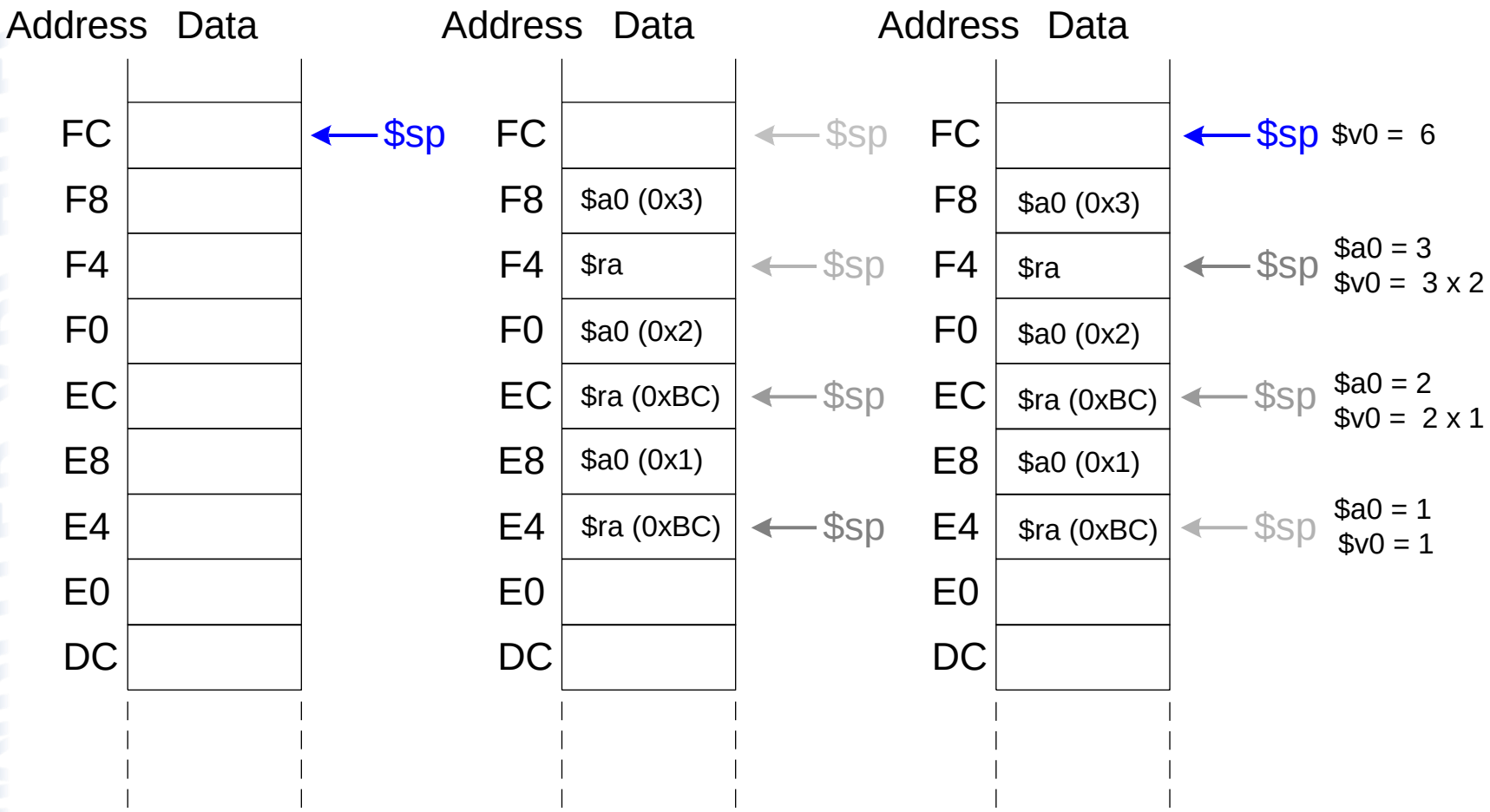
MIPS assembly code

Recursive Function Call

MIPS assembly code

```
0x90 factorial: addi $sp, $sp, -8   # make room
0x94            sw   $a0, 4($sp)    # store $a0
0x98            sw   $ra, 0($sp)    # store $ra
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0  # a <= 1 ?
0xA4            beq  $t0, $0, else  # no: go to else
0xA8            addi $v0, $0, 1     # yes: return 1
0xAC            addi $sp, $sp, 8    # restore $sp
0xB0            jr   $ra            # return
0xB4       else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial      # recursive call
0xBC            lw   $ra, 0($sp)    # restore $ra
0xC0            lw   $a0, 4($sp)    # restore $a0
0xC4            addi $sp, $sp, 8    # restore $sp
0xC8            mul  $v0, $a0, $v0  # n * factorial(n-1)
0xCC            jr   $ra            # return
```

Stack During Recursive Call



Function Call Summary

- **Caller**
  - Put arguments in `$a0-$a3`
  - Save any needed registers (`$ra`, maybe `$t0-t9`)
  - `jal callee`
  - Restore registers
  - Look for result in `$v0`
- **Callee**
  - Save registers that might be disturbed (`$s0-$s7`)
  - Perform function
  - Put result in `$v0`
  - Restore registers
  - `jr $ra`

Addressing Modes

How do we address the operands?

