
Computer Science 141

Computing Hardware

Fall 2009

Harvard University

Instructor: David Brooks
dbrooks@eecs.harvard.edu

Instruction Set Architecture

“Instruction Set Architecture is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (timing independent) program for that machine.”

IBM, Introducing the IBM 360 (1964)

- The ISA defines:
 - Operations that the processor can execute
 - Data Transfer mechanisms + how to access data
 - Control Mechanisms (branch, jump, etc)
 - “Contract” between programmer/compiler + HW
- CS141 focuses on MIPS architecture
 - Chapter 6 of Harris&Harris has more detail/examples

ISA Design Choices

- types of **operations** supported
 - e.g. arithmetic/logical, data transfer, control transfer, system, floating-point, decimal (BCD), string
- types of **operands** supported
 - e.g. byte, character, digit, halfword, word (32-bits), doubleword, floating-point number
- types of **operand storage** allowed
 - e.g. stack, accumulator, registers, memory
- **implicit** vs. **explicit operands** in instructions and number of each
- **orthogonality** of operands, operand location, and addressing modes

Classification of ISAs

Type of Architecture	for ALU instruction	
	Source operands	Destination
Stack	Top 2 elements on stack	Top of stack
Accumulator	Accumulator (1) Memory (other)	Accumulator
Register set	Registers or memory	Registers or memory

Register

- Examples: IBM 360, DEC VAX, Motorola 680x0, RISC, etc.
- Most common approach
 - Fast, temporary storage (small)
 - Explicit operands (register IDs)
- Example: $C = A + B$

Register-memory

Load R1, A

Add R3, R1, B

Store R3, C

load/store (reg-reg)

Load R1, A

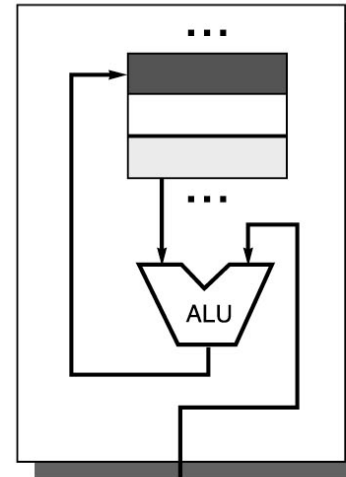
Load R2, B

Add R3, R1, R2

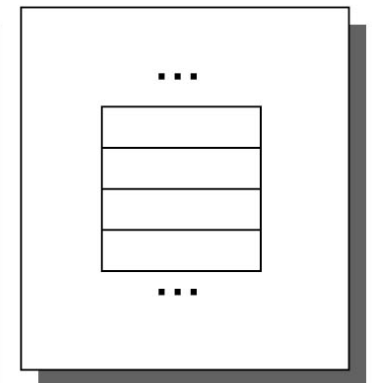
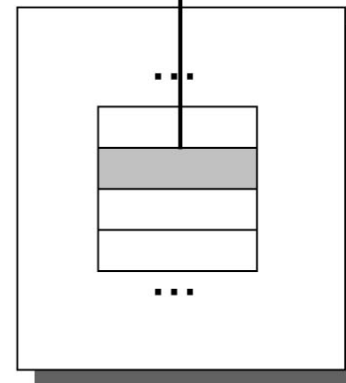
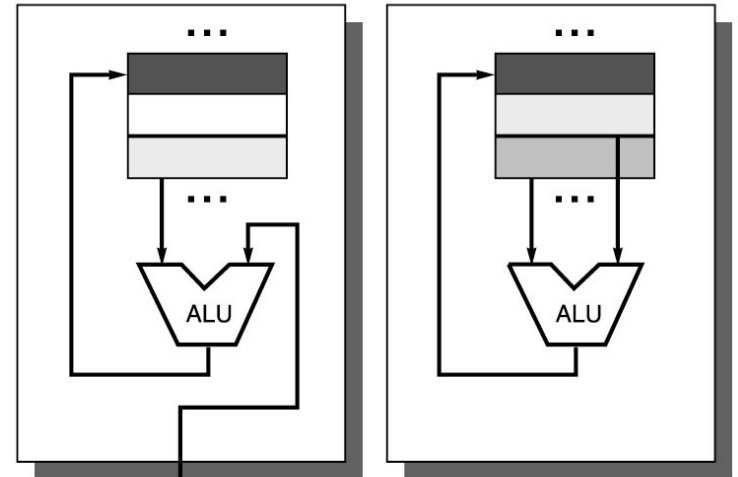
Store R3, C

- All RISC ISAs are load/store
- IBM 360, Intel x86, Moto 68K are register-memory

(c) Register-memory



(d) Register-register/load-store



What leads to a good/bad ISA?

