COMP4611: Design and Analysis of Computer Architectures

Instruction Set Architectures

RISC, CISC, and MIPS

CISC (Complex Instruction Set Computers)

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: Pentium III added another 70 instructions (SSE)
- 2001: Another 144 instructions (SSE2)
- 2003: AMD extends the architecture to expand 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”

The Rationale for CISC

- One of the most visible forms of evolution associated with computers is that of programming languages
- As the cost of hardware has dropped, the relative cost of software has risen.
- Complexity of modern software has increased the prevalence of faults (bugs).
- Thus, the major cost in the lifecycle of a system is software, not hardware.

The Rationale for CISC

The response from researchers and industry has been to develop ever more powerful and complex high-level languages.

- The high-level languages (HLL) allow the programmer to express algorithms more concisely, take care of much of the detail, and naturally support structured programming and object-oriented design.
- This solution gave rise to another problem, known as the ***semantic gap***. This is the difference between the operations provided in HLLs and those provided in computer architecture ISA.

The Rationale for CISC

- Symptoms of this gap include:
  - Execution inefficiency
  - Excessive program size
  - Compiler complexity
- Designers responded with architectures intended to close this gap. Key feature include:
  - Large instruction sets
  - Dozens of addressing modes
  - Various HLL statements implemented in hardware.

The Rationale for CISC

Such complex instruction sets are intended to:

- Ease the task of the compiler writer
- Improve execution efficiency, because complex sequences of operations can be implemented in microcode
- Provide support for even more complex and sophisticated HLLs

RISC (Reduced Instruction Set Computers)

The Rationale for RISC

- A number of studies have been done to determine the characteristics and patterns of execution of machine instructions generated from HLL programs.
- The results of these studies inspired some researchers to look for a different approach.
- Namely, to make the architecture that supports the HLL simpler, rather than more complex.

RISC

- RISC systems have been defined and designed in a variety of ways, the key elements shared by most designs are:
  - A limited and simple instruction set.
  - A large number of general-purpose registers, and the use of compiler technology to optimize register usage.
  - An emphasis on optimizing the instruction pipeline.

Characteristics of RISC Architectures

Although there are a variety of approaches to RISC architectures, certain characteristics are typical to RISC architectures, particularly early systems:

- **One instruction per cycle** – RISC machine instructions comprise only one cycle of fetch, execute, store. With simple, one-cycle instructions, there is no need for microcode (as in CISC); machine instructions can be hardwired. Such instructions should execute faster than comparable machine instructions on CISC machines, as it is not necessary to access a micro-program control store.

- **Register-to-register operation** – If most register operations are register-to-register, the instruction set and therefore the control unit are simplified. For example, a RISC instruction set may only include one or two ADD instructions; VAX has 25 different ADD instructions. This also encourages the optimization of register use.

Characteristics of RISC Architectures

- **Simple addressing modes** – Almost all RISC instructions use simple register addressing. Complex addressing modes can be synthesized in software from simple ones. Again, this design feature simplifies the instruction set and the control unit.

- **Simple instruction formats** - Generally, only one or a few formats are used. Instruction length is fixed and aligned on word boundaries. Field locations, especially the opcode, are fixed. This has a number of benefits:
  - With fixed fields, opcode decoding and register operand accessing can occur simultaneously.
  - Simplified formats simplify the control unit.
  - Instruction fetching is optimized because word-length units are fetched.
  - Alignment on word boundary also means that a single instruction does not cross page boundaries.

Prospective Benefits of RISC

The benefits of RISC fall into the following two main categories

Performance

- More effective compiler optimization: With more primitive instructions, there are more opportunities for moving operations out of loops, reorganizing code, maximizing register utilization, etc.
- Simple instructions (and little or no microcode) permit a relatively simple control unit, which is likely to be faster than a more complex one.
- Instruction pipelining. RISC researchers feel that the instruction pipelining technique can be applied much more effectively with a reduced instruction set.

VLSI Implementation

- Chip real estate: an early CISC processor typically devotes about half of its area to the control unit. A RISC processor typically uses only about 10% of the area for the control unit, using precious real estate for registers instead.
- Design and implementation time. The simple control unit and circuitry of RISC result in faster design cycles.

CISC vs. RISC Characteristics

- After the initial enthusiasm for RISC, there has been a growing realization that RISC designs may benefits from the inclusion of some CISC features, and vice-versa.
- The result is that more recent RISC design, PowerPC and SPARC, are no longer "pure" RISC and the more recent CISC designs, notably the Pentium and Core Duo incorporate core RISC characteristics internally.
  - Recent Intel processors implement an internal instruction set that shares similarity with RISC and support x86 (CISC) instruction set externally.
  - Hardware translates x86 instructions into the internal instruction set

CRISC—Complex-Reduced Instruction Set Computer?

