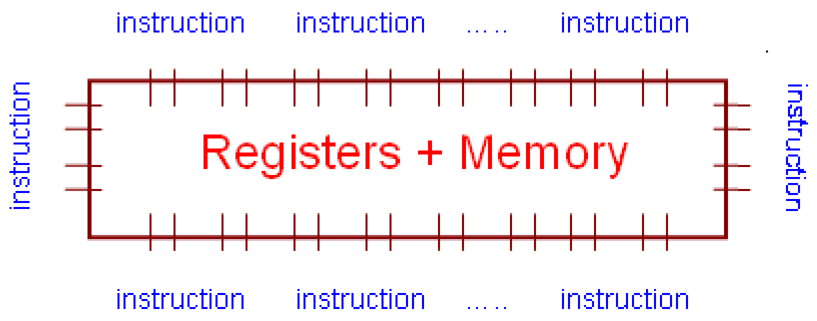
Computer Architecture

MIPS Instruction Set Architecture

Instruction Set Architecture

- An Abstract Data Type
  - Objects ≡ Registers & Memory
  - Operations ≡ Instructions

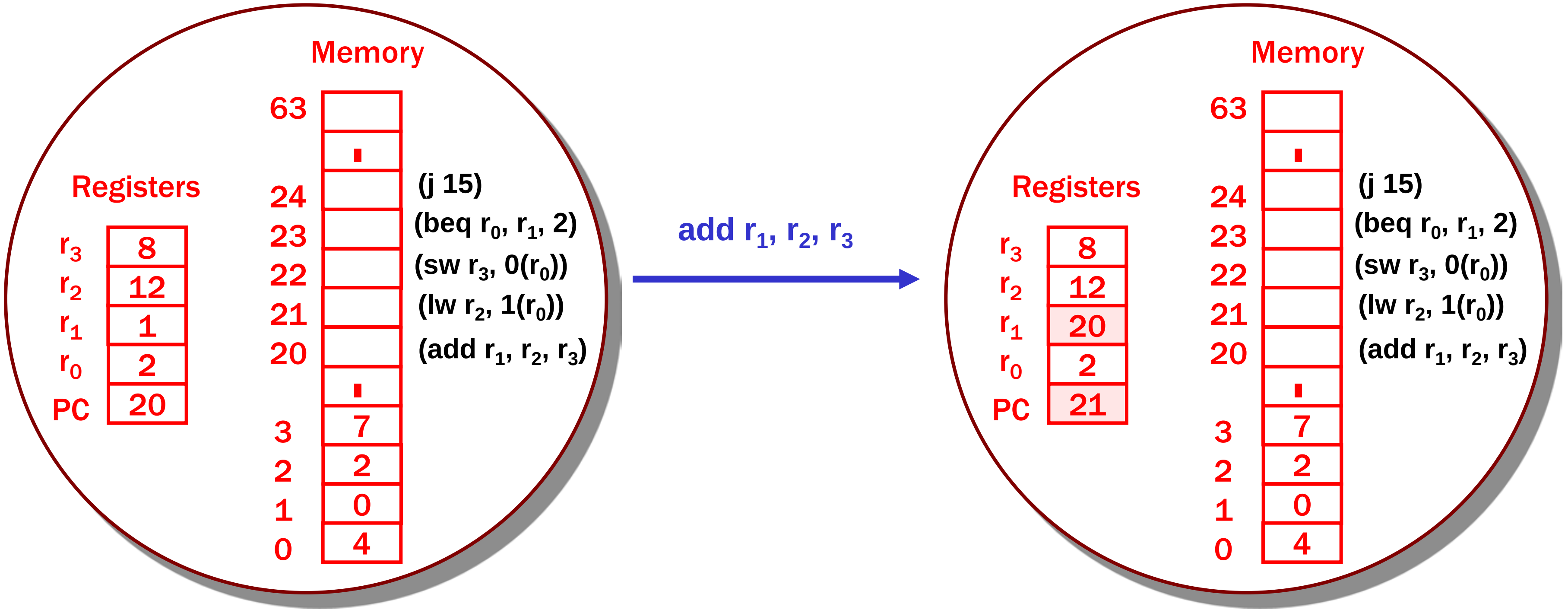


- Goal of Instruction Set Architecture Design
  - To allow high-performance & low-cost implementations while satisfying constraints imposed by applications including operating system and complier

Instruction Set Architecture as an ADT (Review)