- Register Only
- Immediate
- Base Addressing
- PC-Relative
- Pseudo Direct

Addressing Modes

Register Only

- Operands found in registers
  - **Example:** `add $s0, $t2, $t3`
  - **Example:** `sub $t8, $s1, $0`

Immediate

- 16-bit immediate used as an operand
  - **Example:** `addi $s4, $t5, -73`
  - **Example:** `ori  $t3, $t7, 0xFF`

Addressing Modes

Base Addressing

- Address of operand is:

  `base address + sign-extended immediate`

  - **Example:** `lw $s4, 72($0)`
    - address = `$0 + 72`
  - **Example:** `sw $t2, -25($t1)`
    - address = `$t1 - 25`

Addressing Modes

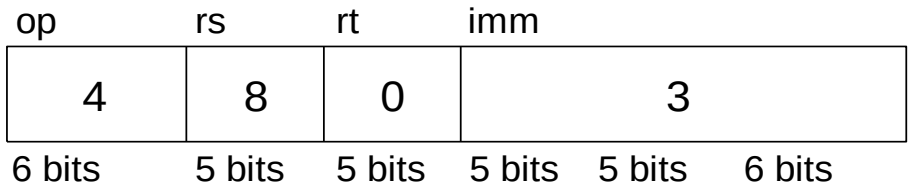
PC-Relative Addressing

```
0x10              beq   $t0, $0, else
0x14              addi  $v0, $0, 1
0x18              addi  $sp, $sp, i
0x1C              jr          $ra
0x20      else:   addi  $a0, $a0, -1
0x24              jal   factorial
```

Assembly Code

```
beq $t0, $0, else
(beq $t0, $0, 3)
```

Field Values



Addressing Modes

Pseudo-direct Addressing

```
0x0040005C          jal   sum
...
0x004000A0  sum:    add   $v0, $a0, $a1
```

JTA 0000 0000 0100 0000 0000 0000 1010 0000 (0x004000A0)

26-bit addr 0000 0000 0100 0000 0000 0000 1010 0000 (0x0100028)