Example CISC ISA:
Intel X86,386/486/Pentium

12 addressing modes:

- Register.
- Immediate.
- Direct.
- Base.
- Base + Displacement.
- Index + Displacement.
- Scaled Index + Displacement.
- Based Index.
- Based Scaled Index.
- Based Index + Displacement.
- Based Scaled Index + Displacement.
- Relative.

Operand sizes:

- Can be 8, 16, 32, 48, 64, or 80 bits long.
- Also supports string operations.

Instruction Encoding:

- Variable-size instructions
- The first bytes generally contain the opcode, mode specifiers, and register fields.
- The remainder bytes are for address displacement and immediate data.

Example RISC ISA: PowerPC

8 addressing modes:

- Register direct.
- Immediate.
- Register indirect.
- Register indirect with immediate index (loads and stores).
- Register indirect with register index (loads and stores).
- Absolute (jumps).
- Link register indirect (calls).
- Count register indirect (branches).

Operand sizes:

- Four operand sizes: 1, 2, 4 or 8 bytes.

Instruction Encoding:

- Instruction set has 15 different formats with many minor variations.
- All are 32 bits in length.

Example RISC ISA:

HP Precision Architecture, HP-PA

7 addressing modes:

- Register
- Immediate
- Base with displacement
- Base with scaled index and displacement
- Predecrement
- Postincrement
- PC-relative

Operand sizes:

- Five operand sizes ranging in powers of two from 1 to 16 bytes.

Instruction Encoding:

- Instruction set has 12 different formats.
- All are 32 bits in length.

Example RISC ISA: SPARC

5 addressing modes:

- Register indirect with immediate displacement.
- Register indirect indexed by another register.
- Register direct.
- Immediate.
- PC relative.

Operand sizes:

- Four operand sizes: 1, 2, 4 or 8 bytes.

Instruction Encoding:

- Instruction set has 3 basic instruction formats with 3 minor variations.
- All are 32 bits in length.

Example RISC ISA: Compaq Alpha AXP

4 addressing modes:

- Register direct.
- Immediate.
- Register indirect with displacement.
- PC-relative.

Operand sizes:

- Four operand sizes: 1, 2, 4 or 8 bytes.

Instruction Encoding:

- Instruction set has 7 different formats.
- All are 32 bits in length.