Assumptions

- 8 bit ISA
- # of registers = 4 + PC (Program Counter)
- Memory size = 64B

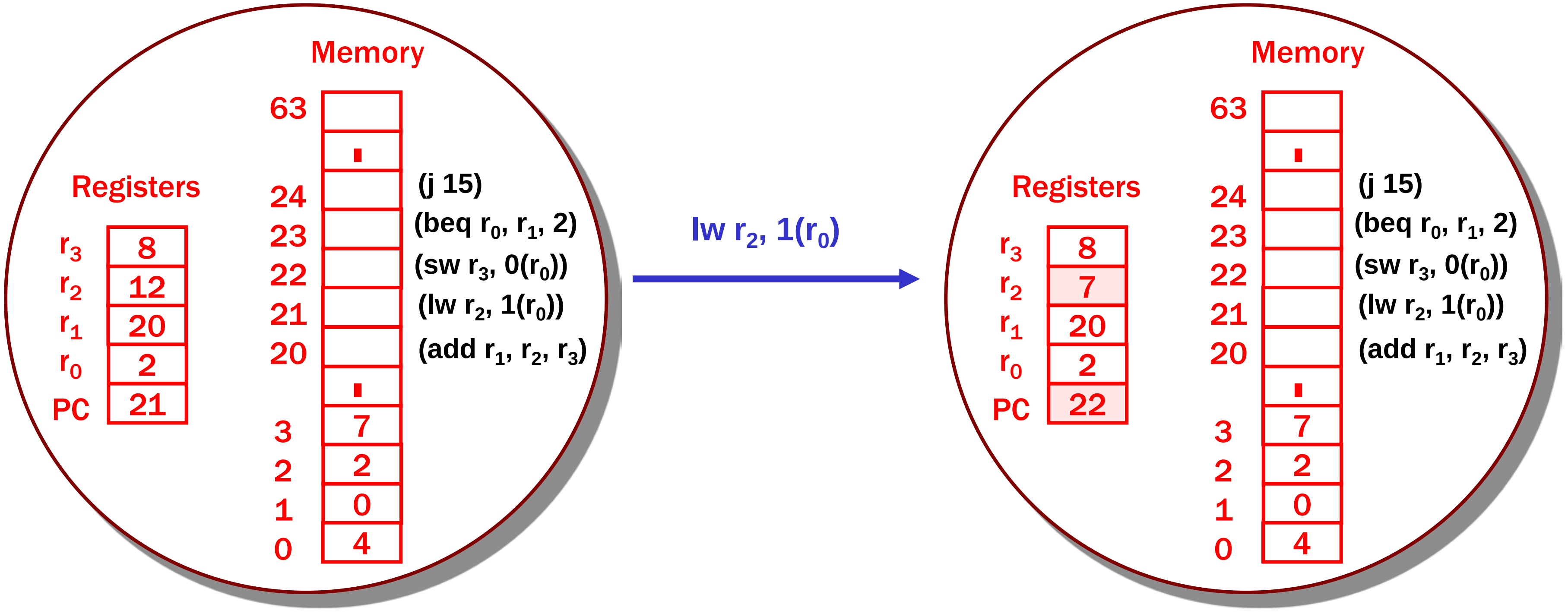


Before Register and Memory **After Register and Memory**

Instruction Set Architecture as an ADT (Review)

Assumptions

- 8 bit ISA
- # of registers = 4 + PC (Program Counter)
- Memory size = 64B

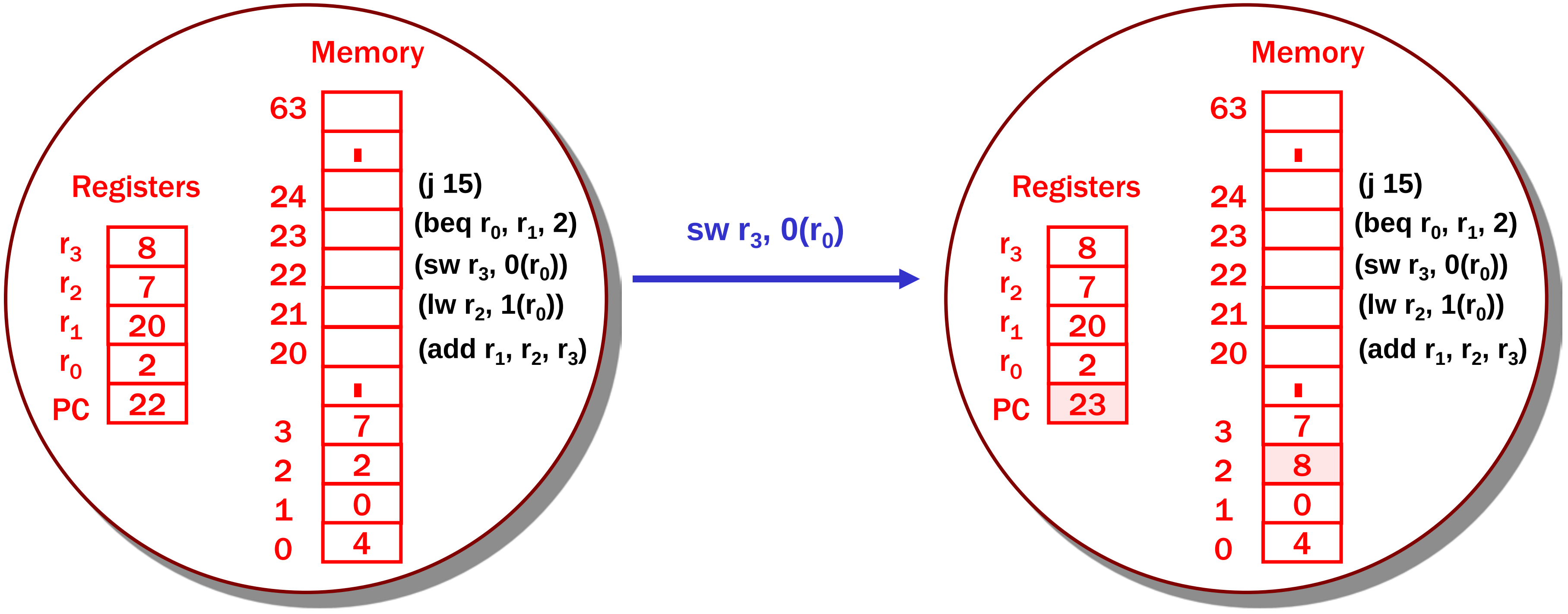


Before Register and Memory **After Register and Memory**

Instruction Set Architecture as an ADT (Review)