- Ease of Implementation (Job of Architect/Designer)
 - Does the ISA lead itself to efficient implementations?
- Ease of Programming (Job of Programmer/Compiler)
 - Can the compiler use the ISA effectively?
- Future Compatibility
 - ISAs may last 30+yrs
 - Special Features, Address range, etc. need to be thought out

Operations

- Arithmetic
 - Add, subtract, multiply, divide
 - Register, immediate
 - Signed, unsigned, integer, floating-point
- Logical
 - Conjunction, disjunction, shift left, shift right
 - Register, immediate
- Data Transfer
 - Load, store
- Conditional Branch
 - Branch on equal, not equal
 - Set on less than
- Unconditional Jump

Operands

- Registers
 - MIPS has 32 architected 32-bit registers
 - Effective use of registers is key to program performance
- Data Transfer
 - 32 words in registers, millions of words in main memory
 - Instruction accesses memory via memory address
 - Address indexes memory, a large single-dimensional array
- Alignment
 - Most architectures address individual bytes
 - Addresses of sequential words differ by 4
 - Words must always start at addresses that are multiples of 4

Assembly Examples

- Registers and Instructions
 - Registers corresponding to variables in C program (\$s0, \$s1, ...)
 - Temporary registers needed to compile into MIPS (\$t0, \$t1, ...)
 - add/sub \$s2, \$s0, \$s1 # \$s2 = \$s0 +/- \$s1
 - lw \$s1, k(\$s2) # \$s1 = A[k] where \$s2 is base of array
 - sw \$s1, k(\$s2) # A[k] = \$s1 where \$s2 is base of array
- Example 1: Compile the C assignment “f=(g+h)-(i+j)” assuming f, g, h, i are assigned to registers \$s0, \$s1, \$s2, \$s3, \$s4, respectively.

Assembly Examples

- Solution 1: Compile the C assignment “ $f=(g+h)-(i+j)$ ” assuming f, g, h, i are assigned to registers $\$s0, \$s1, \$s2, \$s3, \$s4$, respectively.

```
add $t0, $s1, $s2      # register $t0 contains g+h
add $t1, $s3, $s4      # register $t1 contains i+j
sub $s0, $t0, $t1      # f gets $t0-$t1
```

- Example 2: Assuming variable h is associated with $\$s2$ and base address of A is in $\$s3$, what is MIPS assembly for “ $A[12] = h + A[8]$ ”?

Assembly Examples

- Solution 2: Assuming variable h is associated with \$s2 and base address of A is in \$s3, what is MIPS assembly for “A[12] = h + A[8]”?

```
lw $t0, 32($s3)           # temporary reg $t0 gets A[8]
add $t0, $s2, $t0         # temporary reg $t0 gets h+A[8]
sw $t0, 48($s3)          # store $t0 into A[12]
```

Why are the offsets 32/48 for load/store?

- Example 3: Assuming variables g, h, i are associated with \$s1, \$s2, \$s4 and base address of A is in \$s3, what is MIPS assembly for “g = h + A[i]”?

Assembly Examples

- Solution 3: Assuming variables g, h, i are associated with \$s1, \$s2, \$s4 and base address of A is in \$s3, what is MIPS assembly for “g = h + A[i]”?

```
add $t1, $s4, $s4      # $t1 = 2*i
add $t1, $t1, $t1      # $t1 = 4*I
add $t1, $t1, $s3      # $t1 addresses A[i], 4*i+$s3
lw $t0, 0($t1)         # $t0 = A[i]
add $s1, $s2, $t0      # g = h + A[i]
```

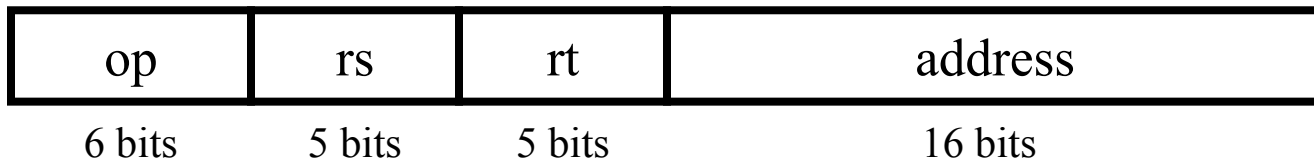
Machine Instructions

- Data and instructions both represented in bits
 - 32-bit architectures employ 32-bit instructions
 - Combination of fields specifying operations/operands
- R-Type Instruction Fields
 - op: basic operation of instruction, called the *opcode*
 - rs, rt: first, second register source operand
 - rd: register destination operand
 - shamt: shift amount for shift instructions
 - funct: specifies a variant of the operation, called *function code*

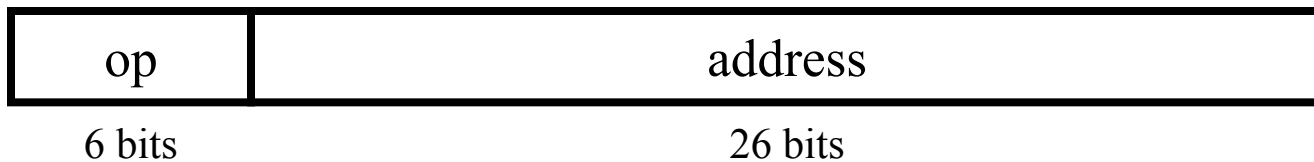


Machine Instructions

- I-Type Instruction Fields
 - Opcode specifies instruction format



- J-Type Instruction Fields



Machine Examples

- MIPS Instruction Encoding
 - \$t0-\$t7 map to 8-15
 - \$s0-\$s7 map to 16-23

Instruction	Format	Op	Rs	Rt	Rd	Shamt	Funct	Address
add	R	0	reg	reg	reg	0	32	n/a
sub	R	0	reg	reg	reg	0	34	n/a
lw	I	35	reg	reg	n/a	n/a	n/a	address
sw	I	45	reg	reg	n/a	n/a	n/a	address

- Example 4: Assuming \$t1 has base of array A and \$s2 corresponds to h, compile “A[300] = h + A[300]” into machine language.