Architectural Evolution Macro-Level

Common Instruction Set Architecture

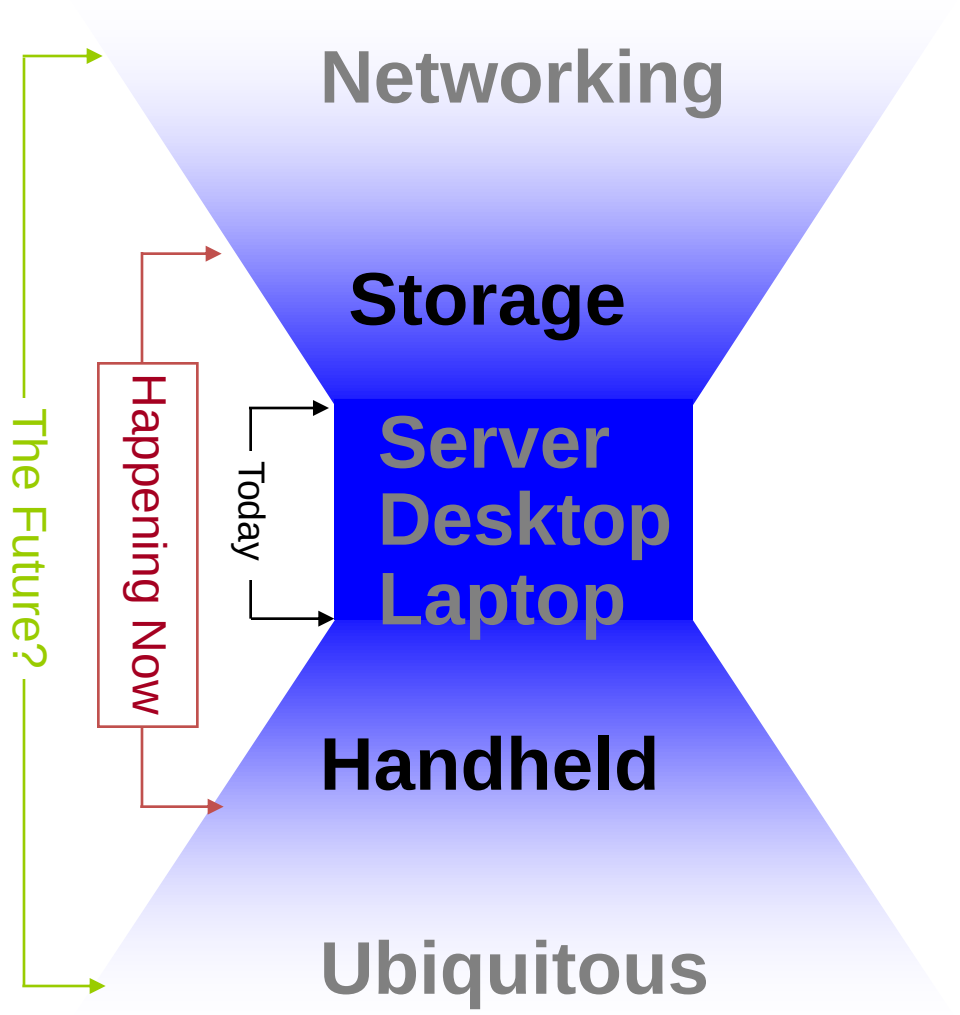
No need to port

No need for multiple validations

Built in OS integration

Robust security

Investment protection

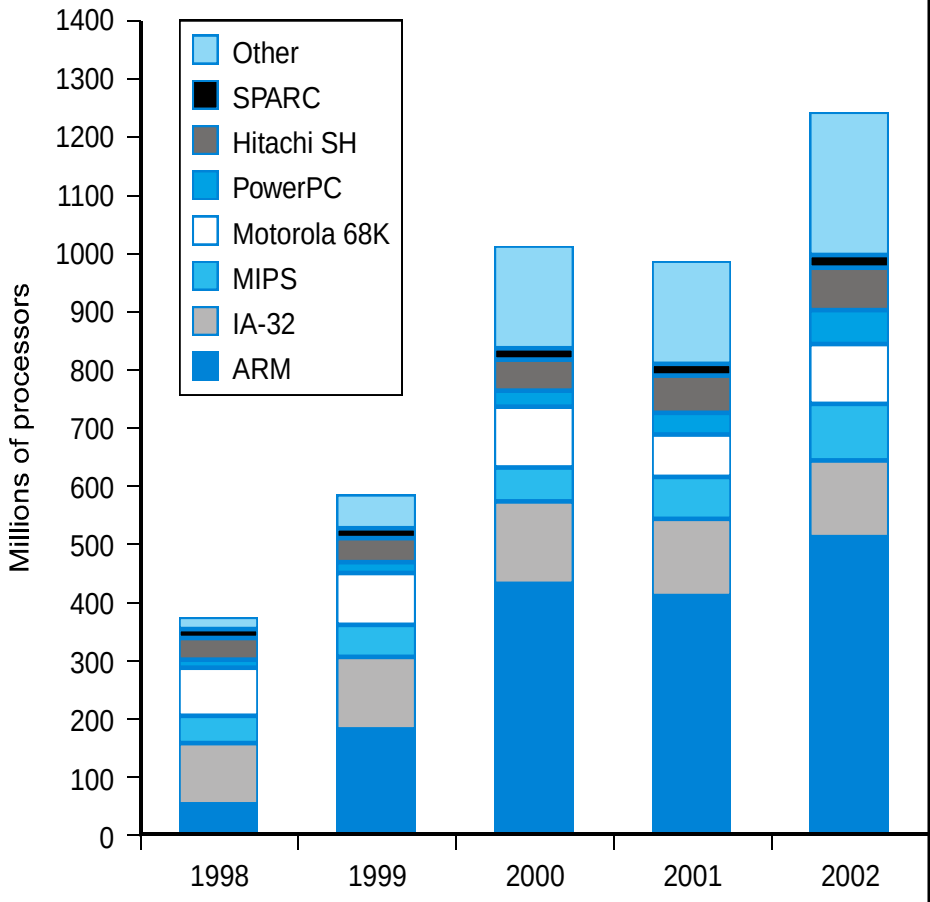


MIPS: Case Study of Instruction Set Architecture

MIPS

- MIPS: Microprocessor without Interlocked Pipeline Stages

- The MIPS instruction set architecture
  - is similar to other RISC architectures (SPARC)
  - has almost 200 million MIPS processors manufactured in 2006
  - is used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, networking equipment, …



