MIPS

Instruction Set Architecture

Introduction

- Hardware
  - Voltages
  - Logic Gates
  - Latches
  - Flip-Flops
  - Registers
  - Adders
  - Arithmetic Logic Unit

Introduction

- Hardware Languages
  - Instruction Sets
  - MIPS Assembly Language
  - Machine Code
    - Binary
    - Hexadecimal

Instruction Set

- Instructions: the words of a computer language
- Instruction set: vocabulary
  - Repertoire of instructions of a computer

- Instruction sets may differ from computer to computer, but have many things in common.
  - Computational operations
  - Memory access & addressing
  - Branches
  - Procedure calls

MIPS Instruction Set

- All instruction sets promote a common goal.
    - Make it easy to build the hardware and the compiler while maximizing performance and minimizing cost.

- MIPS is a popular instruction set and still has a share of the embedded core market

Arithmetic Operations

- Addition

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

$a = b + c$

Arithmetic Operations

- Each instruction performs only one operation.
- Each instruction must have three variables.
  - Two sources and one destination.

Arithmetic Operations

- a = b + c + d + e

```
add     a, b, c         #a = b + c
add     a, a, d         #a = b + c + d
add     a, a, e         #a = b + c + d + e
```

Arithmetic Operations

- Each line can only have one instruction on it
- Anything following a # is a comment.

Arithmetic Operations

- From C to MIPS

```
a = b + c;
d = a – e;
```

```
add     a, b, c
sub     d, a, e
```

Arithmetic Operations

- From C to MIPS

```
f = (g + h) – (i + j);

add     t0, g, h        #temporary variable t0 = g + h
add     t1, i, j        #temporary variable t1 = i + j
sub     f, t0, t1       #f = t0 – t1
```

Operands

- Instructions use register operands
- Registers
  - Made from Flip-Flops
  - Primitive used in hardware design
- Register Size
  - 32 bits
  - Called a “word”
- Number of Registers
  - 32 registers in the register file
  - Use for frequently accessed data
  - Numbered 0 to 31

Operands

- MIPS naming conventions
  - $00 - $31
  - $XX
    - A $ followed by two characters that represent the register
- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables

Arithmetic Operations

- From C to MIPS

```
f = (g + h) – (i + j);

f       g       h       i       j
$s0     $s1     $s2     $s3     $s4

add     $t0, $s1, $s2 #temporary variable t0 = g + h
add     $t1, $s3, $s4 #temporary variable t1 = i + j
sub     $s0, $t0, $t1  #f = t0 – t1
```

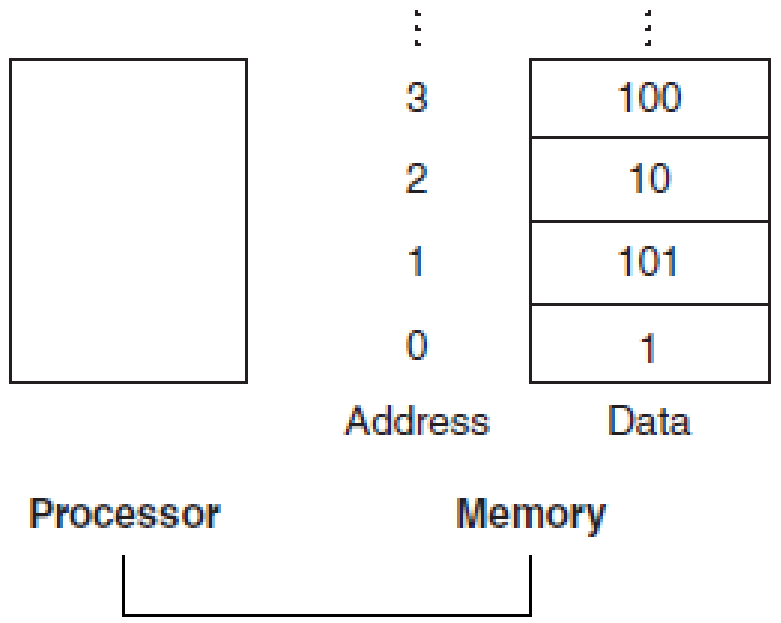
Memory Operands

- Simple variables
  - Integers, Characters, etc.
- Data Structures
  - Arrays, Structures

- Store data structures in main memory

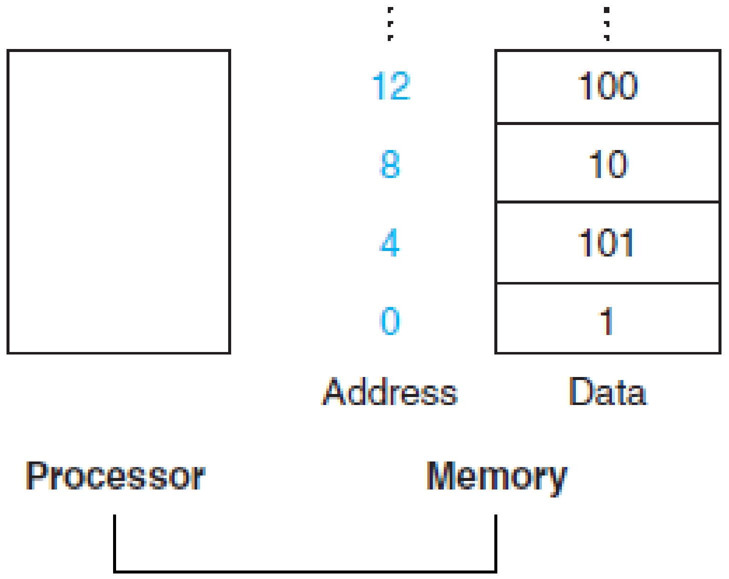
Memory Operands

- To use data from main memory:
  - Load values from memory into registers
  - Store result from register to memory



Memory Operands

- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4



Memory Operands

- load
  - Copies data from memory to a register
- store
  - Copies data from a register to memory

Memory Operands

```
lw      destination, constant(register)
```

The “load word” instruction takes the sum of the constant and the register to determine a memory address. The data at this address is placed in the destination register.

Memory Operands

- From C to MIPS
  - A is an array of 100 words
  - g is a variable in $s1
  - h is a variables in $s2
  - base address of A is in $s3

```
g = h + A[8];
```

Offset
Base Register

- First, we have to transfer A[8] to a register.
- A[8] is stored in memory address $s3 + 8*4

```
lw      $t0, 32($s3)    # t0 = A[8]
add     $s1, $s2, $t0   # g = h + A[8]
```

Memory Operands

```
sw      data, constant(register)
```

The “store word” instruction takes the sum of the constant and the register to determine a memory address. The data in the first register operand will be placed at this address.

Memory Operands

- From C to MIPS
  - A is an array of 100 words
  - h is a variables in $s2
  - base address of A is in $s3

```
A[12] = h + A[8]
```

```
lw      $t0, 32($s3)   # t0 = A[8]
add     $t0, $s2, $t0  # t0 = h + A[8]
sw      $t0, 48($s3)   # store t0 in A[12]
```

Constant Operands

- Sometimes we need to use a constant value
- So far, we would need to load the constant into a register
  - Requires two instructions: load word and add

- Instead, the “add immediate” instruction allows us to use a constant instead of one of the register operands

Constant Operands

- From C to MIPS

```
s = s + 4

addi   $s0, $s0, 4
```

Representing Instructions

- We know that computers use binary to represent data

- Register names are also mapped to numbers

08	$t0	16	$s0
09	$t1	17	$s1
10	$t2	18	$s2
11	$t3	19	$s3
12	$t4	20	$s4
13	$t5	21	$s5
14	$t6	22	$s6
15	$t7	23	$s7

Representing Instructions

- C Programming Language
  - fruit = num_apples + num_oranges;

- MIPS Assembly
  - add	$s0, $s1, $s2

- Machine Code
  - 0000 0010 0011 0010 1000 0000 0010 0000

Representing Instructions

- Instruction Format
  - The layout of an instruction
  - formed by pieces of the instruction called fields