Machine Examples

- Solution 4: Assuming \$t1 has base of array A and \$s2 corresponds to h, compile “A[300] = h + A[300]” into machine language.

```
lw $t0, 1200($t1)    # $t0 gets A[300]
add $t0, $s2, $t0    # $t0 gets h + A[300]
sw $t0, 1200($t1)    # store $t0 back into A[300]
```

Instruction	Format	Op	Rs	Rt	Rd	Address/Shamt	Funct
lw \$t0, 1200(\$t1)	I	35	9	8		1200	
add \$t0, \$s2, \$t0	R	0	18	8	8	0	32
sw \$t0, 1200(\$t1)	I	43	9	8		1200	

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Control Flow Instructions

- Branches
 - `beq $t0, $t1, L1` # go to statement labeled L1 if $\$t0 == \$t1$
 - `bne $t0, $t1, L1` # go to statement labeled L1 if $\$t0 \neq \$t1$
 - `j L1` # jump to statement labeled L1
 - Enables “if”, “if-then-else”, “loops”, “while”
- Relative Comparison
 - `slt $t0, $s3, $s4` # if $\$s3 < \$s4$, set $\$t0=1$, else set $\$t0=0$
- Example 5: Assuming variables f-j correspond to $\$s0$ - $\$s4$, compile “if (i==j) f = g + h; else f = g - h;”

Control Examples

- Solution 5: Assuming variables f-j correspond to \$s0-\$s4, compile “if (i==j) f = g + h; else f = g - h;”

```
        bne    $s3, $s4, ELSE    # go to ELSE if i != j
        add    $s0, $s1, $s2    # f = g + h
        j      EXIT             # exit “if-then-else” stmt
ELSE:   sub    $s0, $s1, $s2    # f = g - h (skipped if i == j)
EXIT:
```

- Example 6: Assuming variables i-k correspond to \$s3-\$s5 and base of array A is in \$s6, compile “while (A[i] == k) i = i + j;”

Control Examples

- Solution 6: Assuming variables i-k correspond to \$s3-\$s5 and base of array A is in \$s6, compile “while (A[i] == k) i = i + j;”

```
LOOP: add  $t1, $s3, $s3      # $t1 = 2*i
      add  $t1, $t1, $t1     # $t2 = 4*I
      add  $t1, $t1, $s6     # $t1 addresses A[i]
      lw   $t0, 0($t1)      # $t0 = A[i]
      bne  $t0, $s5, EXIT   # go to EXIT if A[i]!=k
      add  $s3, $s3, $s4    # i = i + j
      j    LOOP            # go to LOOP
```

```
EXIT:
```

Procedures

- Program must follow six steps in executing a procedure
 - Place parameters in a place where procedure can access them
 - Transfer control to procedure
 - Acquire storage resources needed for the procedure
 - Perform desired tasks
 - Place results in a place where calling program can access them
 - Return control to point of origin

- Instruction Support
 - `jal ProcedureAddress` # jump and store PC+4 into \$ra
 - `jr $ra` # jump to address

Procedures

- Register Support
 - \$t0-\$t9: ten temporary registers *not* preserved by callee
 - \$s0-\$s7: eight saved registers preserved by callee
 - \$a0-\$a3: four argument registers to pass parameters into procedure
 - \$v0-\$v1: two result registers to pass parameters out of procedure
 - \$ra: one return address register to return to point of origin
 - \$sp, \$fp: one stack and frame pointer

- Register Spilling / Stack Pointer
 - Suppose a compiler needs more registers than 4 args and 2 results
 - Existing contents of any additional registers needed (\$s0-\$s7) must be saved to memory
 - Stack pointer (\$sp) is most recently allocated address in stack

Procedures

- Program must follow six steps in executing a procedure
 - Place parameters in $\$a0$ - $\$a3$ for access by procedure
 - Jump and link to procedure address (`jal ProcedureAddr`)
 - Acquire any additional registers needed ($\$s0$ - $\$s7$ into memory beginning at address in $\$sp+4$)
 - Perform desired tasks
 - Place results in $\$v0$ - $\$v1$ for access by caller
 - Return control to point of origin (`jr $ra`)
- Example 7: Implement a recursive factorial procedure in MIPS assembly