0 1 0 0 0 2 8

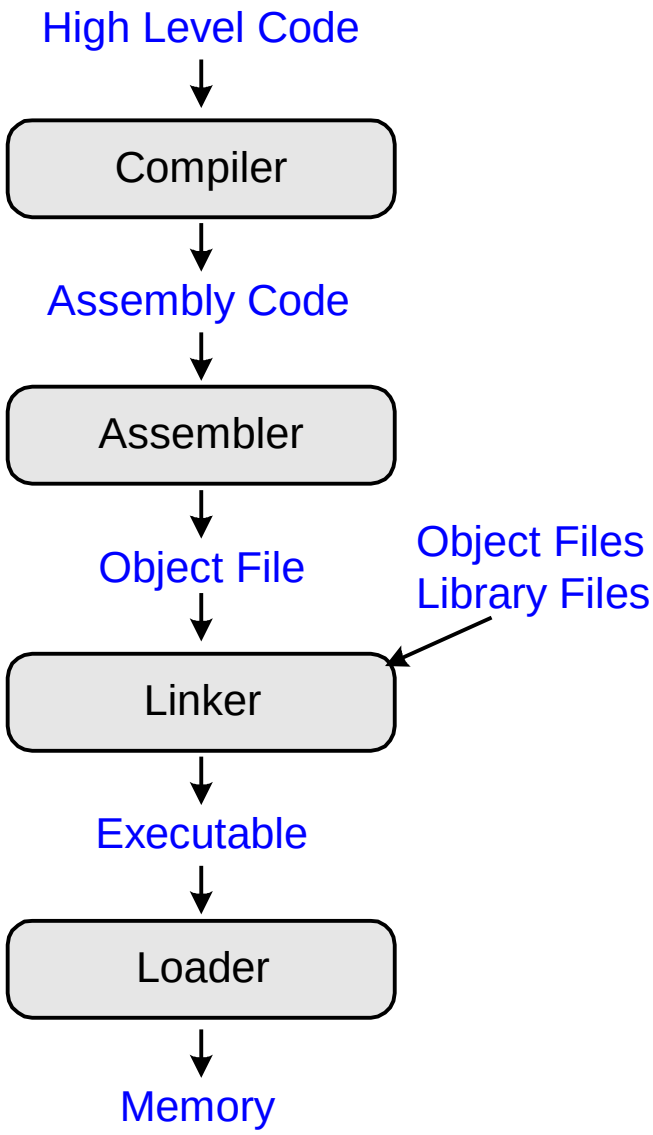
Field Values

op	imm
3	0x0100028
6 bits	26 bits

Machine Code

op	addr	
000011	00 0001 0000 0000 0000 0010 1000	(0x0C100028)
6 bits	26 bits	

How to Compile & Run a Program



Grace Hopper, 1906-1992

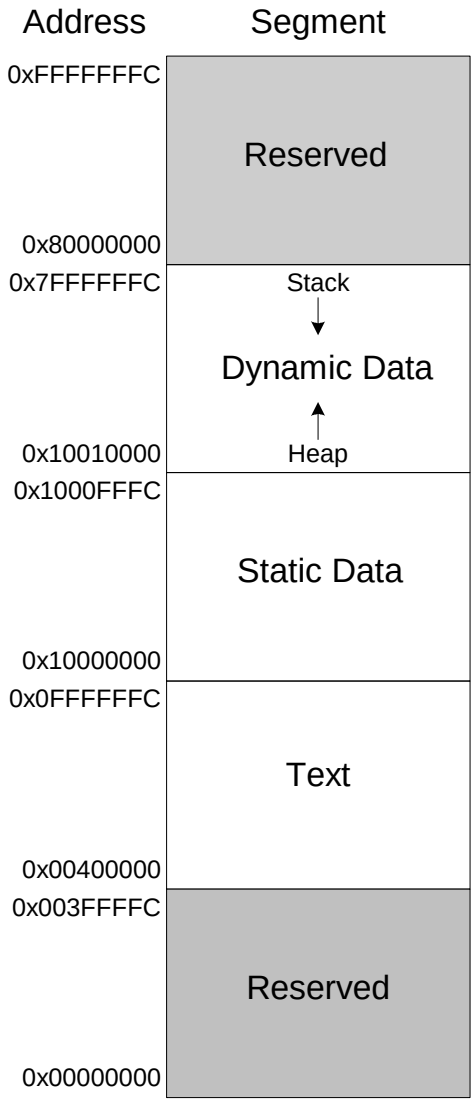
- Graduated from Yale University with a Ph.D. in mathematics
- Developed first compiler
- Helped develop the COBOL programming language
- Highly awarded naval officer
- Received World War II Victory Medal and National Defense Service Medal, among others

What is Stored in Memory?

- Instructions (also called *text*)
- Data
  - Global/static: allocated before program begins
  - Dynamic: allocated within program

- How big is memory?
  - At most 2^{32} = 4 gigabytes (4 GB)
  - From address 0x00000000 to 0xFFFFFFFF

MIPS Memory Map



Example Program: C Code

```
int f, g, y;  // global variables


int main(void)
{
  f = 2;
  g = 3;
  y = sum(f, g);

  return y;
}


int sum(int a, int b) {
  return (a + b);
}
```

Example Program: MIPS Assembly

```
int f, g, y;  // global


int main(void)
{


  f = 2;
  g = 3;

  y = sum(f, g);
  return y;
}

int sum(int a, int b) {
  return (a + b);
}
```

```
.data
f:
g:
y:
.text
main:
  addi $sp, $sp, -4    # stack frame
  sw   $ra, 0($sp)     # store $ra
  addi $a0, $0, 2      # $a0 = 2
  sw   $a0, f          # f = 2
  addi $a1, $0, 3      # $a1 = 3
  sw   $a1, g          # g = 3
  jal  sum             # call sum
  sw   $v0, y          # y = sum()
  lw   $ra, 0($sp)     # restore $ra
  addi $sp, $sp, 4     # restore $sp
  jr   $ra             # return to OS
sum:
  add  $v0, $a0, $a1   # $v0 = a + b
  jr   $ra             # return
```

Example Program: Symbol Table

Symbol	Address

Example Program: Symbol Table

Symbol	Address
`f`	**0x10000000**
`g`	**0x10000004**
`y`	**0x10000008**
`main`	**0x00400000**
`sum`	**0x0040002C**