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Representing Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

Representing Instructions

add $t0, $s1, $s2

Represented using decimal numbers in each field:

0	17	18	8	0	32

Convert each field to binary:

000000	10001	10010	01000	00000	100000

Result:

$00000010001100100100000000100000_2$

Representing Instructions

- Hexadecimal
  - Instead of writing strings of 32-bit binary numbers, we can use hexadecimal – a base that converts easily into binary.
  - Hexadecimal is base 16
    - Uses digits 0-9, A-F
    - Replaces a group of four binary numbers with a single hexadecimal digit

0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

Representing Instructions

add $t0, $s1, $s2

$00000010001100100100000000100000_2$

0000	0010	0011	0010	0100	0000	0010	0000
0	2	3	2	4	0	2	0

0x02324020

Representing Instructions

- Convert from hexadecimal to binary:

0xECA86420

E	C	A	8	6	4	2	0
1110	1100	1010	1000	0110	0100	0010	0000

Instruction Formats

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- What if we need longer fields?
  - The load word instruction specifies two registers and a constant.
  - With our current fields, the constant is restricted to 5 or 6 bits

- Same size for all instructions
  vs. Same format for all instructions

Instruction Formats

- R-type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- I-type

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- J-type

op	address
6 bits	26 bits

Instruction Formats

- I-type

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- A 16 bit address allows:
  - lw and sw to access a range of 8192 words
  - addi to add constants in a range of +/- 2^{15}

Instruction Formats

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

```
lw      $t0, 32($s3)
```

Represented using decimal:

```
35      19      8       32
```

Represented in binary:

```
100011   10011   01000   0000000000100000
1000 1110 0110 1000 0000 0000 0010 0000
```

Represented in hexadecimal:

```
0x8E680020
```

Instruction Formats

- Different formats complicate decoding, but allow 32-bit instructions uniformly
- Keep formats as similar as possible
  - First three fields of R-type and I-type are the same
  - Fourth field of I-type is the size of the last 3 fields of R-type
- The first field (op) determines the type and is the same in all three formats

From C to Machine Code

```
A[300] = h + A[300];

lw      $t0, 1200($t1)        #$t1 is the base register
add     $t0, $s2, $t0         #$s2 is h
sw      $t0, 1200($t1)
```

```
35      9       8       1200
0       18      8       8       0       32
43      9       8       1200
```

From C to Machine Code

```
A[300] = h + A[300];

lw      $t0, 1200($t1)        #$t1 is the base register
add     $t0, $s2, $t0         #$s2 is h
sw      $t0, 1200($t1)

100011  01001  01000  0000010010110000
000000  10010  01000  01000  00000  100000
101011  01001  01000  0000010010110000
```

Logical Operations

- Shifts

Logical Operation	C Operator	MIPS Instruction
Shift Left	<<	sll
Shift Right	>>	srl

Shift Operations

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

Shift Operations

- 0000 0000 0000 00000 000 0000 0000 0000 1001 (9)
- Shift left by four:
- 0000 0000 0000 0000 0000 0000 0000 1001 0000 (144)
- $9 * 2^4 = 144$

Shift Operations

```
sll $t2, $s0, 4          # t2 = s0 << 4
```

- R-type instruction that uses the shamt field:

0	0	16	10	4	0

Logical Operations

- Logical

Logical Operation	C Operator	MIPS Instruction
Bitwise AND	&	and, andi
Bitwise OR	|	or, ori
Bitwise NOT	~	nor

AND

- Bit-by-bit operation that leaves 1 in the result only if both bits of the operands are 1.

```
and    $t0, $t1, $t2
```

\$t2	0000 0000 0000 0000 0000 1101 1100 0000
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	0000 0000 0000 0000 0000 1100 0000 0000

- Use AND to “mask” some bits in a word.

OR

- Bit-by-bit operation that leaves 1 in the result only if either bits of the operands are 1.

or $t0, $t1, $t2

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0011 1101 1100 0000

NOT

- Bit-by-bit operation on only one operand that inverts each bit: changes a 0 to a 1 and 1 to a 0

- NOR
    - NOT OR
    - A NOR 0 = NOT (A OR 0) = NOT A

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten

- Useful for common operations
  - move between registers

    ```
    add $t2, $s1, $zero
    ```
  - NOT

    ```
    nor $t0, $t1, $zero
    ```

NOT

```
nor     $t0, $t1, $zero          #t0 = ~(t1 | 0)
```

$t1	0000	0000	0000	0000	0011	1100	0000	0000
$t0	1111	1111	1111	1111	1100	0011	1111	1111

Logical Operations with Constants

- and immediate
    - andi
- Or immediate
    - ori

Branches

- Computer programs can make decisions
  - If statements, Switch statements

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially

- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;

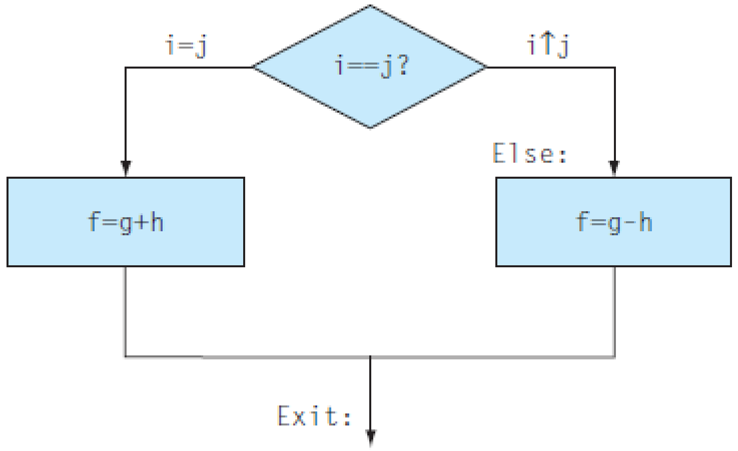
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;

- `j L1`
  - unconditional jump to instruction labeled L1

Branches

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

i	j	f	g	h
$s0	$s1	$s2	$s3	$s4



Branches

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

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

i	$s0
j	$s1
f	$s2
g	$s3
h	$s4

Loops

- Some computer programs require iteration
  - while loops, for loops

Loops

```
while (save[i] == k)
        i +=1;
```

Assume i in $s3, k in $s5, base address of save in $s6.

```
Loop: sll     $t1, $s3, 2
      add     $t1, $t1, $s6
      lw      $t0, 0($t1)
      bne     $t0, $s5, Exit
      addi    $s3, $s3, 1
      j       Loop
Exit:
```

More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with `beq`, `bne`

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```

Loops

```
for (i = 0; i < 10; i++)
        sum += i;
```

Assume i in $s1 and sum in $s3.

```
        addi    $s1, $zero, 0
Loop:   slti    $t0, $s1, 10
        beq     $t0, $zero, Exit
        add     $s3, $s3, $s1
        addi    $s1, $s1, 1
        j       Loop
Exit:
```

Branch Instruction Design

- Why not `blt`, `bge`, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
- `beq` and `bne` are the common case
  - slt, slti, beq, bne can be used to create any other necessary conditions.
- This is a good design compromise

Signed vs. Unsigned

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `slt  $t0, $s0, $s1  # signed`
    - –1 < +1 ⇒ $t0 = 1
  - `sltu $t0, $s0, $s1  # unsigned`
    - +4,294,967,295 > +1 ⇒ $t0 = 0

Switch Statements

```
switch (selection) {
        case 1:
        ….
        case 2:
        …
        default:
}
```

```
if (selection == 1)
        goto 1
else if (selection == 2)
        goto 2
else
        goto 3
```

Switch Statements

```
switch (selection) {
        case 1:
        ….
        case 2:
        …
        default:
}
```