Procedure Examples

- Solution 7: C Implementation

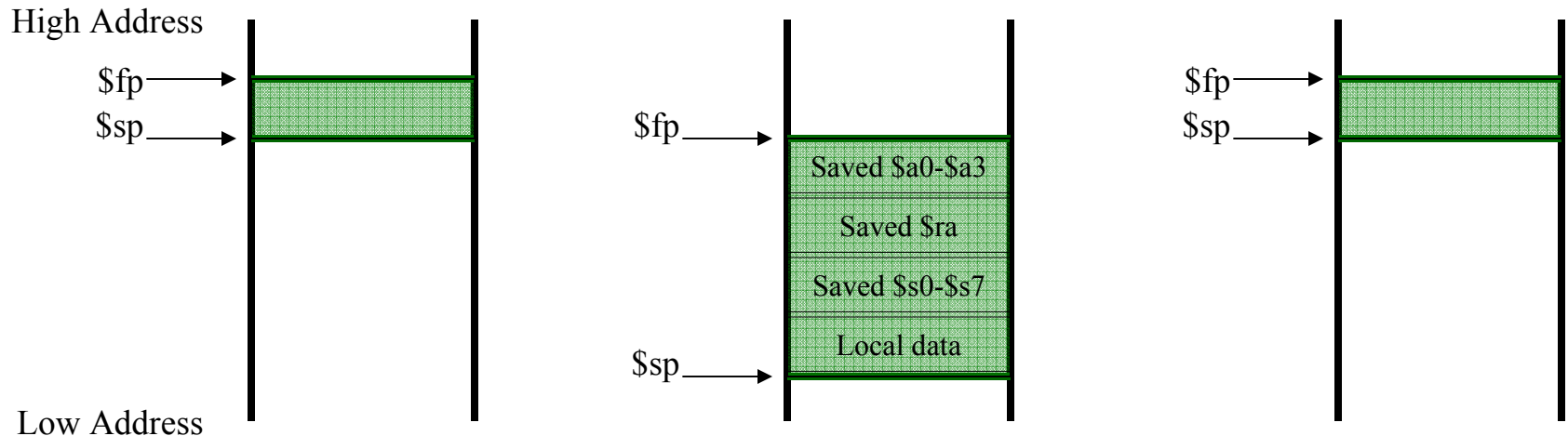
```
int fact (int n) {if (n < 1) return (1); else return (fact(n-1)*n);}
```

- MIPS Assembly

FACT:	sub	\$sp, \$sp, 8	# adjust stack for 2 items
	sw	\$ra, 4(\$sp)	# save return address
	sw	\$a0, 0(\$sp)	# save argument n
	slt	\$t0, \$a0, 1	# test if n < 1
	beq	\$t0, \$zero, L1	# if n >= 1, go to L1
	add	\$v0, \$zero, 1	# return 1
	add	\$sp, \$sp, 8	# pop 2 items off stack
	jr	\$ra	# return to address after jal
L1:	sub	\$a0, \$a0, 1	# n >= 1, \$a0 gets n-1
	jal	fact	# fact(n-1)
	lw	\$a0, 0(\$sp)	# fact returns, restore argument n
	lw	\$ra, 4(\$sp)	# restore the return address
	add	\$sp, \$sp, 8	# pop 2 items off stack
	mul	\$v0, \$a0, \$v0	# return n*fact(n-1)
	jr	\$ra	# return to caller

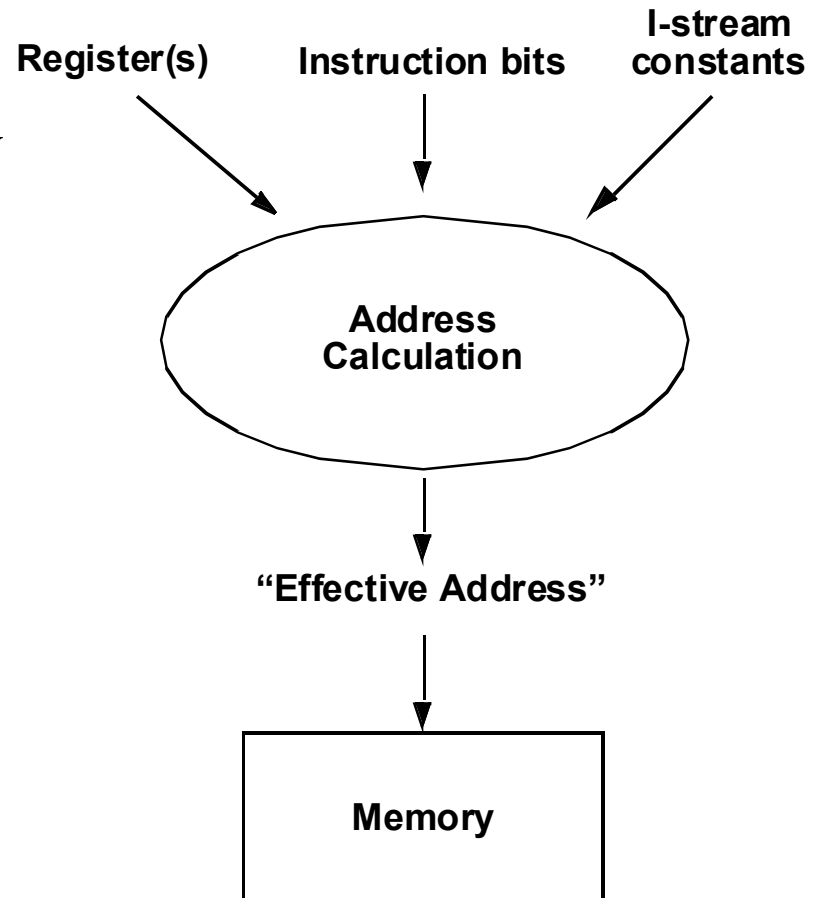
Register / Stack Management

- Register Management
 - The callee (procedure) must preserve \$s0-\$s7, \$sp, \$ra
 - The caller (program) must preserve \$t0-\$t9, \$a0-\$a3, \$v0-\$v1
- Stack Management
 - The callee must preserve the stack above stack pointer (\$sp)
 - The frame pointer (\$fp) provides a frame of reference
 - Stack allocation (a) before, (b) during, and (c) after procedure call