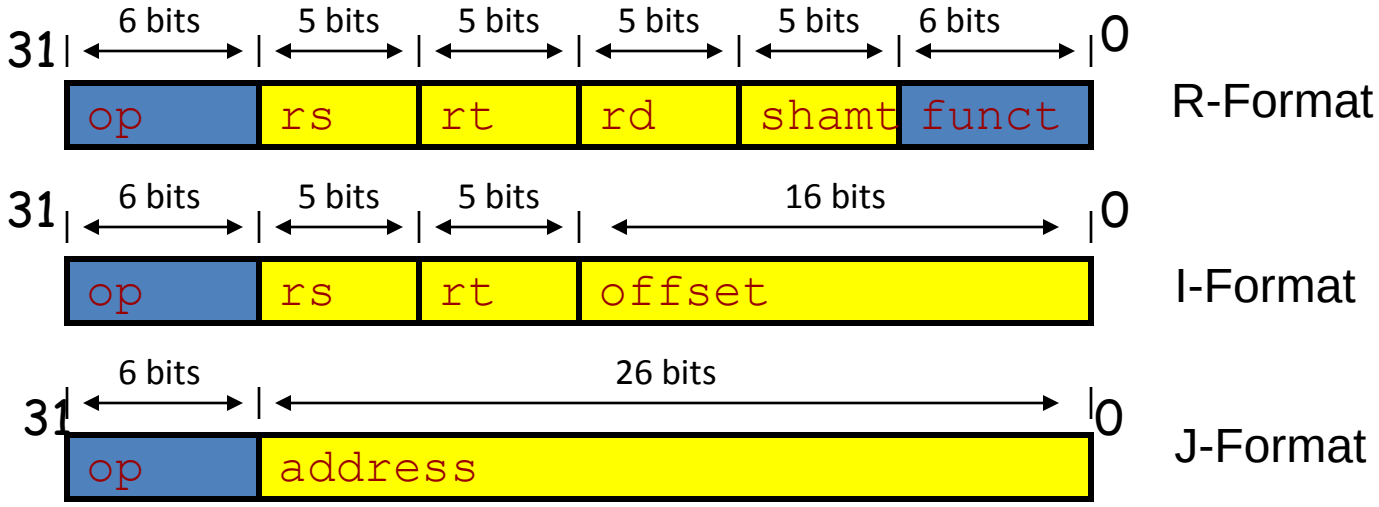
MIPS Design Principles

1. Simplicity Favors Regularity
   - Keep all instructions a single size
   - Always require three register operands in arithmetic instructions
2. Smaller is Faster
   - Has only 32 registers rather than many more
3. Good Design Makes Good Compromises
   - Comprise between providing larger addresses and constants in the instruction and keeping instructions the same length
4. Make the Common Case Fast
   - PC-relative addressing for conditional branches
   - Immediate addressing for constant operands

MIPS Instruction Set (RISC)

- Instructions perform 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.

MIPS Instructions



- All instructions exactly 32 bits wide
- Different formats for different purposes
- Similarities in formats ease implementation

MIPS Instruction Types

- Arithmetic & Logical - manipulate data in registers

```
add $s1, $s2, $s3        $s1 = $s2 + $s3
or $s3, $s4, $s5         $s3 = $s4 OR $s5
```

- Data Transfer - move register data to/from memory

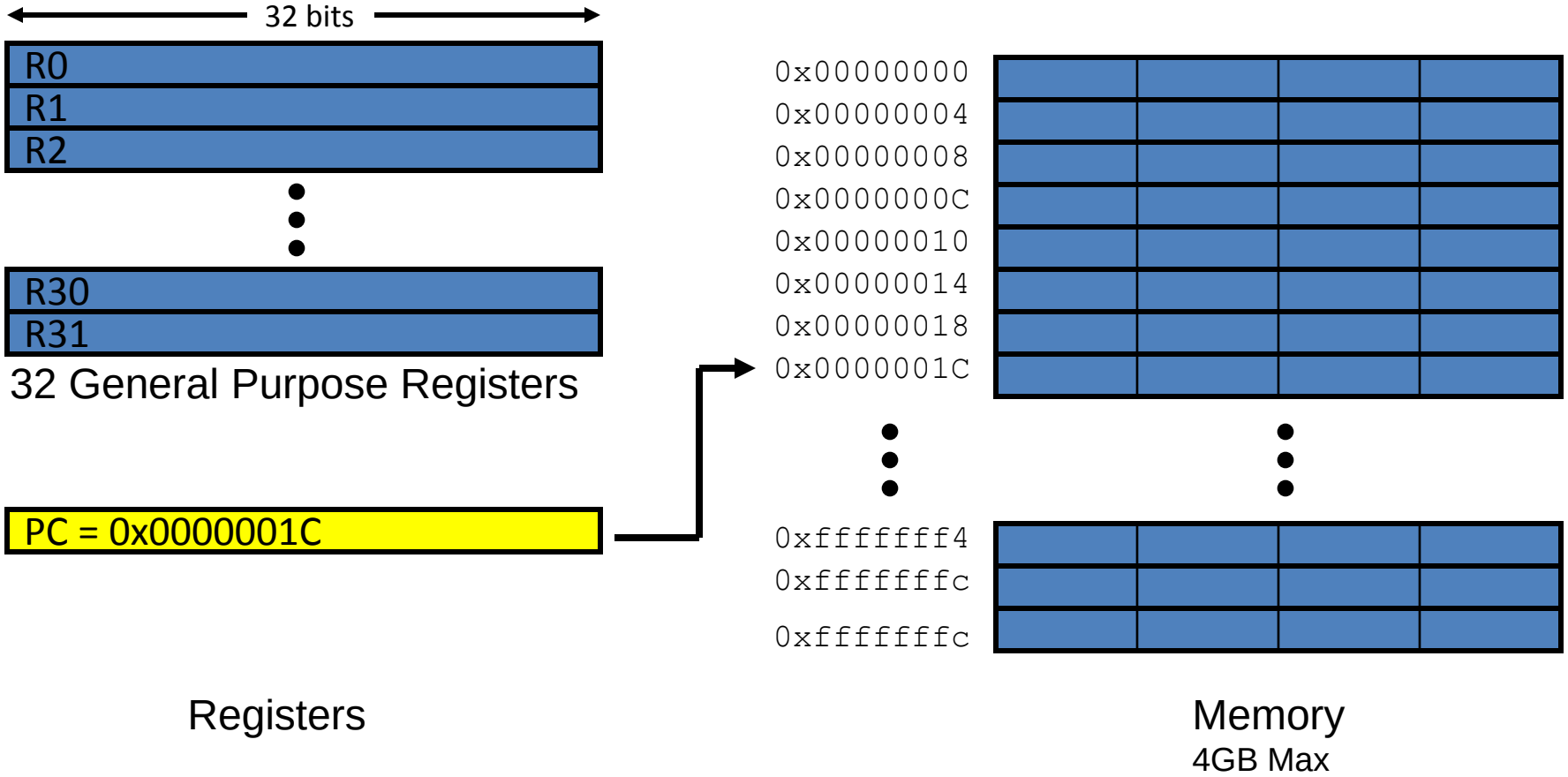
```
lw $s1, 100($s2)         $s1 = Memory[$s2 + 100]
sw $s1, 100($s2)         Memory[$s2 + 100] = $s1
```