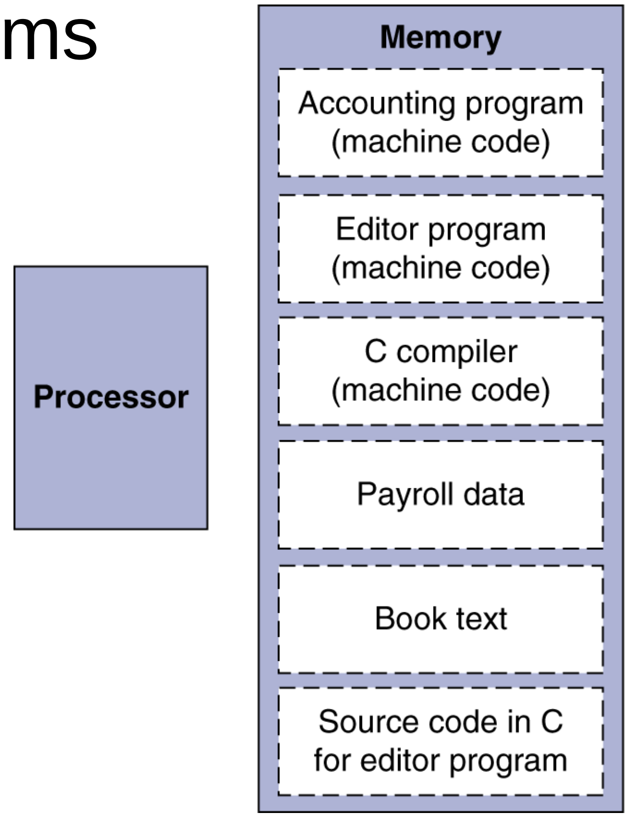
Jump Address Table

- selection is an index to the table
- the table contains addresses

```
jr $s0
```

Stored Program Computers

- Instructions represented in binary
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs


Processor
Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program

Procedures

- We often write functions or methods
  - procedures

- A procedure is stored subroutine that performs a specific task based on the parameters it is provided

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

Registers

- $a0-$a3: four argument registers
- $v0-$v1: two return value registers
- $ra: one return address register

Procedure Call Instructions

- Procedure call: jump and link

```
jal ProcedureLabel
```

  - Address of following instruction put in $ra
    - $ra = PC + 4
  - Jumps to target address

- Procedure return: jump register

```
jr $ra
```

  - Copies $ra to program counter (PC)

Procedures

- The parent program (caller) places parameters in $a0-$a3

- The caller uses jal to jump to the location of the function being called (callee) and store the return address

- The callee completes its task and stores the result in $v0-$v1

- The callee returns control with jr $ra

Stacks

- What if four argument registers and two return value registers aren’t enough?

- Spilling Registers
  - At the beginning of a procedure the contents of $s0-$s7 can be saved in main memory
  - The procedure can then use $s0-$s7 normally
  - At the end, the previous values of $s0-$s7 are retrieved from main memory

- Stacks are a natural structure to allocate dynamic data for procedures

Stacks

- Stacks are a last-in-first-out structure
- Requires a stack pointer to place/remove registers
  - Adjust the pointer by one word when items are added (pushed) or removed (popped)

- MIPS stack pointer is a register: $sp
  - The “bottom” of the stack is the highest address and the stack “grows” from higher to lower addresses.

- Push: decrement $sp
  write to main memory (at $sp)

- Pop: read from main memory (at $sp)
  increment $sp

Leaf Procedure Example

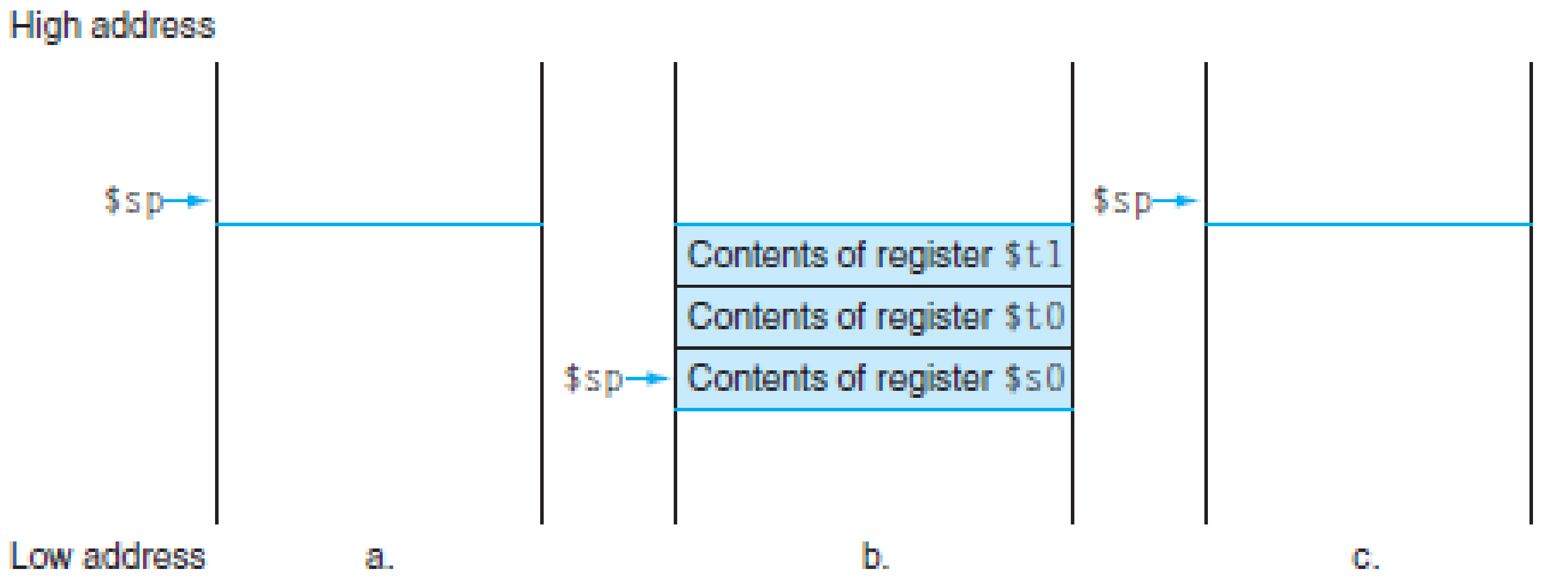
```
int leaf_example (int g, int h, int i, int j) {
        int f;
        f = (g+h) – (i+j);
        return f;
}
```

- Arguments g - j in $a0 - $a3
- f in $s0 (hence, need to save $s0 on stack)
- Result in $v0

Leaf Procedure Example

```
leaf_example:
      addi $sp, $sp, -12
      sw   $t1, 8($sp)
      sw   $t0, 4($sp)
      sw   $s0, 0($sp)
      add  $t0, $a0, $a1
      add  $t1, $a2, $a3
      sub  $s0, $t0, $t1
      add  $v0, $s0, $zero
      lw   $s0, 0($sp)
      lw   $t0, 4($sp)
      lw   $t1, 8($sp)
      addi $sp, $sp, 12
      jr   $ra
```

Leaf Procedure Example



- To reduce register spilling, we can choose to not save and restore temporary registers.