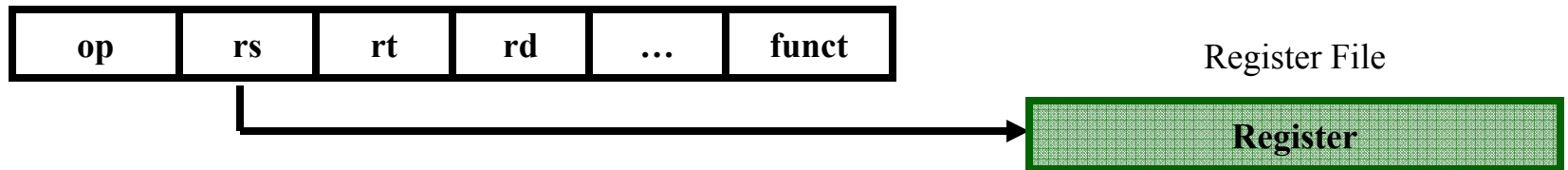
Addressing Modes

- Addressing modes describe how an instruction can find the location of its operands
- Effective address = actual memory address specified by an addressing mode

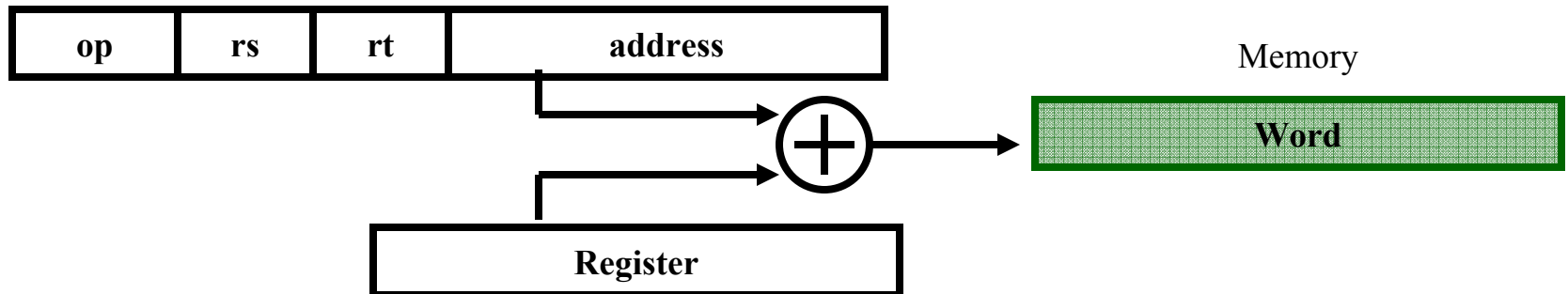


Addressing Modes

- Register Addressing
 - Operand is a register (e.g., `add $s2, $s0, $s1`)

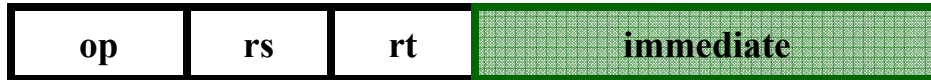


- Base or Displacement Addressing
 - Operand is at memory location whose address is sum of register and constant (e.g., `lw $s1, 8($s0)`)

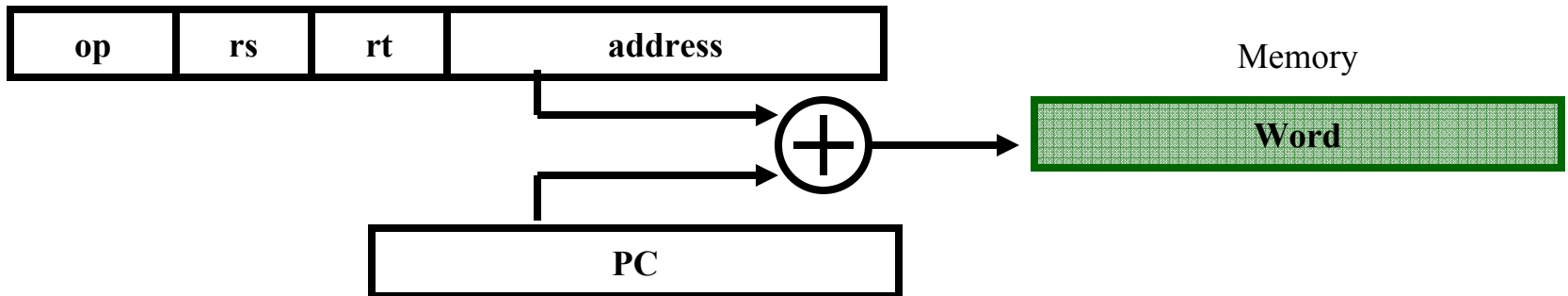


Addressing Modes

- Immediate Addressing
 - Operand is constant within instruction (e.g., `addi $s1, $s0, 4`)