Example Program: Executable

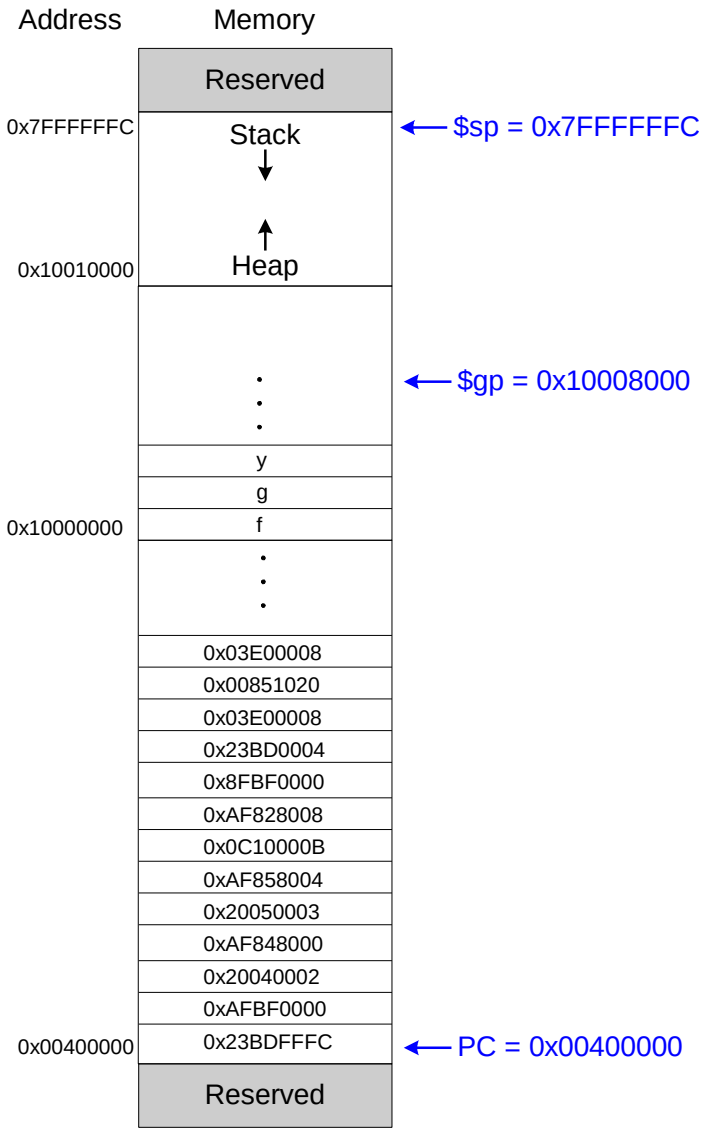
Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)

Text segment	Address	Instruction
	0x00400000	0x23BDFFFC
	0x00400004	0xAFBF0000
	0x00400008	0x20040002
	0x0040000C	0xAF848000
	0x00400010	0x20050003
	0x00400014	0xAF858004
	0x00400018	0x0C10000B
	0x0040001C	0xAF828008
	0x00400020	0x8FBF0000
	0x00400024	0x23BD0004
	0x00400028	0x03E00008
	0x0040002C	0x00851020
	0x00400030	0x03E00008

Data segment	Address	Data
	0x10000000	f
	0x10000004	g
	0x10000008	y

```
addi $sp, $sp, -4
sw   $ra, 0 ($sp)
addi $a0, $0, 2
sw   $a0, 0x8000 ($gp)
addi $a1, $0, 3
sw   $a1, 0x8004 ($gp)
jal  0x0040002C
sw   $v0, 0x8008 ($gp)
lw   $ra, 0 ($sp)
addi $sp, $sp, -4
jr   $ra
add  $v0, $a0, $a1
jr   $ra
```

Example Program: In Memory



Odds & Ends

- Pseudoinstructions
- Exceptions
- Signed and unsigned instructions
- Floating-point instructions

Pseudoinstructions

Pseudoinstruction	MIPS Instructions
`li $s0, 0x1234AA77`	`lui $s0, 0x1234` `ori $s0, 0xAA77`
`clear $t0`	`add $t0, $0, $0`
`move $s1, $s2`	`add $s2, $s1, $0`
`nop`	`sll $0, $0, 0`

Exceptions

- Unscheduled function call to *exception handler*
- Caused by:
  - Hardware, also called an *interrupt*, e.g., keyboard
  - Software, also called *traps*, e.g., undefined instruction
- When exception occurs, the processor:
  - Records the cause of the exception
  - Jumps to exception handler (at instruction address 0x80000180)
  - Returns to program