Assumptions

- 8 bit ISA
- # of registers = 4 + PC (Program Counter)
- Memory size = 64B

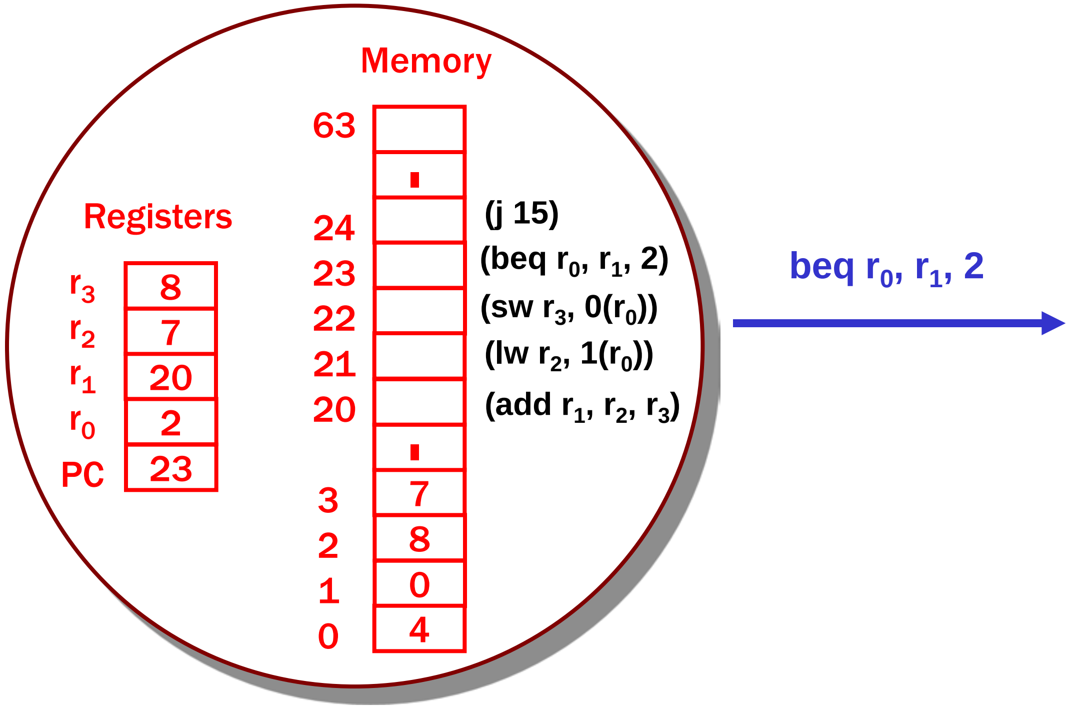


Before Register and Memory **After Register and Memory**

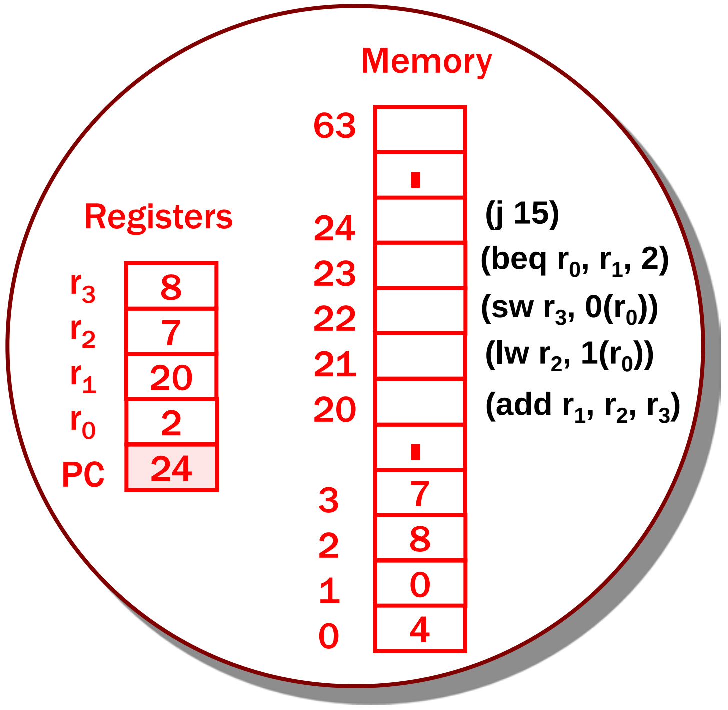
Instruction Set Architecture as an ADT (Review)

Assumptions

- 8 bit ISA
- # of registers = 4 + PC (Program Counter)
- Memory size = 64B



Before Register and Memory

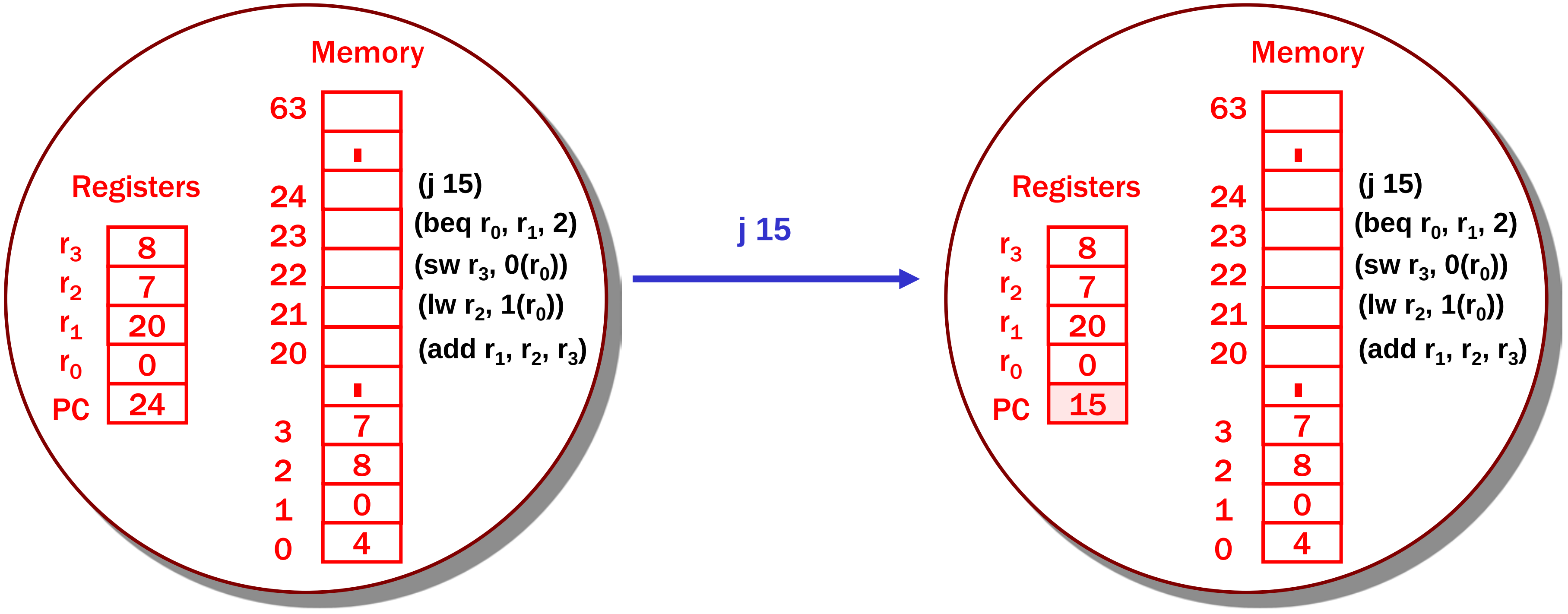


After Register and Memory

Instruction Set Architecture as an ADT (Review)