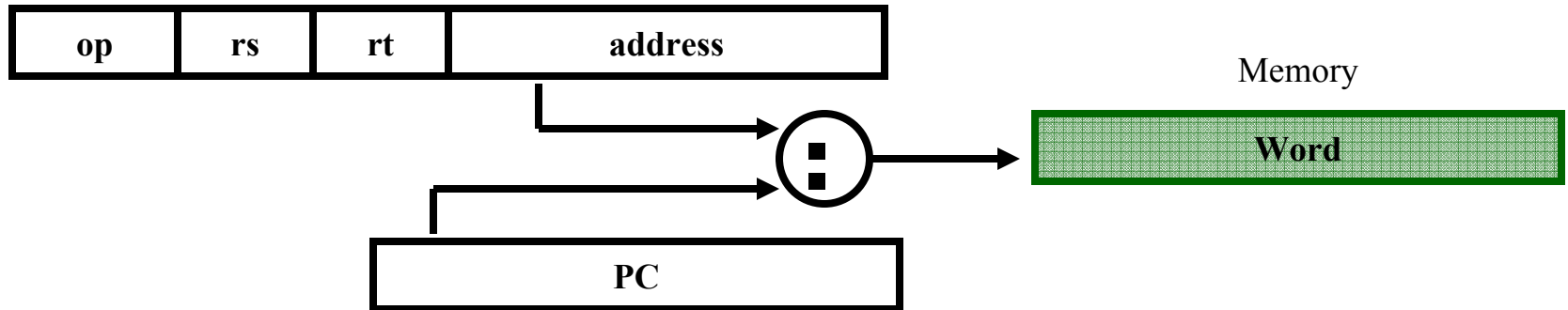


- PC-Relative Addressing
 - Address is sum of PC and constant within instruction (e.g., `beq $s0, $s1, 16`)



Addressing Modes

- Pseudo-direct Addressing
 - Address is 26 bits of constant within instruction concatenated with upper 6 bits of PC (e.g., j 1000)



Branch/Jump Addressing

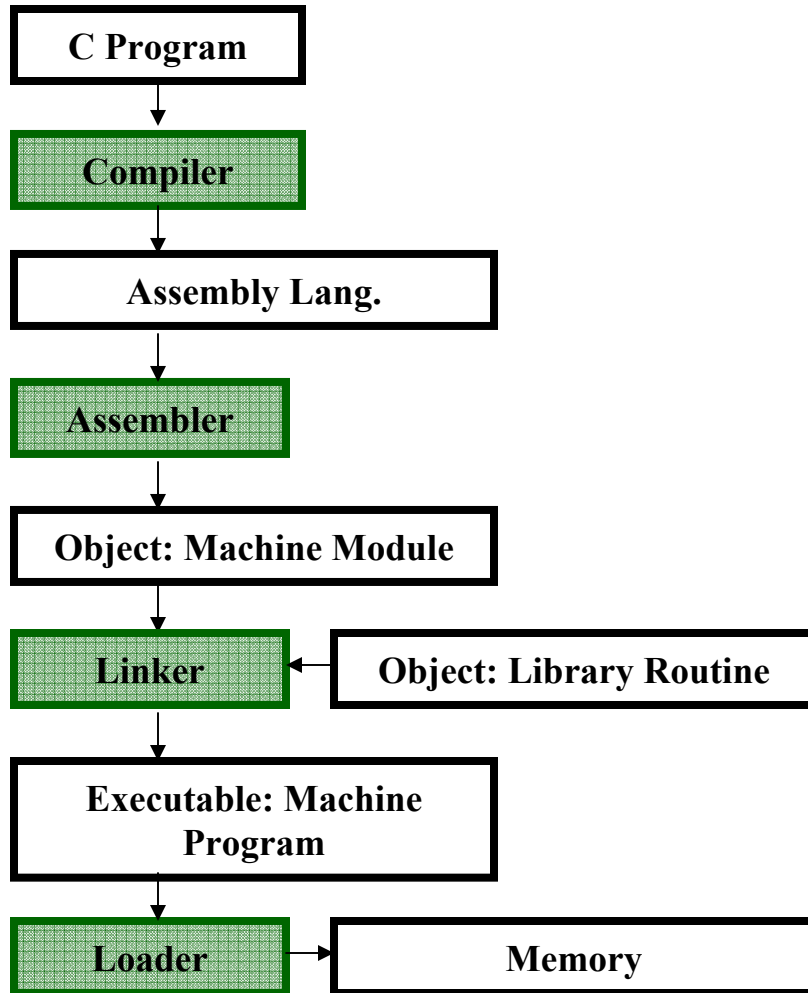
- Instruction Formats
 - Jumps (j): 6-bit opcode, 26-bit address
 - Branches (beq, bne): 6-bit opcode, 2 5-bit reg, 16-bit address
- Example 9: Given a branch on register \$s0 being equal to register \$s1, replace “beq \$s0, \$s1, L1” with a pair of instructions that offers much greater branching distance.

```
bne $s0, $s1, L2
```

```
j    L1
```

```
L2:
```