Leaf Procedure Example

- `leaf_example:`

```
addi $sp, $sp, -4
sw   $s0, 0($sp)
add  $t0, $a0, $a1
add  $t1, $a2, $a3
sub  $s0, $t0, $t1
add  $v0, $s0, $zero
lw   $s0, 0($sp)
addi $sp, $sp, 4
jr   $ra
```

Procedures

- Not all procedures are leaf procedures
  - Procedures call other procedures
  - Procedures call copies of themselves (recursion)

- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

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

- Argument n in $a0
- Result in $v0

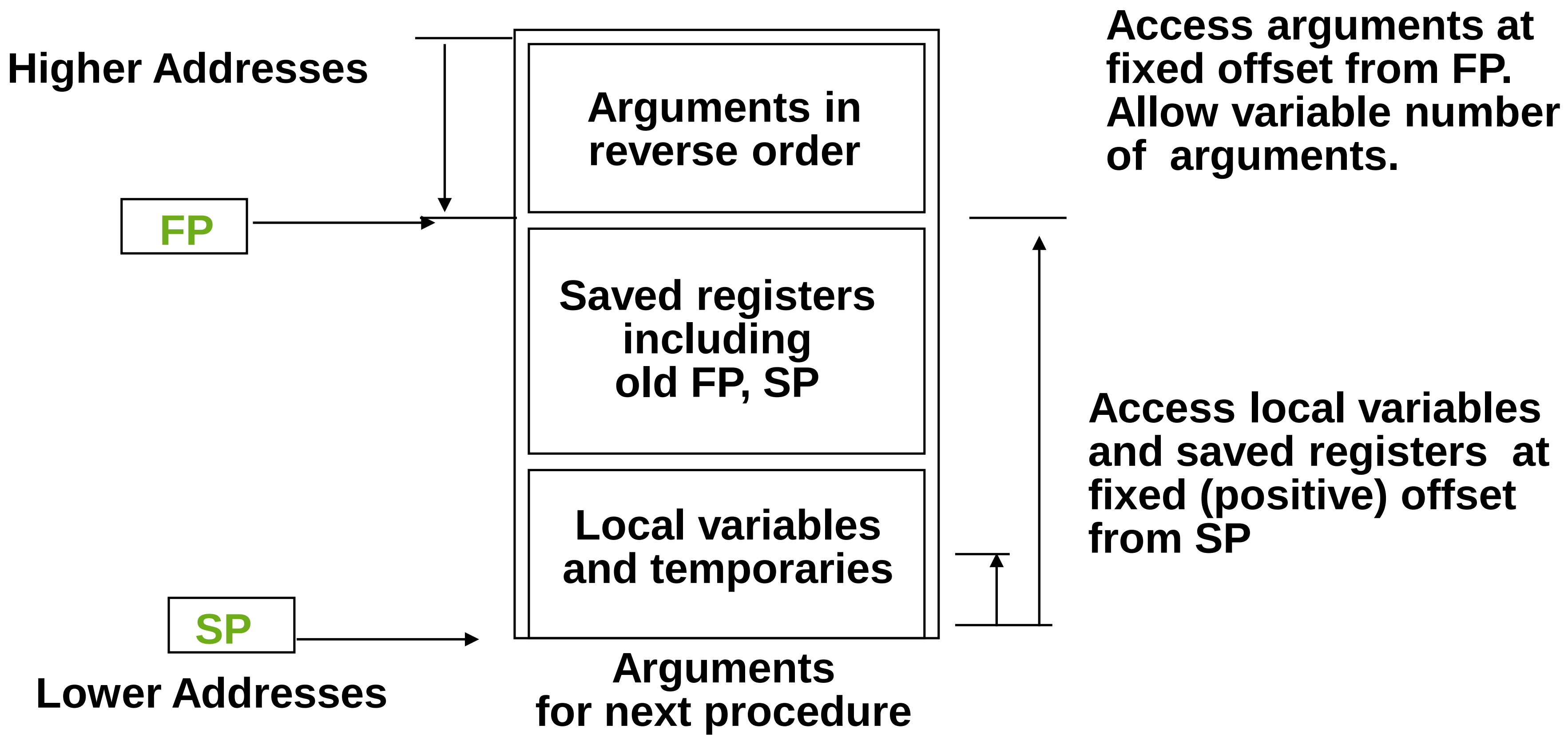
Non-Leaf Procedure Example

```
fact:
    addi $sp, $sp, -8     # adjust stack for 2 items
    sw   $ra, 4($sp)      # save return address
    sw   $a0, 0($sp)      # save argument
    slti $t0, $a0, 1      # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1    # if so, result is 1
    addi $sp, $sp, 8      #   pop 2 items from stack
    jr   $ra              #   and return
L1: addi $a0, $a0, -1     # else decrement n
    jal  fact             # recursive call
    lw   $a0, 0($sp)      # restore original n
    lw   $ra, 4($sp)      #   and return address
    addi $sp, $sp, 8      # pop 2 items from stack
    mult $v0, $a0         # multiply to get result
    mflo $v0              # copy result into vo
    jr   $ra              # and return
```

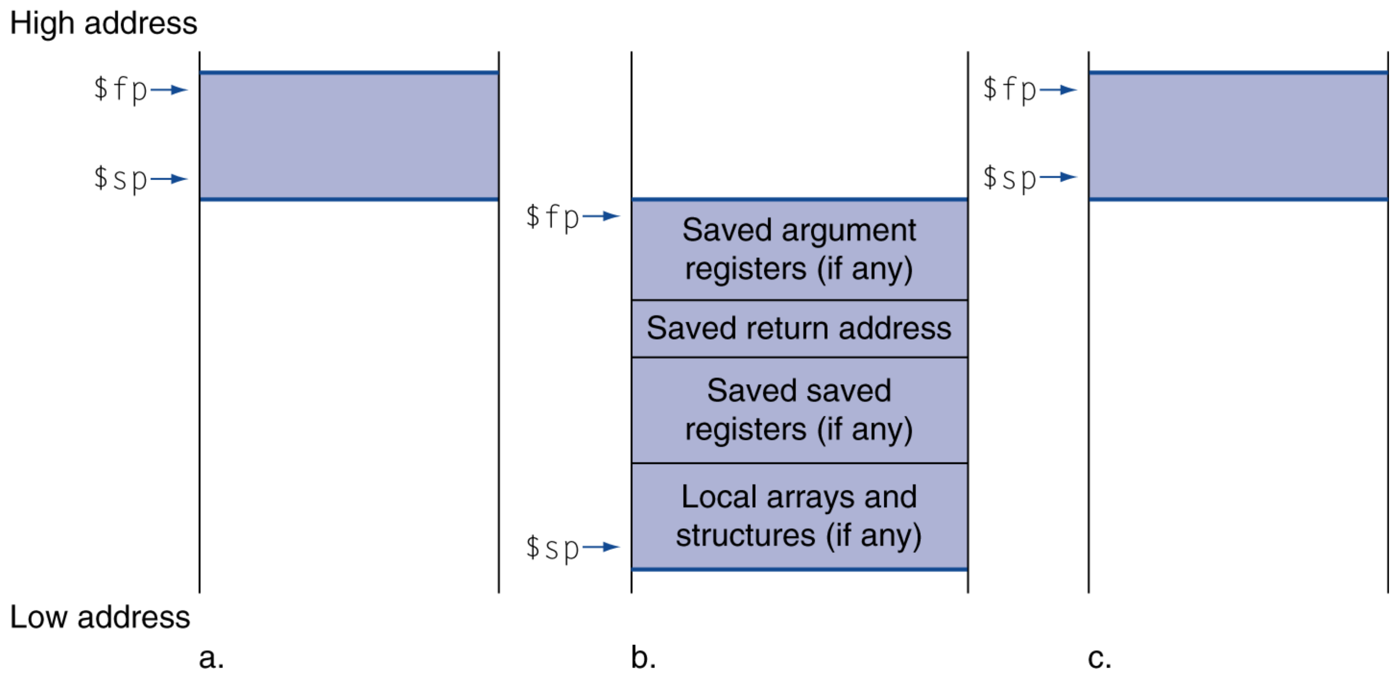
Procedures

Preserved	Not Preserved
Saved registers: $s0-$s7	Temporary registers: $t0-$t9
Stack pointer: $sp	Argument registers: $a0-$a3
Return address: $ra	Return value registers: $v0-$v1
Stack above the stack pointer	Stack below the stack pointer

Stack Frame Layout


Higher Addresses
Arguments in reverse order
Access arguments at fixed offset from FP. Allow variable number of arguments.
FP
Saved registers including old FP, SP
Access local variables and saved registers at fixed (positive) offset from SP
Local variables and temporaries
SP
Lower Addresses
Arguments for next procedure