Assumptions

- 8 bit ISA
- # of registers = 4 + PC (Program Counter)
- Memory size = 64B



Before Register and Memory **After Register and Memory**

Examples of ISAs and Implementations

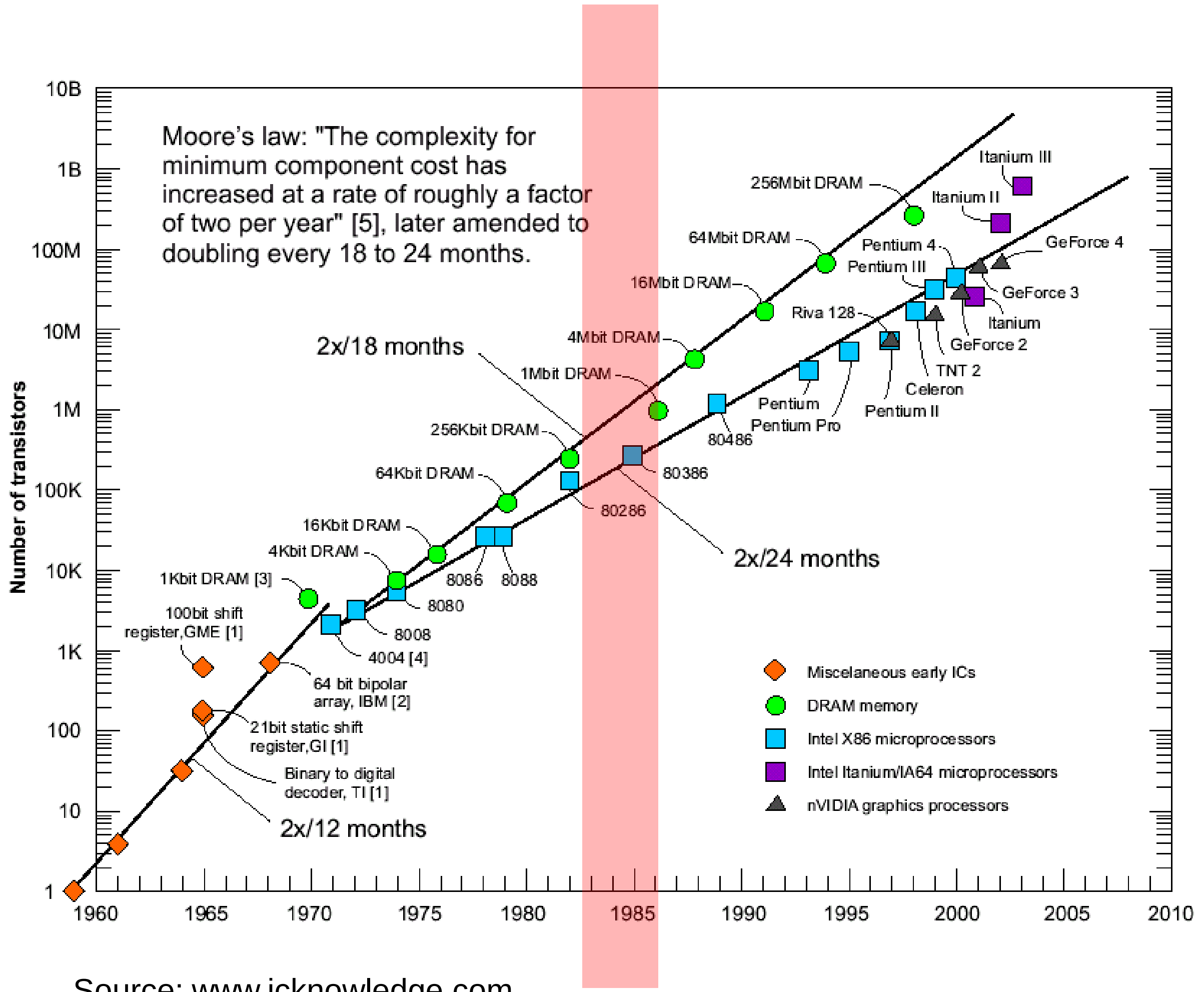
- Instruction Set Architectures
  - IBM System/360, IA-32 (x86), IA-64, MIPS, SPARC, Alpha, PA-RISC, ARM, ...
- Implementations
  - IA-32 (x86)
    - Intel: 8086, 8088, 80186, 80286, 80386, 80486, Pentium, Pentium Pro, Pentium II, Celeron, Pentium III, Pentium 4, ...
    - AMD: K5, K6, K6-II, K6-III, Athlon, Duron, ...
    - Cyrix: 80486, 5x86, 6x86,...
  - IA-64: Itanium, Itanium 2, ...
  - Alpha: 21064, 21164, 21264, 21364, ...

History

- Hot topics in Computer Architecture
  - High-level language computer architectures in the 1970s
  - RISC architectures in the early 1980s
  - Shared-memory multiprocessors in the late 1980s
  - Out-of-order speculative execution processors in the 1990s
  - Multi-core architectures in the 2000s

From "Single-Chip Multiprocessors: the Rebirth of Parallel Architecture" by Prof. Guri Sohi

A Critical point in VLSI Technology



Source: www.icknowledge.com

History of RISC Architecture

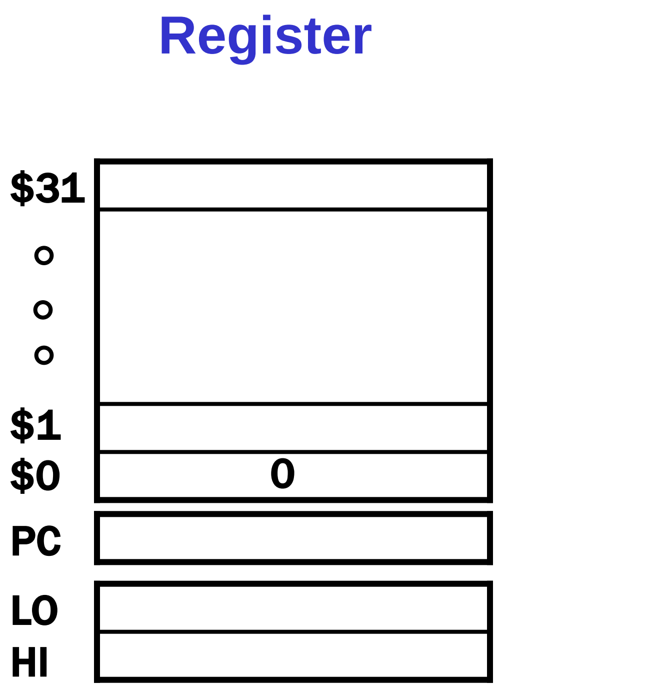
- Integration of processors on a single chip
  - A critical point ("*epoch*")
  - Argued for different architectures (RISC)
    - Small repertoire of instructions in a uniform format
    - Pipelined execution
    - Cache memory
    - Load/store architecture
- More transistors allowed for different optimizations
  - Large/multi-level caches
  - Co-processors
  - Superscalar
  - etc