- Branch - alter program flow

```
beq $s1, $s2, 25         if ($s1==$s2) PC = PC + 4 + 4*25
```

MIPS Registers and Memory

– 2^{32} bytes with addresses 0, 1, 2, …, 2^{32}-1



MIPS Registers and Usage

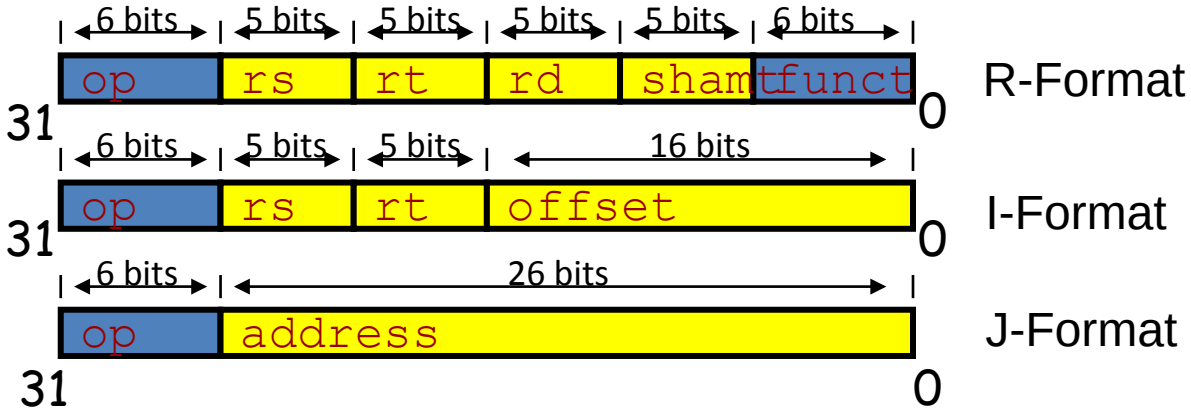
Name	Register number	Usage
$zero	0	the constant value 0
$at	1	reserved for assembler
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporary registers
$s0-$s7	16-23	saved registers
$t8-$t9	24-25	more temporary registers
$k0-$k1	26-27	reserved for Operating System kernel
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

Each register can be referred to by number or name.

Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - Example: `add $t1, $s1, $s2`
  - registers have numbers, `$t1=9, $s1=17, $s2=18`
- Instruction Format:

```
000000 10001 10010 01001 00000 100000
  op     rs    rt    rd   shamt funct
```



MIPS Data Transfer Instructions

- Transfer data between registers and memory
- Instruction format (assembly)

```
lw $dest, offset($addr)    load word
sw $src, offset($addr)     store word
```

- Uses:
  - Accessing a variable in main memory
  - Accessing an array element

Memory Instructions

- Load and store instructions
- Example:

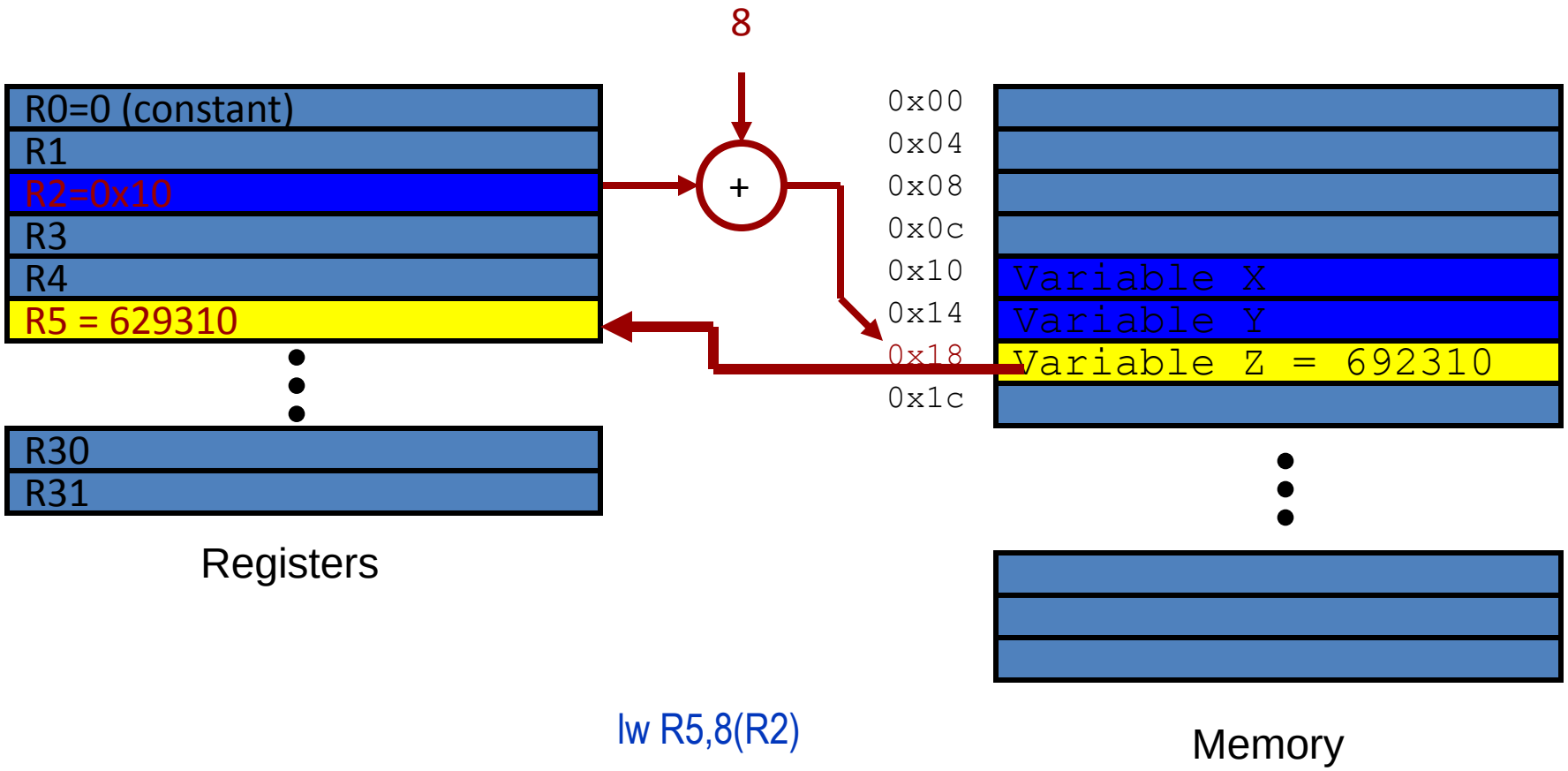
C code:
```
A[12] = h + A[8];
```

MIPS code:
```
lw $t0, 32($s3)    #addr of A in reg s3
add $t0, $s2, $t0 #h in reg s2
sw $t0, 48($s3)
```

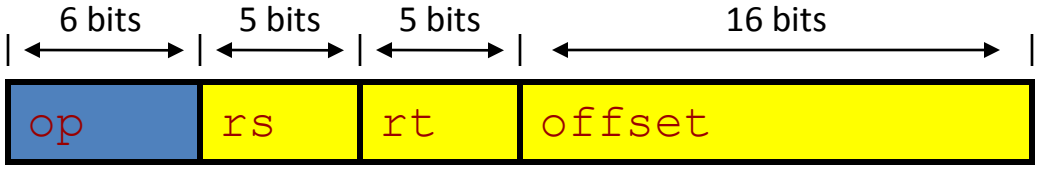
- Can refer to registers by name (e.g., $s2, $t2) instead of number
- Store word has destination last
- Remember arithmetic operands are registers, not memory!