Starting a Program



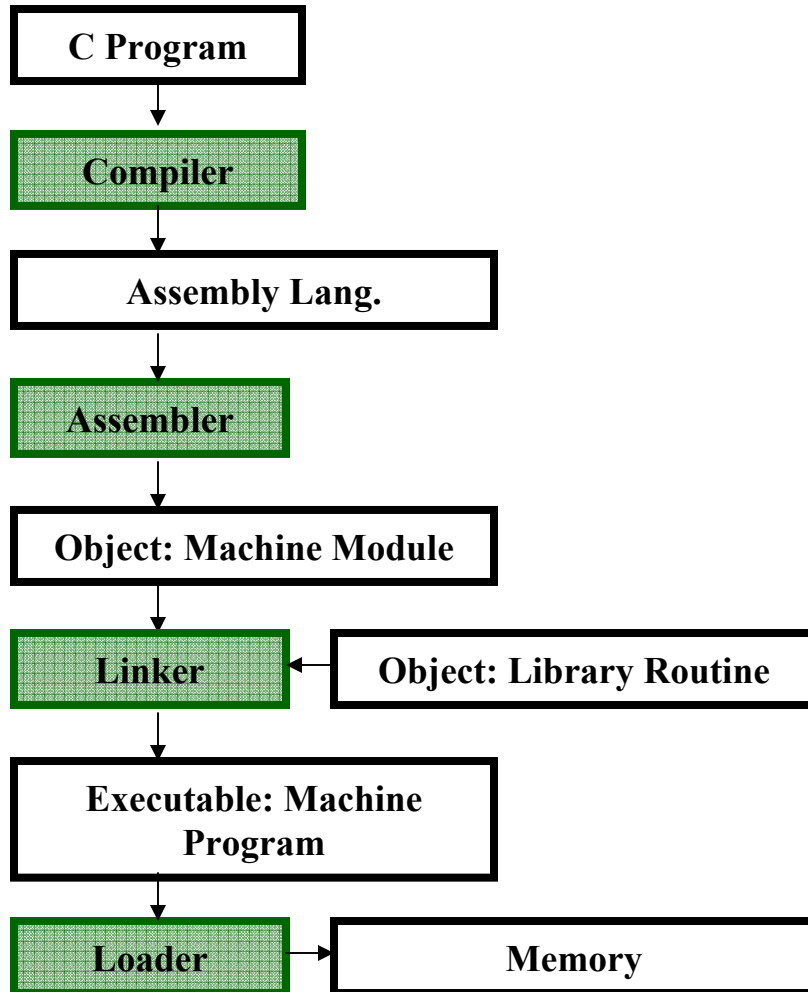
- **Compiler**

- Produces assembly code from C
- High-level languages increase programmer productivity

- **Assembler**

- Breaks pseudoinstructions (e.g., “move \$t0, \$t1” assembles “add \$t0, \$zero, \$t1”)
- Produces object file from assembly containing 6 pieces:
 1. object file handler
 2. text segment
 3. data segment
 4. relocation information
 5. symbol table
 6. debugging info

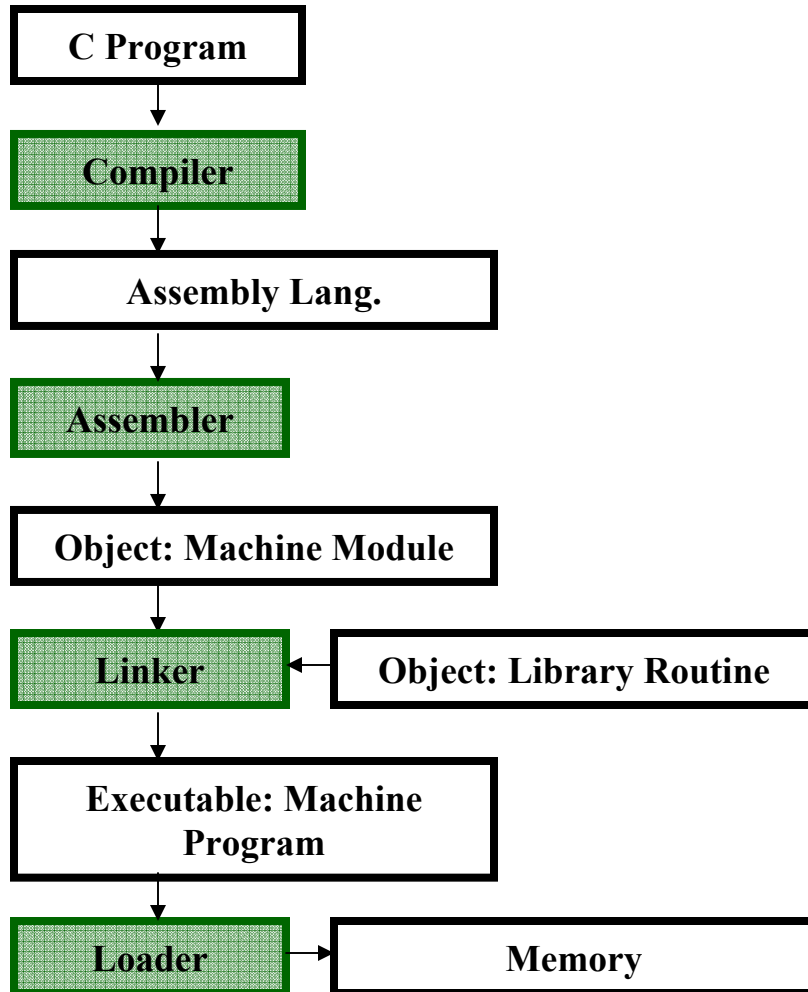
Starting a Program



- **Assembler (Object File)**

- Object file handler: describes the size and position of other pieces
- Text segment: machine language code
- Data segment: contains static/dynamic data associated with program
- Relocation information: identifies instructions and data words that depend on absolute addresses
- Symbol table: remaining undefined labels, external references