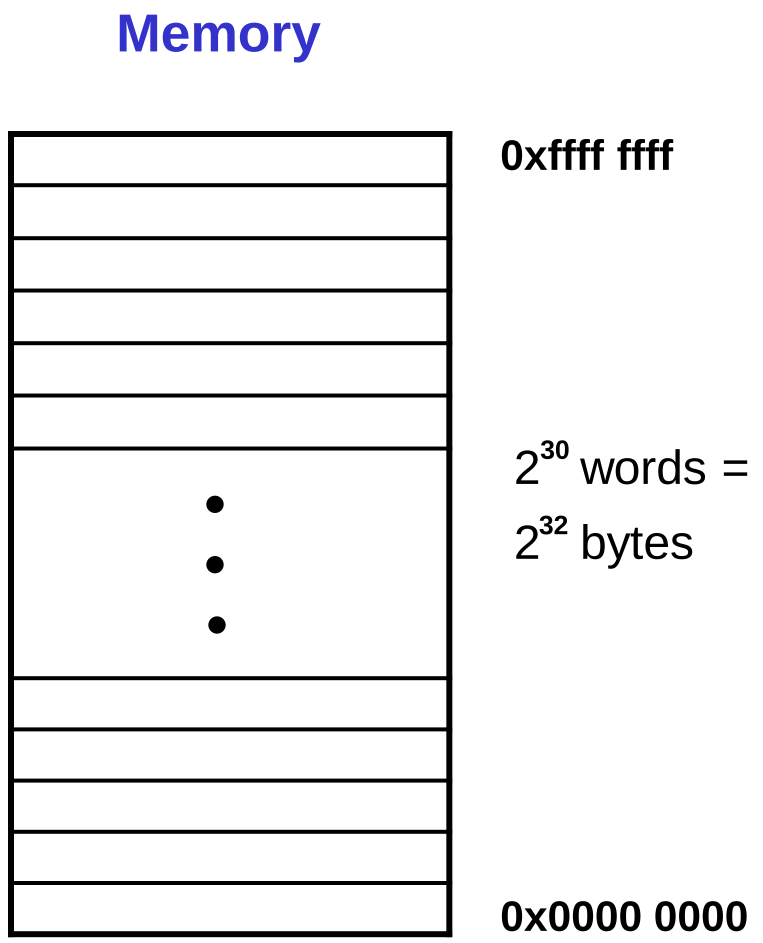
From "Single-Chip Multiprocessors: the Rebirth of Parallel Architecture" by Prof. Guri Sohi

MIPS Instruction Set Architecture

- One of the Pioneering RISC Instruction Set Architectures
  - Small repertoire of instructions in a uniform format
  - Pipelined execution
  - Cache memory
  - Load/store architecture
- Starts with a 32-bit architecture, later extended to 64-bit
- Even currently used in many embedded applications
  - Game consoles – Nintendo 64, PlayStation, PlayStation 2, etc
  - Network devices – IP phone, WLAN Access points, etc
  - Residential Devices – High Definition TV, Digital Photo Frame,
  - etc

MIPS ISA State (Register & Memory)





2^{30} words = 2^{32} bytes

MIPS Register Usage (Software Convention for Interoperability)

0	$zero	constant 0
1	$at	reserved for assembler
2	$v0	return values
3	$v1	
4	$a0	arguments
5	$a1	
6	$a2	
7	$a3	
8	$t0	temporary
. . .		
15	$t7	
16	$s0	permanent (For variables in a high-level language program)
. . .		
23	$s7	
24	$t8	temporary
25	$t9	
26	$k0	OS kernel (reserved)
27	$k1	
28	$gp	global pointer
29	$sp	stack pointer
30	$fp	frame pointer
31	$ra	return address

MIPS Instructions

- Arithmetic/Logic instructions
- Data Transfer (Load/Store) instructions
- Conditional branch instructions
- Unconditional jump instructions

MIPS Instruction Format

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

MIPS Integer Arithmetic Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1, \$s2, \$s3	\$s1 = \$s2 + \$s3	3 operands; exception possible
	add immediate	addi \$s1, \$s2, 100	\$s1 = \$s2 + 100	+ constant; exception possible
	add unsigned	addu \$s1, \$s2, \$s3	\$s1 = \$s2 + \$s3	3 operands; no exceptions
	add immediate unsigned	addiu \$s1, \$s2, 100	\$s1 = \$s2 + 100	+ constant; no exceptions
	subtract	sub \$s1, \$s2, \$s3	\$s1 = \$s2 – \$s3	3 operands; exception possible
	subtract unsigned	subu \$s1, \$s2, \$s3	\$s1 = \$s2 – \$s3	3 operands; no exceptions
	set less than	slt \$s1, \$s2, \$s3	\$s1 = (\$s2 < \$s3)	compare signed <
	set less than immediate	slti \$s1, \$s2, 100	\$s1 = (\$s2 < 100)	compare signed < constant
	set less than unsigned	sltu \$s1, \$s2, \$s3	\$s1 = (\$s2 < \$s3)	compare unsigned <
	set less than immediate unsigned	sltiu \$s1, \$s2, 100	\$s1 = (\$s2 < 100)	compare unsigned < constant

MIPS Integer Arithmetic Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	multiply	mult $s2, $s3	HI, LO ← $s2 x $s3	64 bit signed product
	multiply unsigned	multu $s2, $s3	HI, LO ← $s2 x $s3	64 bit unsigned product
	divide	div $s2, $s3	LO ← $s2 ÷ $s3,	LO ← quotient
			HI ← $s2 mod $s3	HI ← remainder
	divide unsigned	divu $s2, $s3	LO ← $s2 ÷ $s3,	Unsigned quotient
			HI ← $s2 mod $s3	Unsigned remainder
	move from HI	mfhi $s1	$s1 ← HI	Used to get copy of HI
	move from LO	mflo $s1	$s1 ← LO	Used to get copy of LO