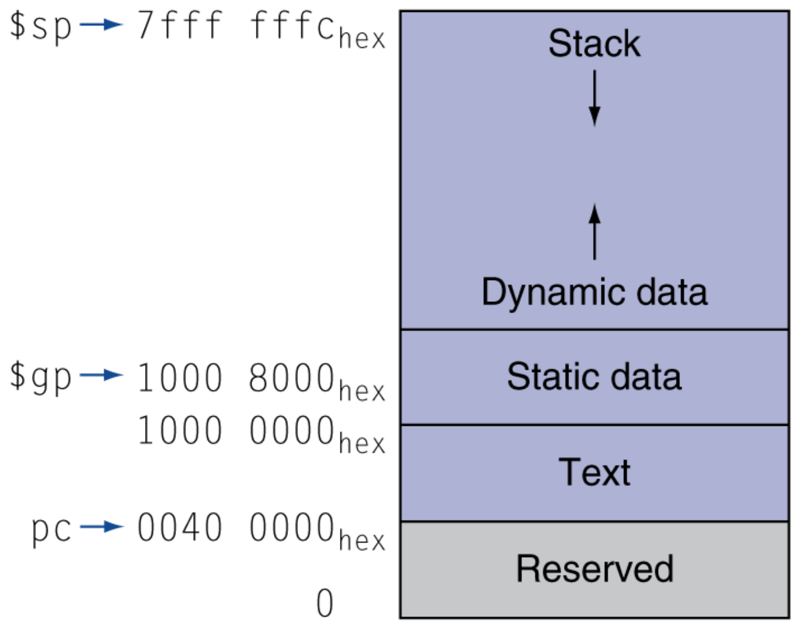
Local Data on the Stack


High address
$fp
$sp
$fp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
$sp
$fp
$sp
Low address
a.
b.
c.

- Local data allocated by callee
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

Memory Layout

- Text: program code
- Static data: global variables
  - static variables in C
  - constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - “malloc” in C
  - “new” in Java
- Stack: automatic storage



MIPS: Software Register Convention

0	$0	zero constant 0
1	$at	reserved for assembler
2	$v0	expression evaluation &
3	$v1	function results
4	$a0	arguments (caller saves)
5	$a1	
6	$a2	
7	$a3	
8	$t0	temporary: caller saves
. . .		
15	$t7	

16	$s0	callee saves
. . .		
23	$s7	
24	$t8	temporary (cont'd)
25	$t9	
26	$k0	reserved for OS kernel
27	$k1	
28	$gp	global pointer
29	$sp	stack pointer
30	$fp	frame pointer
31	$ra	Return Address

MIPS Register Usage and Saving

- The MIPS Architecture Has *General-Purpose Registers*
  - the usage for arguments, stack/data/global pointers, preservation, OS-reserved, etc. is a software convention

MIPS Register Usage and Saving

- Why are registers partitioned into caller and callee save?
- The goal is to minimize the number of registers saved and restored across procedure calls
  - but at the time a procedure is compiled there is generally incomplete information about register usage by the procedure’s caller and those it calls
  - if the convention were purely caller save, then the caller would have to save all registers whose value will be used again, even if the callee is a simple leaf procedure that requires only the argument registers
  - if the convention were purely callee save, then the callee would have to save any register it uses, even if the caller does not need the register’s value
  - so an efficient compromise is a partition of caller/callee save registers

Characters

- Characters are represented in 8-bit bytes
  - American Standard Code for Information Interchange (ASCII)
    - 128 characters: 95 graphic, 33 control
  - Latin-1
    - 256 characters: ASCII characters + 96 more graphic characters
  - Unicode
    - Variable length: 8 bits (UTF-8), 16 bits (UTF-16), 32 bits
    - Contains most of the world’s alphabets, plus symbols

Characters

- To load a character:
  - Use load word to retrieve the correct 32 bit word
  - Use logical instructions to extract the correct byte

- MIPS byte instructions
  - load byte
  - store byte

Byte Operations

- load byte lb rt, offset(rs)
  - Loads a byte from rs+offset, placing it in the rightmost 8 bits of rt
- store byte sb rt, offset(rs)
  - Stores the rightmost 8 bits of rt in rs+offset

```
lb $t0,0($sp)        # Read byte from source
sb $t0,0($gp)        # Write byte to destination
```

Strings

- Representing a string:
  1. the first position of the string is reserved to give the length of a string

     4 C a r t   Java programming language
  2. an accompanying variable has the length of the string

     "Cart"
     4
  3. the last position of a string is indicated by a character used to mark the end of a string.

     C a r t \0   C programming language

String Copy Example

```
void strcpy (char x[], char y[]) {
  int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

- Addresses of x, y in $a0, $a1
- i in $s0

String Copy Example

```
strcpy:
    addi $sp, $sp, -4      # adjust stack for 1 item
    sw   $s0, 0($sp)       # save $s0
    add  $s0, $zero, $zero # i = 0
L1: add  $t1, $s0, $a1     # addr of y[i] in $t1
    lb   $t2, 0($t1)       # $t2 = y[i]
    add  $t3, $s0, $a0     # addr of x[i] in $t3
    sb   $t2, 0($t3)       # x[i] = y[i]
    beq  $t2, $zero, L2    # exit loop if y[i] == 0
    addi $s0, $s0, 1       # i = i + 1
    j    L1                # next iteration of loop
L2: lw   $s0, 0($sp)       # restore saved $s0
    addi $sp, $sp, 4       # pop 1 item from stack
    jr   $ra               # and return
```

String Copy Example

- Since strcpy is a leaf procedure, we could store i in a temporary register
  - Avoids saving and restoring $s0 from the stack

- We can think of temporary registers as registers that the callee should use whenever convenient.

- When a compiler finds a leaf procedure, it exhausts all temporary registers before using registers it must save.

Halfword Operations

- load halfword		lh rt, offset(rs)
  - Loads a halfword from rs+offset, placing it in the rightmost 16 bits of rt
- store halfword		sh rt, offset(rs)
  - Stores the rightmost 16 bits of rt in rs+offset