Can't write:
```
add 48($s3), $s2, 32($s3)
```

Example - Loading a Simple Variable



Data Transfer Instructions - Binary Representation



- Used for load, store instructions
  - `op`: Basic operation of the instruction (*opcode*)
  - `rs`: first register source operand
  - `rt`: second register source operand
  - `offset`: 16-bit signed address offset (-32,768 to +32,767)
- Also called "I-Format" or "I-Type" instructions

MIPS Arithmetic Instructions

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

Example:

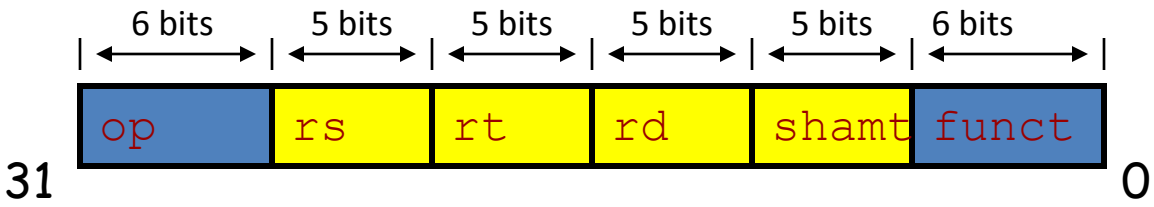
C code: `a = b + c`

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

(a, b, c are the corresponding registers)

"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"

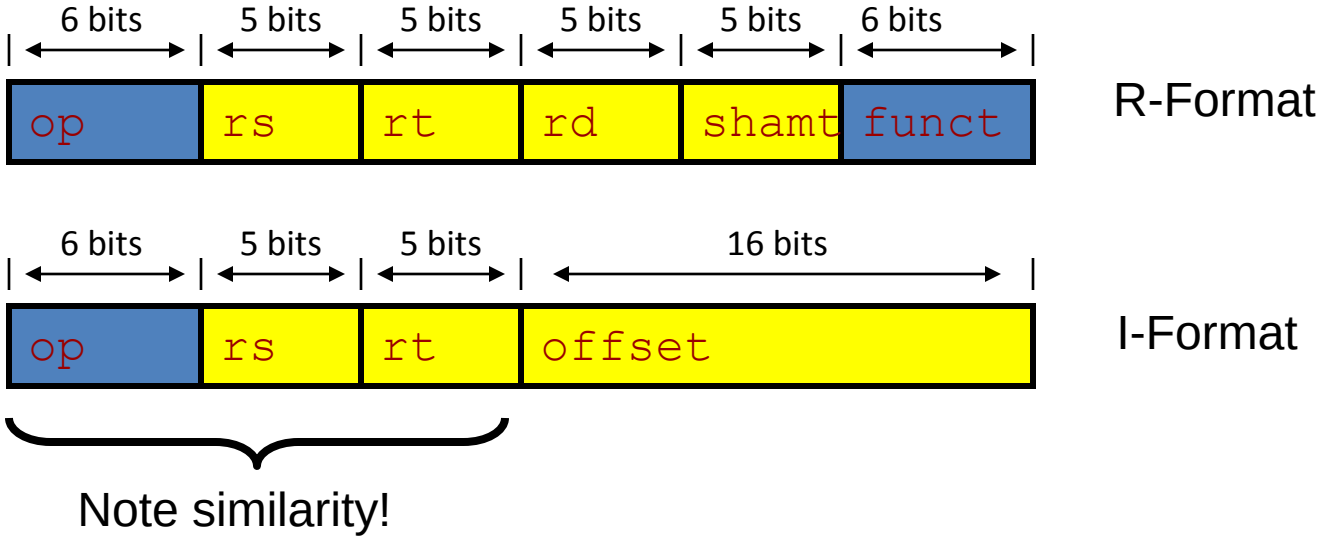
Arithmetic & Logical Instructions - Binary Representation



- Used for arithmetic, logical, shift instructions
  - `op`: Basic operation of the instruction (*opcode*)
  - `rs`: first register source operand
  - `rt`: second register source operand
  - `rd`: register destination operand
  - `shamt`: shift amount (more about this later)
  - `funct`: function - specific type of operation
- Also called "R-Format" or "R-Type" Instructions