MIPS Logical Instructions

Category	Instruction	Example	Meaning	Comments
Logical	and	and $s1, $s2, $s3	$s1 = $s2 & $s3	Thee reg. operands; bit–by–bit AND
	and immediate	andi $s1, $s2, 100	$s1 = $s2 & 100	Bit–by–Bit AND reg with constant
	or	or $s1, $s2, $s3	$s1 = $s2 | $s3	Thee reg. operands; bit–by–bit OR
	or immediate	ori $s1, $s2, 100	$s1 = $s2 | 100	Bit–by–Bit OR reg with constant
	xor	xor $s1, $s2, $s3	$s1 = $s2 xor $s3	Logical XOR
	xor immediate	xori $s1, $s2, 10	$s1 = $s2 xor 10	Logical XOR w/ constant
	nor	nor $s1, $s2, $s3	$s1 = ┐ ($s2 ∨ $s3)	Logical NOR
	shift left logical	sll $s1, $s2, 10	$s1 = $s2 ≪ 10	Shift left by constant
	shift right logical	srl $s1, $s2, 10	$s1 = $s2 ≫ 10	Shift right by constant
	shift right arithmetic	sra $s1, $s2, 10	$s1 = $s2 ≫ 10	Shift right (sign extended)
	shift left logical variable	sllv $s1, $s2, $s3	$s1 = $s2 ≪ $s3	Shift left by variable
	shift right logical variable	srlv $s1, $s2, $s3	$s1 = $s2 ≫ $s3	Shift right by variable
	shift right arithmetic variable	srav $s1, $s2, $s3	$s1 = $s2 ≫ $s3	Shift right arithmetic by variable
	load upper immediate	lui $s1, 40	$s1 = 40 ≪ 16	Places immediate into upper 16 bits

MIPS Data Transfer (Load/Store) Instructions

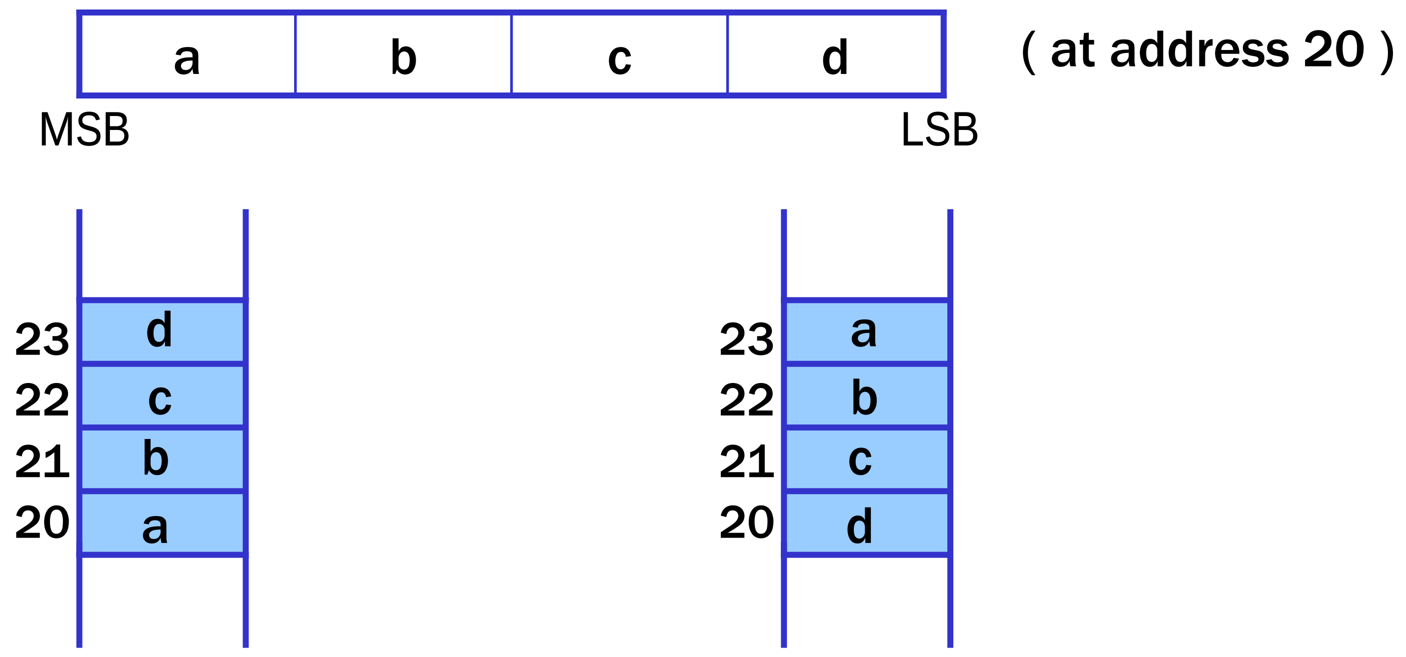
Category	Instruction	Example	Meaning	Comments
Date transfer	store word	sw $s1, 100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
	store halfword	sh $s1, 100($s2)	Memory[$s2 + 100] = $s1	Store only lower 16 bits
	store byte	sb $s1, 100($s2)	Memory[$s2 + 100] = $s1	Store only lowest byte
	store float	swc1$f1,100($s2)	Memory[$s2 + 100] = $f1	Store FP word
	load word	lw $s1, 100($s2)	$s1 = Memory[$s2 + 100]	Load word
	load halfword	lh $s1, 100($s2)	$s1 = Memory[$s2 + 100]	Load halfword; sign extended
	load half unsigned	lhu $s1, 100($s2)	$s1 = Memory[$s2 + 100]	Load halfword; zero extended
	load byte	lb $s1, 100($s2)	$s1 = Memory[$s2 + 100]	Load byte; sign extended
	load byte unsigned	lbu $s1, 100($s2)	$s1 = Memory[$s2 + 100]	Load byte; zero extended
	load float	lwc1 $f1,100($s2)	$f1 = Memory[$s2 + 100]	Load FP register
	load upper immediate	lui $s1, 100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits

More about Loads and Stores

- All memory accesses are exclusively through loads and stores (load-store architecture)
- Alignment restriction
  - Word addresses must be multiples of 4
  - Halfword addresses must be multiples of 2
- Partial word (halfword or byte) loads from memory
  - Sign-extended for signed operations
  - Zero-extended for unsigned operations