```
lh $t0,0($sp)          # Read halfword from source
sh $t0,0($gp)          # Write halfword to destination
```

Constants

- Immediate-type instructions have 16 bits for constant values.
  - 50% to 60% of constants fit within 8 bits
  - 75% to 80% of constants fit within 16 bits

- The occasional 32-bit constant must be loaded into a register before it can be used.
  - Set the upper half of the register
    - load upper immediate
  - Set the lower half of the register
    - or immediate

32-bit Constants

0000 0000 0011 1101 0000 1001 0000 0000

1. lui $s0, 61   # 61 = 0000 0000 0011 1101

$s0 = 0000 0000 0011 1101 0000 0000 0000 0000

2. ori $s0, $s0, 2304   # 2304 = 0000 1001 0000 0000

$s0 = 0000 0000 0011 1101 0000 1001 0000 0000

32-bit Constants

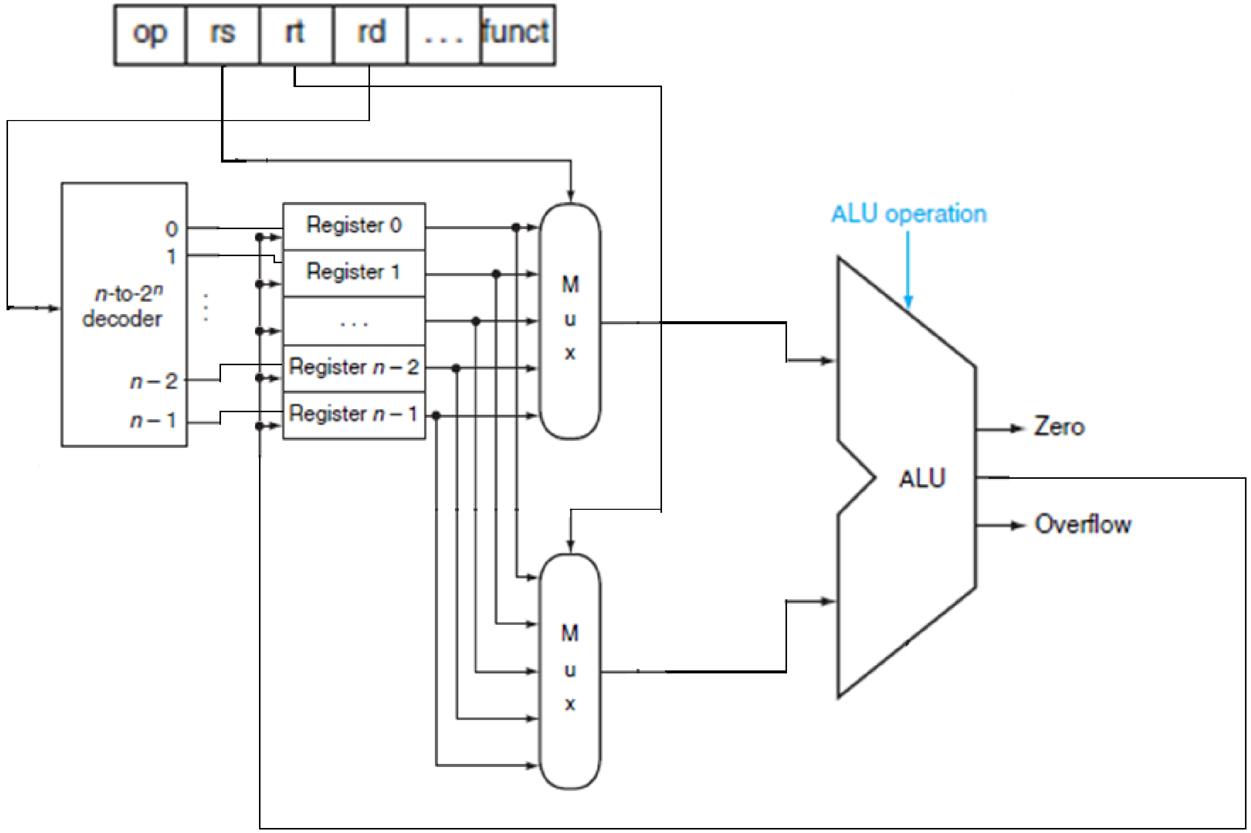
- The MIPS assembler must break larger constants into pieces and reassemble them into a register.
  - This is why there is one register reserved for the assembler: $at.

Addressing Modes

- Register Addressing
- Immediate Addressing
- Base Addressing
- PC-Relative Addressing
- Pseudodirect addressing

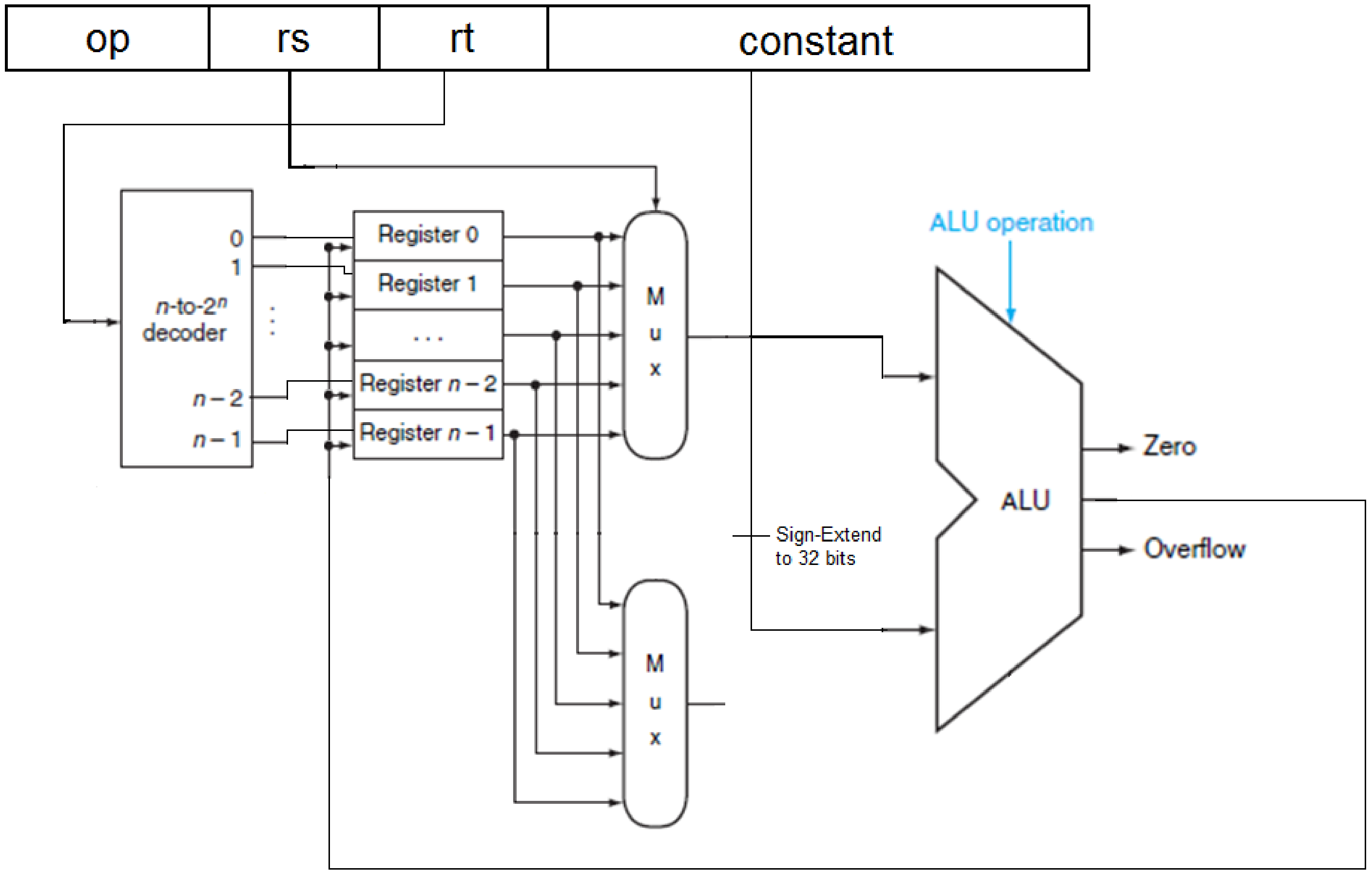
Register Addressing

- Destination and source operands are specified by registers
  - R-Type Instructions



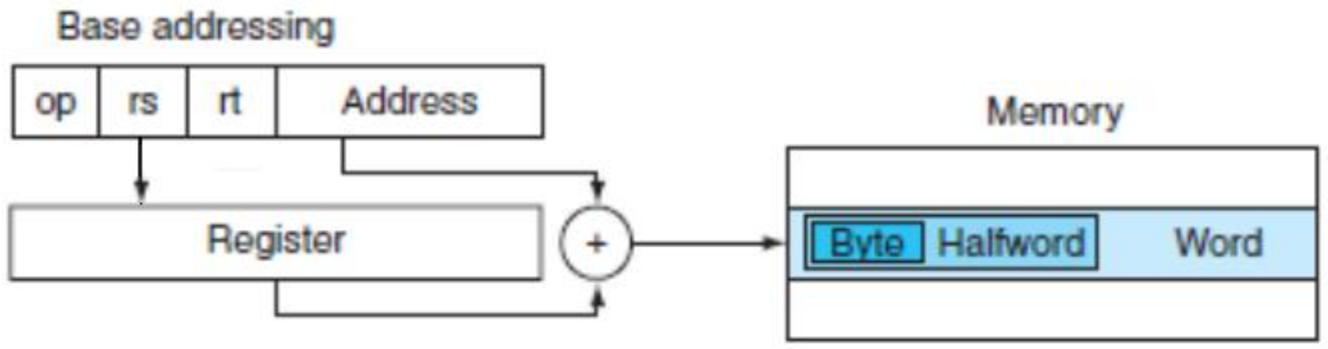
Immediate Addressing

- Source operand is specified by an immediate value.
  - Arithmetic I-Types like addi, andi, ori, and slti



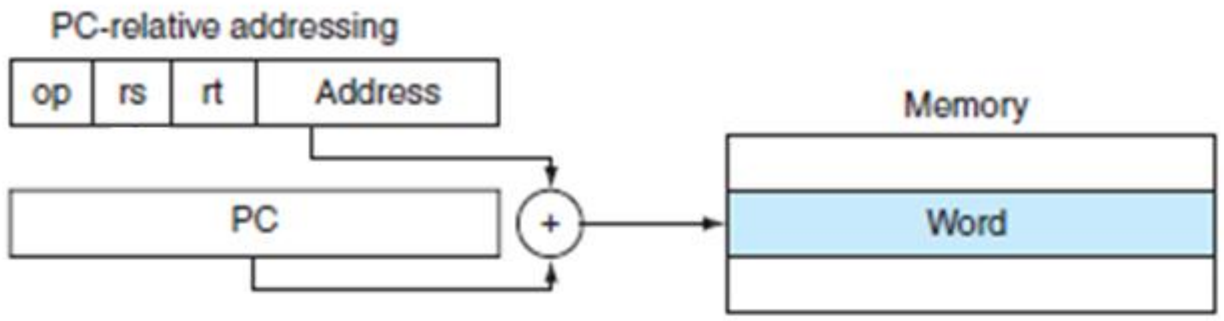
Base Addressing

- Source is determined by register + branch address
  - I-Types lw, lh, lb
- Destination is determined by register + branch address
  - I-Types sw, sh, sb



PC-relative Addressing

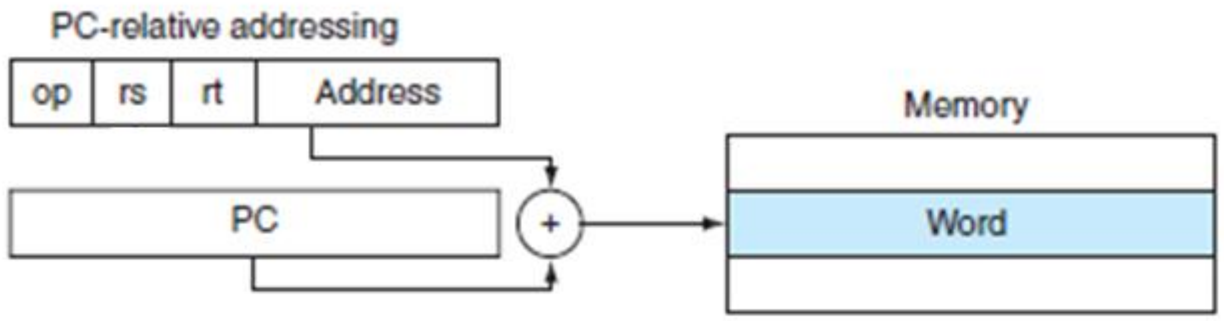
- Destination is determined by PC + Address x 4.
  - I-Types beq and bne



- PC (program counter) is the location of our next address.

PC-relative Addressing

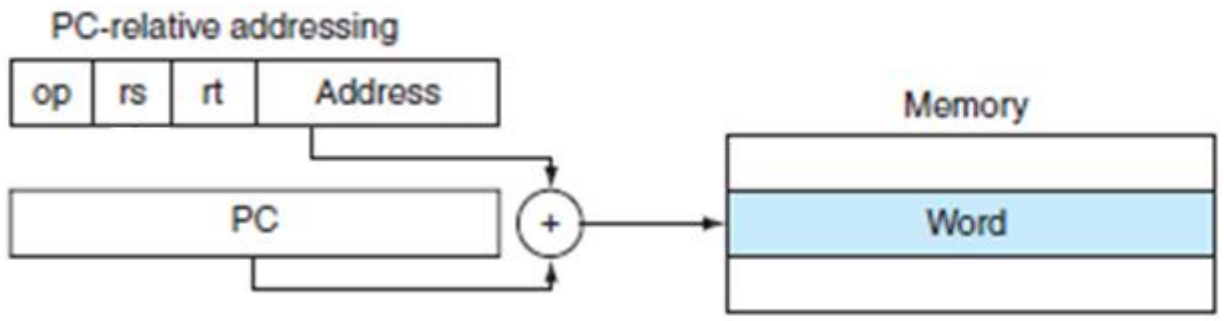
- Destination is determined by PC + Address x 4.
  - I-Types beq and bne



- Address field is 16 bits
- If all addresses had to fit in 16 bits, programs could only be 2^{16} bytes or 16,384 words long.

PC-relative Addressing

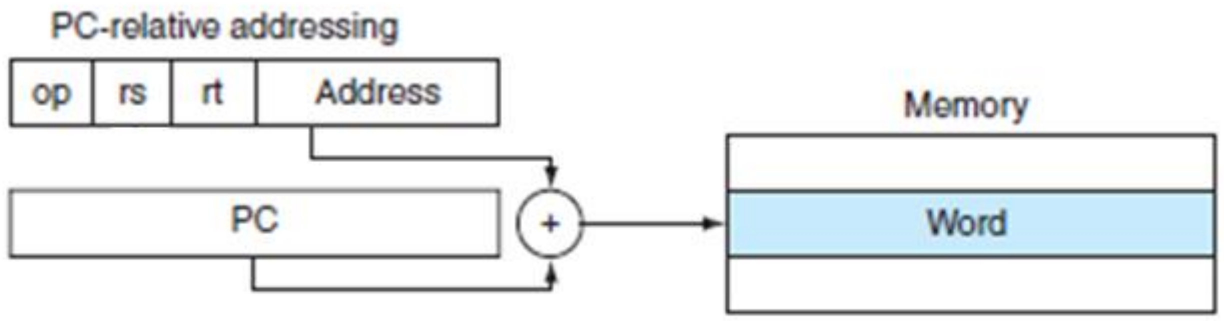
- Destination is determined by PC + Address x 4.
  - I-Types beq and bne



- Conditional branches tend to branch to nearby instructions