I-Format vs. R-Format Instructions

- Compare with R-Format



MIPS Conditional Branch Instructions

- Conditional branches allow decision making

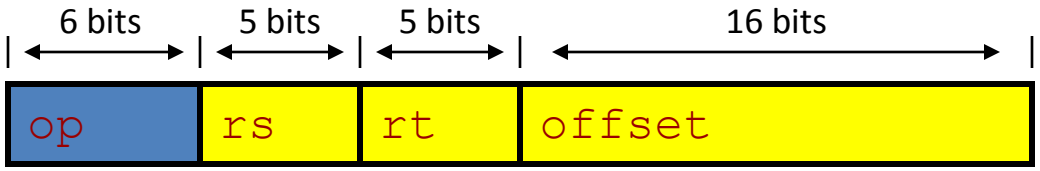
```
beq R1, R2, LABEL      if R1==R2 goto LABEL
bne R3, R4, LABEL      if R3!=R4 goto LABEL
```

Example

```
C Code      if (i==j) goto L1;
            f = g + h;
      L1:   f = f - i;

Assembly    beq $s3, $s4, L1
            add $s0, $s1, $s2
      L1:   sub $s0, $s0, $s3
```

Binary Representation - Branch



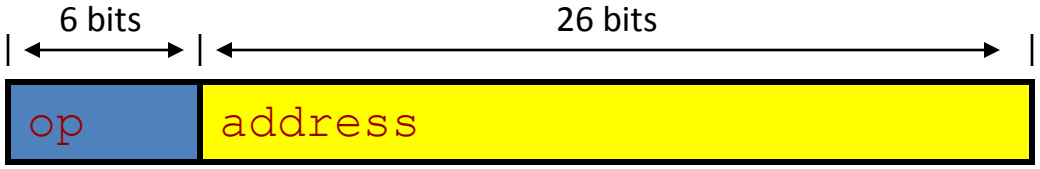
- Branch instructions use I-Format
- `offset` is added to PC when branch is taken

beq r0, r1, offset

has the effect:

if (r0==r1) pc = pc + 4 + offset
else pc = pc + 4;

Binary Representation - Jump



- Jump Instruction uses J-Format (op=2)
- What happens during execution?

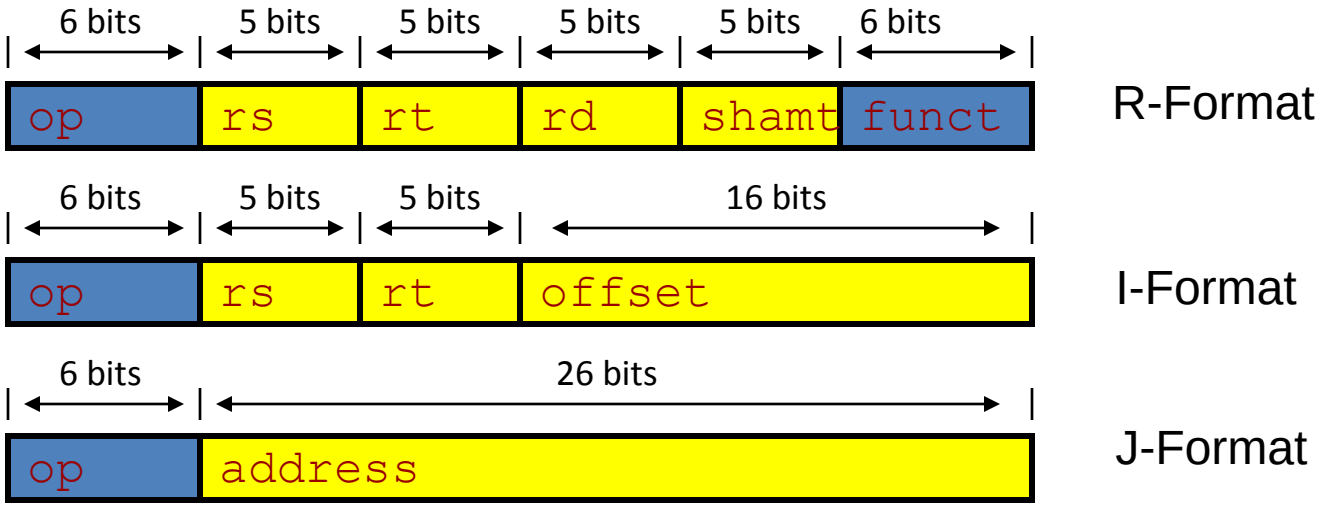
PC = PC[31:28] : (IR[25:0] << 2)

Concatenate upper 4 bits of PC to form complete 32-bit address

Conversion to word offset

Summary - MIPS Instruction Set

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats



Summary: MIPS Instructions

MIPS assembly language				
Category	Instruction	Example	Meaning	Comments
	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
Arithmetic	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 – $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
	load word	`lw $s1, 100($s2)`	`$s1 =` Memory[`$s2` + 100	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Word from register to memory
Data transfer	load byte	`lb $s1, 100($s2)`	`$s1 =` Memory[`$s2` + 100	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
Conditional branch	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
	jump	`j 2500`	go to 10000	Jump to target address
Uncondi-	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
tional jump	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

Addressing Modes