More about Loads and Stores

- Big Endian vs. Little Endian



Big Endian

(Macintosh, Sun SPARC)

Little Endian

(DEC Station 3100, Intel 80x86)

MIPS Conditional Branch Instructions

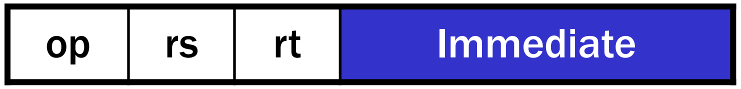
Category	Instruction	Example	Meaning	Comments
conditional branch	branch on equal	beq $s1, $s2, L	if($s1==$s2) go to L	Equal test and branch
	branch on not equal	bne $s1, $s2, L	if($s1!=$s2) go to L	Not equal test and branch

MIPS Unconditional Jump Instructions

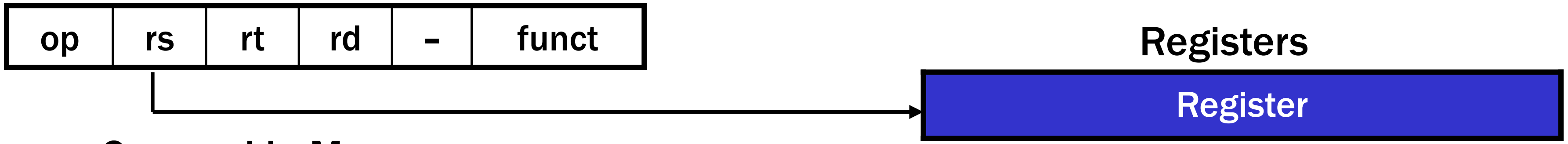
Category	Instruction	Example	Meaning	Comments
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4 go to 10000	For procedure call

MIPS Addressing Modes

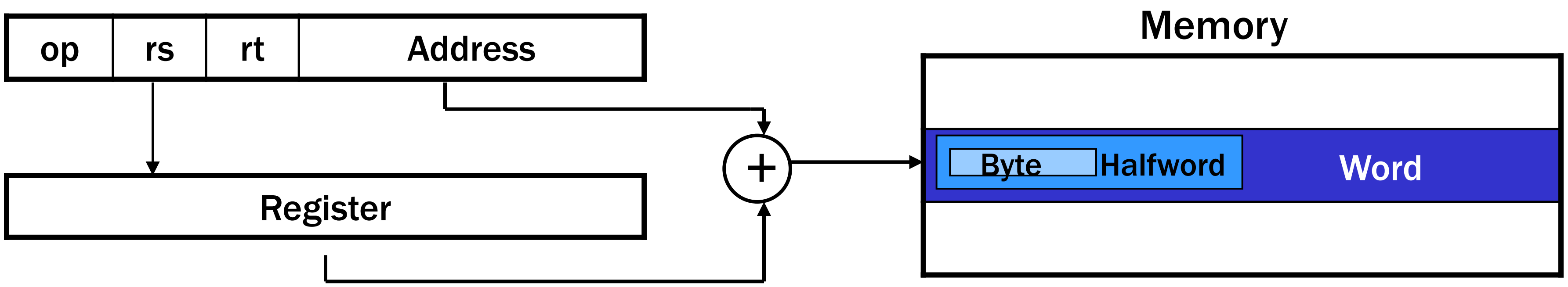
- Operand in instruction itself
  - Immediate addressing



- Operand in register
  - Register addressing

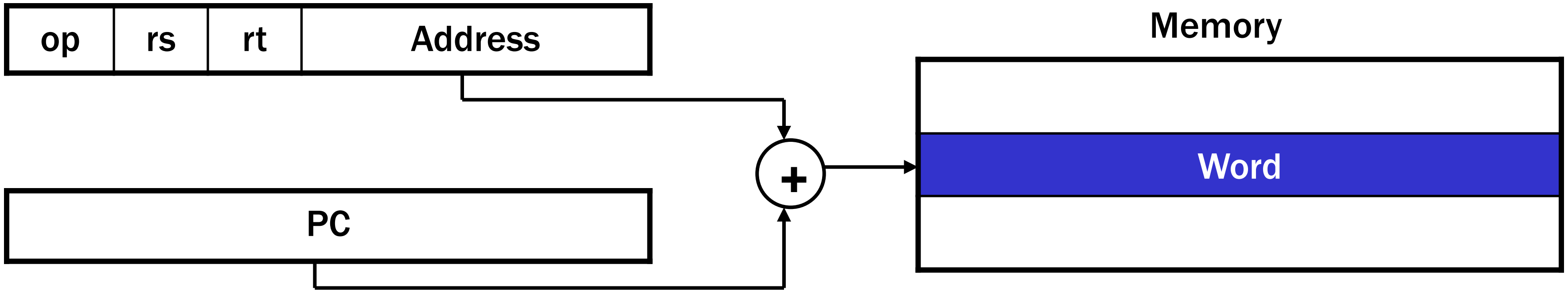


- Operand in Memory
  - Base addressing

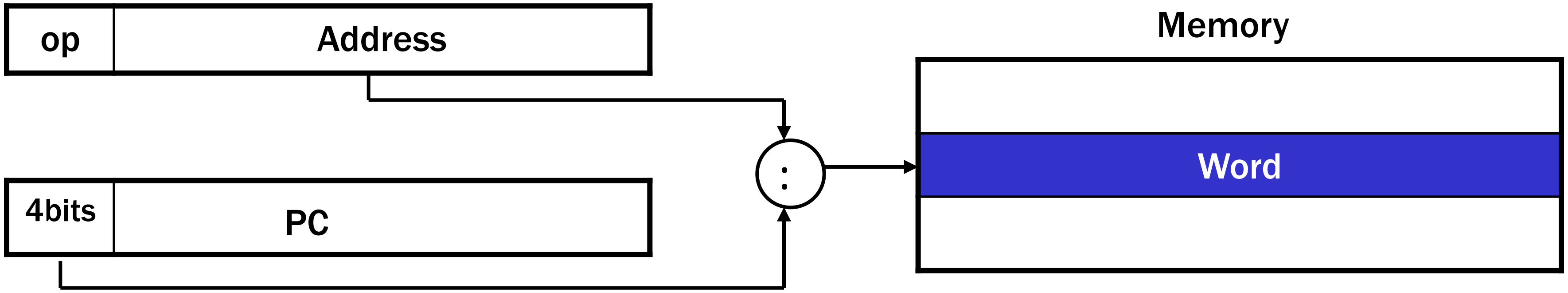


MIPS Addressing Modes

- Instruction in memory
  - PC-relative addressing (branch)