Exception Registers

- Not part of register file
  - **`Cause`**: Records cause of exception
  - **`EPC`** (Exception PC): Records PC where exception occurred
- `EPC` and `Cause`: part of Coprocessor 0
- Move from Coprocessor 0
  - `mfc0 $k0, EPC`
  - Moves contents of `EPC` into `$k0`

Exception Causes

Exception	Cause
Hardware Interrupt	0x00000000
System Call	0x00000020
Breakpoint / Divide by 0	0x00000024
Undefined Instruction	0x00000028
Arithmetic Overflow	0x00000030

Exception Flow

- Processor saves cause and exception PC in `Cause` and `EPC`
- Processor jumps to exception handler (0x80000180)
- Exception handler:
  - Saves registers on stack
  - Reads `Cause` register

    ```
    mfc0 $k0, Cause
    ```
  - Handles exception
  - Restores registers
  - Returns to program

    ```
    mfc0 $k0, EPC
    jr $k0
    ```

Signed & Unsigned Instructions

- Addition and subtraction
- Multiplication and division
- Set less than

Addition & Subtraction

- **Signed:** `add`, `addi`, `sub`
  - Same operation as unsigned versions
  - But processor takes exception on overflow
- **Unsigned:** `addu`, `addiu`, `subu`
  - Doesn't take exception on overflow

Note: `addiu` sign-extends the immediate

Multiplication & Division

- **Signed:** `mult`, `div`
- **Unsigned:** `multu`, `divu`

Set Less Than

- **Signed:** `slt`, `slti`
- **Unsigned:** `sltu`, `sltiu`

Note: `sltiu` sign-extends the immediate before comparing it to the register

Loads

- **Signed:**
  - Sign-extends to create 32-bit value to load into register
  - Load halfword: `lh`
  - Load byte: `lb`
- **Unsigned:**
  - Zero-extends to create 32-bit value
  - Load halfword unsigned: `lhu`
  - Load byte: `lbu`

Floating-Point Instructions

- Floating-point coprocessor (Coprocessor 1)
- 32 32-bit floating-point registers (`$f0-$f31`)
- Double-precision values held in two floating point registers
  - e.g., `$f0` and `$f1`, `$f2` and `$f3`, etc.
  - Double-precision floating point registers: `$f0`, `$f2`, `$f4`, etc.

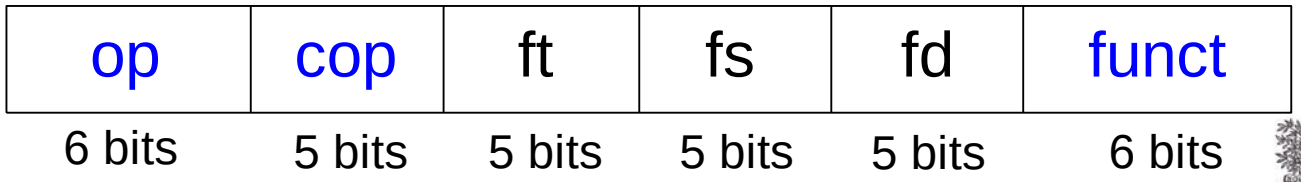
Floating-Point Instructions

Name	Register Number	Usage
`$fv0 - $fv1`	0, 2	return values
`$ft0 - $ft3`	4, 6, 8, 10	temporary variables
`$fa0 - $fa1`	12, 14	Function arguments
`$ft4 - $ft8`	16, 18	temporary variables
`$fs0 - $fs5`	20, 22, 24, 26, 28, 30	saved variables

F-Type Instruction Format

- `Opcode` = 17 (010001_2)
- Single-precision:
  - `cop` = 16 (010000_2)
  - `add.s`, `sub.s`, `div.s`, `neg.s`, `abs.s`, etc.
- Double-precision:
  - `cop` = 17 (010001_2)
  - `add.d`, `sub.d`, `div.d`, `neg.d`, `abs.d`, etc.
- 3 register operands:
  - `fs`, `ft`: source operands
  - `fd`: destination operands

F-Type



Floating-Point Branches

- Set/clear condition flag: `fpcond`
  - Equality: `c.seq.s`, `c.seq.d`
  - Less than: `c.lt.s`, `c.lt.d`
  - Less than or equal: `c.le.s`, `c.le.d`
- Conditional branch
  - `bclf`: branches if `fpcond` is FALSE
  - `bclt`: branches if `fpcond` is TRUE
- Loads and stores
  - `lwc1`: `lwc1 $ft1, 42($s1)`
  - `swc1`: `swc1 $fs2, 17($sp)`

Looking Ahead

Microarchitecture – building MIPS processor in hardware

Bring colored pencils