Starting a Program



- **Linker**

- Produces executable by combining independently assembled machine language modules and libraries
- Use relocation information and symbol table of each object module to relocate absolute references and resolve all undefined labels

- **Loader**

- Allocate memory and copy text and data into memory
- Copy any parameters to stack, set stack pointer to first free location
- Jump to start-up routine which calls main routine in program

Survey of MIPS Instruction Set

- Arithmetic/Logical
- Data Transfer
- Conditional Branches
- Unconditional Jumps
- Floating-point operations

Arithmetic Operations

- Addition
 - `add $s1, $s2, $s3` # $\$s1 = \$s2 + \$s3$, overflow detected
 - `addu $s1, $s2, $s3` # $\$s1 = \$s2 + \$s3$, overflow undetected
- Subtraction
 - `sub $s1, $s2, $s3` # $\$s1 = \$s2 - \$s3$, overflow detected
 - `subu $s1, $s2, $s3` # $\$s1 = \$s2 - \$s3$, overflow undetected
- Immediate/Constants
 - `addi $s1, $s2, 100` # $\$s1 = \$s2 + 100$, overflow detected
 - `addiu $s1, $s2, 100` # $\$s1 = \$s2 + 100$, overflow undetected

Arithmetic Operations

- Multiply

- mult \$s2, \$s3 # Hi, Lo = \$s2 x \$s3, 64-bit signed product
- multu \$s2, \$s3 # Hi, Lo = \$s2 x \$s3, 64-bit unsigned product

- Divide

- div \$s2, \$s3 # Lo = \$s2/\$s3, Hi = \$s2 mod \$s3
- divu \$s2, \$s3 # Lo = \$s2/\$s3, Hi = \$s2 mod \$s3, unsigned

- Ancillary

- mfhi \$s1 # \$s1 = Hi, get copy of Hi
- mflo \$s1 # \$s1 = Lo, get copy of low

Logical Operations

- Boolean Operations

- and \$s1, \$s2, \$s3 # \$s1 = \$s2 & \$s3, bit-wise AND
- or \$s1, \$s2, \$s3 # \$s1 = \$s2 | \$s3, bit-wise OR

- Immediate/Constants

- andi \$s1, \$s2, 100 # \$s1 = \$s2 & 100, bit-wise AND
- ori \$s1, \$s2, 100 # \$s1 = \$s2 | 100, bit-wise OR

- Shifting

- sll \$s1, \$s2, 10 # \$s1 = \$s2 << 10, shift left
- srl \$s1, \$s2, 10 # \$s1 = \$s2 >> 10, shift right

Data Transfer Operations

- Load Operations

- `lw $s1, 100($s2)` # $\$s1 = \text{Mem}(\$s2+100)$, load word
- `lbu $s1, 100($s2)` # $\$s1 = \text{Mem}(\$s2+100)$, load byte
- `lui $s1, 100` # $\$s1 = 100 * 2^{16}$, load upper imm.

- Store Operations

- `sw $s1, 100($s2)` # $\text{Mem}[\$s2+100] = \$s1$, store word
- `sb $s1, 100($s2)` # $\text{Mem}[\$s2+100] = \$s1$, store byte

Conditional Branches

- Branch Operations

- `beq $s1, $s2, 25` # if($\$s1 == \$s2$), go to PC+4+100, PC-relative
- `bne $s1, $s2, 25` # if($\$s1 \neq \$s2$), go to PC+4+100, PC-relative

- Comparison Operations

- `slt $s1, $s2, $s3` # if($\$s2 < \$s3$) $\$s1 = 1$, else $\$s1 = 0$, 2's comp.
- `slti $s1, $s2, 100` # if($\$s2 < 100$), $\$s1 = 1$, else $\$s1 = 0$, 2's comp.
- `sltu $s1, $s2, $s3` # if($\$s2 < \$s3$), $\$s1 = 1$, else $\$s1 = 0$, unsigned
- `sltiu $s1, $s2, 100` # if($\$s2 < 100$) $\$s1 = 1$, else $\$s1 = 0$, unsigned

Unconditional Jumps

- Jump Operations

- `j 2500` # go to 10000
- `jal 2500` # $\$ra = PC+4$, go to 10000, for procedure call
- `jr $ra` # go to $\$ra$, for procedure return

Floating-Point Operations

- Arithmetic
 - {add.s, sub.s, mul.s, div.s} \$f2, \$f4, \$f6 # single-precision
 - {add.d, sub.d, mul.d, div.d} \$f2, \$f4, \$f6 # double-precision
 - Floating-point registers are used in even-odd pairs, using even number register as its name
- Data Transfer
 - {lwc1, swc1} \$f1, 100(\$s2)
 - Transfer data to/from floating-point register file
- Conditional Branch
 - {c.lt.s, c.lt.d} \$f2, \$f4 # if ($\$f2 < \$f4$), cond=1, else cond=0
 - {bc1t, bc1f} 25 # if (cond==1/0), go to PC+4+100

Summary

- The ISA defines:
 - Operations that the processor can execute
 - Data Transfer mechanisms + how to access data
 - Control Mechanisms (branch, jump, etc)
 - “Contract” between programmer/compiler + HW
- Basics of assembly programming
- Encoding/Decoding assembly into machine code
- Implementing procedures
- Addressing modes
- Survey of MIPS instruction set