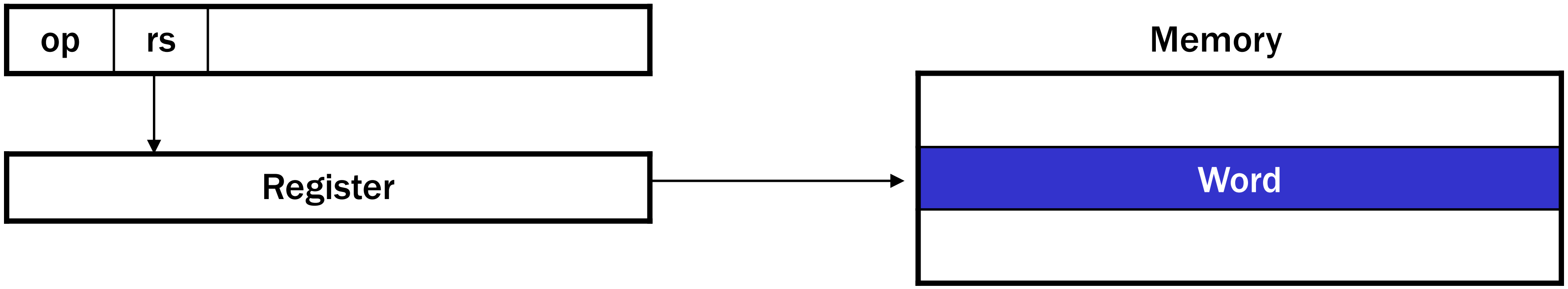


  - Pseudo-direct addressing (jump)



MIPS Addressing Modes

- Instruction in memory
  - Register (jump register)



Addressing Modes (Many not supported in MIPS)

Addressing mode	Example Instruction	Meaning	When used
Register	Add R4, R5, R3	R4 ← R5 + R3	When a value is in a register.
Immediate or literal	Add R4, R5, #3	R4 ← R5 + 3	For constants.
Displacement or based	Add R4, R5, 100(R1)	R4 ← R5 + M[100+R1]	Accessing local variables.
Register deferred or indirect	Add R4, R5, (R1)	R4 ← R5 + M[R1]	Accessing using a pointer or a computed address.
Indexed	Add R3, R5, (R1+R2)	R3 ← R5 + M[R1+R2]	Sometimes useful in array addressing R1=base of array, R2=index amount.
Direct or absolute	Add R1, R5, (1001)	R1 ← R5 + M[1001]	Sometimes useful for accessing static data: address constant may need to be large.
Memory indirect or memory deferred	Add R1, R5, @(R3)	R1 ← R5 + M[M[R3]]	If R3 is the address of a pointer p, then mode yields *p.
Auto-increment	Add R1, R5, (R2)+	R1 ← R5 + M[R2] R2 ← R2 + d	Useful for stepping through arrays within a loop. R2 pointers to start of array; each reference increments R2 by size of an element, d.
Auto-decrement	Add R1, R5, -(R2)	R2 ← R2 - d R1 ← R5 + M[R2]	Same use as autoincrement. Autoincrement/decrement can also be used to implement a stack as push and pop.
Scaled or index	Add R1, R5, 100(R2)[R3]	R1 ← R5 + M[100+R2+R3*d]	Used to index arrays. May be applied to any base addressing mode in some machines.

C vs. Assembly

C

```
f = (g + h) – (i + j);
```

f is mapped to s0
g is mapped to s1
h is mapped to s2
i is mapped to s3
j is mapped to s4

Assembly

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

C vs. Assembly

C

```
g = h + A[i];
```

g is mapped to s1
h is mapped to s2
s3 contains the base
  address of array A[].
i is mapped to s4.

Assembly

```
add $t1, $s4, $s4
add $t1, $t1, $t1
add $t1, $t1, $s3
lw   $t0, 0($t1)
add $s1, $s2, $t0
```

C vs. Assembly

C

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

```
f  is mapped to s0
g is mapped to s1
h is mapped to s2
i  is mapped to s3
j  is mapped to s4
```

Assembly

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j     Exit
Else: sub $s0, $s1, $s2
Exit:
```

C vs. Assembly

C

```
while (save[i] == k)
        i = i + j;
```

i is mapped to s3
j is mapped to s4
k is mapped to s5
s6 contains the base address of array save[].

Assembly

```
Loop: add $t1, $s3, $s3
      add $t1, $t1, $t1
      add $t1, $t1, $s6
      lw  $t0, 0($t1)
      bne $t0, $s5, Exit
      add $s3, $s3, $s4
      j   Loop
Exit:
```

C vs. Assembly

C

```
switch (k) {
    case 0: f = i + j;   break;
    case 1: f = g + h; break;
    case 2: f = g - h;  break;
    case 3: f = i - j;    break;
}
```

f is mapped to s0
g is mapped to s1
h is mapped to s2
i is mapped to s3
j is mapped to s4
k is mapped to s5
t2 contains 4

Assembly

```
      slt   $t3, $s5, $zero
      bne   $t3, $zero, Exit
      slt   $t3, $s5, $t2
      beq  $t3, $zero, Exit
      add  $t1, $s5, $s5
      add  $t1, $t1, $t1
      add  $t1, $t1, $t4
      lw    $t0, 0($t1)
      jr    $t0
L0: add  $s0, $s3, $s4
      j Exit
L1: add  $s0, $s1, $s2
      j Exit
L2: sub  $s0, $s1, $s2
      j Exit
L3: sub  $s0, $s3, $s4
Exit
```

Pseudo Instructions

- Pseudo instruction : Instructions that are available in assembly language but not implemented in hardware
- Pseudo instruction examples

Pseudo instruction	Equivalent real instruction sequence
move $s1, $s2	add $s1, $zero, $s2
blt $s1, $s2, label	slt $at, $s1, $s2 bne $at, $zero, label
abs $s1, $s2	add $s1, $zero, $s2 slt $at, $s2, $zero beq $at, $zero, L sub $s1, $zero, $s1 L:
⋮	⋮