  - Some studies suggest half of all branches go less than 16 instructions away.

PC-relative Addressing

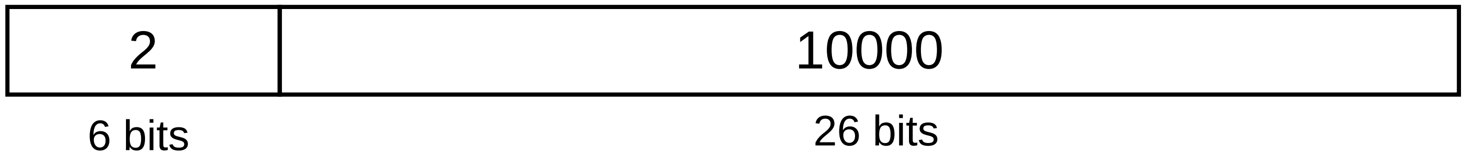
- Destination is determined by PC + Address x 4.
  - I-Types beq and bne



- If we use PC as the register to be added to the address, our branch range will be 2^{15} words in either direction.

Pseudodirect Addressing

- Jump instructions specify a 26 bit address
  - e.g.: j 10000

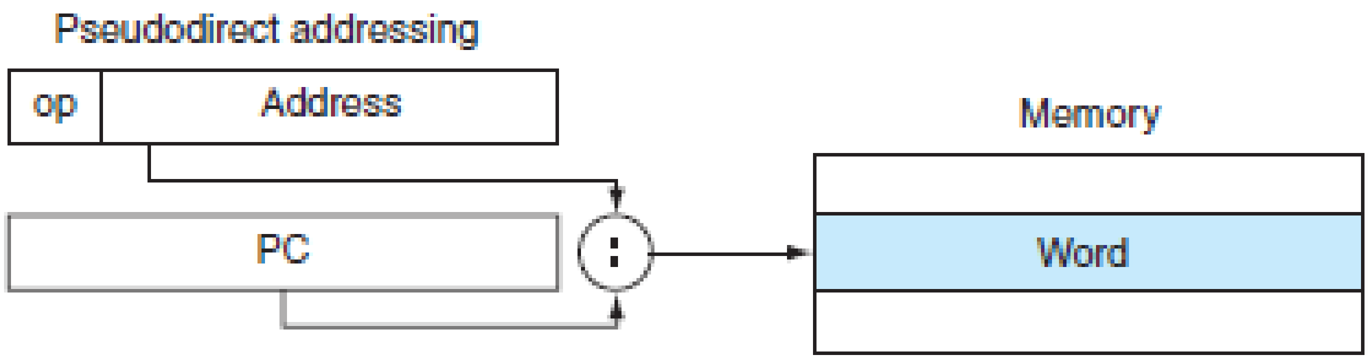


PC = $PC_{31...28}$: (address × 4)

PC = $PC_{31...28}$: 40000

Pseudodirect Addressing

- The 26-bit address field is shifted to the left twice
  - Multiplies by four and creates a 28 bit field
  - Four most significant bits are copied from the current PC



Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

`Loop: sll  $t1, $s3, 2`	80000	0	0	19	9	4	0
`      add  $t1, $t1, $s6`	80004	0	9	22	9	0	32
`      lw   $t0, 0($t1)`	80008	35	9	8	0		
`      bne  $t0, $s5, Exit`	80012	5	8	21	2		
`      addi $s3, $s3, 1`	80016	8	19	19	1		
`      j    Loop`	80020	2	20000				
`Exit: …`	80024						

Decoding Machine Language

- The first six bits are the opcode
- R-type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- I-type

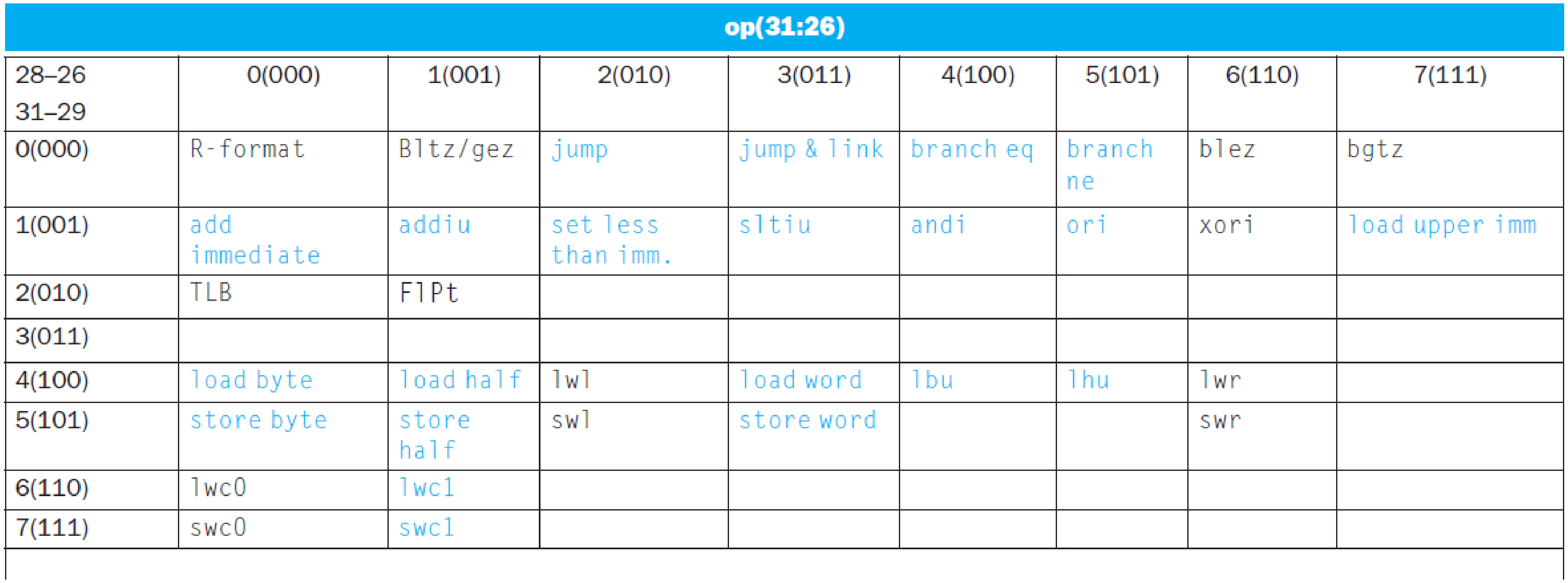
op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- J-type

op	address
6 bits	26 bits

Decoding Machine Language

- Based on the opcode:
  - Bits 31-29 specify row
  - Bits 28-26 specify column



op(31:26)								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	sltiu	andi	ori	xori	load upper imm
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	lbu	lhu	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

Decoding Machine Language

- R-type instructions have an opcode of 000000
  - Instruction is determined by funct field
  - Bits 5-3 specify row
  - Bits 2-0 specify column

op(31:26)=000000 (R-format), funct(5:0)

2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump reg.	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	sltu				
6(110)								
7(111)								

MIPS Integer Arithmetic

- Why doesn’t MIPS have a subtract immediate instruction?
  - Negative constants appear much less frequently in C and Java

  - Since the immediate field holds both negative and positive constants, add immediate with a negative number is equivalent to subtract immediate with a positive number.

Sort Example

```
void sort (int v[], int n) {
        int i, j;
        for (i=0; i<n; i++)
                for (j=i-1; j>= 0 && v[j] > v[j+1]; j--)
                        swap(v, j);
}

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

Translate Swap (Leaf Procedure)

1. Allocate registers to program variables.
2. Produce code for the body of the procedure.
3. Preserve registers across the procedure invocation.

Translate Swap (Leaf Procedure)

1. Allocate registers to program variables.

    void swap(int v[], int k)

    Base address of v in $a0

    k in $a1

    temp in $t0

Translate Swap (Leaf Procedure)

2. Produce code for the body of the procedure.

```
swap:
sll $t1, $a1, 2   # $t1 = k * 4
add $t1, $a0, $t1 # $t1 = v+(k*4)
                  # (address of v[k])
lw $t0, 0($t1)    # $t0 (temp) = v[k]
lw $t2, 4($t1)    # $t2 = v[k+1]
sw $t2, 0($t1)    # v[k] = $t2 (v[k+1])
sw $t0, 4($t1)    # v[k+1] = $t0 (temp)
jr $ra            # return to calling routine
```

Translate sort

```
void sort (int v[], int n) {
        int i, j;
        for (i=0; i<n; i++)
                for (j=i-1; j>= 0 && v[j] > v[j+1]; j--)
                        swap(v, j);
}
```

Translate sort – Outer for loop

```
for (i=0; i<n; i++)



add $s0, $zero, $zero       #initialize i
for1:
slt $t0, $s0, $a1           #reg $t0 = 0 if $s0 >= $a1 (i>=n)
beq $t0, $zero,exit1        #go to exit1 if $s0>=$a1 (i>=n)
…
j for1                   #jump to test of outer loop
exit1:
```

Translate sort – Inner for loop

```
for (j=i-1; j>= 0 && v[j] > v[j+1]; j--)

addi $s1, $s0, –1              # j = i – 1
for2:
slti $t0, $s1, 0               # reg $t0 = 1 if $s1 < 0 (j<0)
bne $t0, $zero,exit2           # go to exit2 if $s1<0 (j<0)
sll $t1, $s1,2                 # reg $t1 = j * 4
add $t2, $a0,$t1               # reg $t2 = v + (j * 4)
lw $t3, 0($t2)                 # reg $t3 = v[j]
lw $t4, 4($t2)                 # reg $t4 = v[j + 1]
slt $t0, $t4, $t3              # reg $t0 = 0 if $t4 >= $t3
beq $t0, $zero,exit2           # go to exit2 if $t4 >= $t3
...
addi $s1, $s1, –1              # j -= 1
j for2                         # jump to test of inner loop
exit2:
```

Translate sort – Calling Swap

- Preserve contents of $a0 and $a1
  - If we have unused registers, storing $a0 and $a1 there will be faster than storing on the stack

```
add $s2, $a0, $zero        # copy parameter $a0 into $s2
add $s3, $a1, $zero        # copy parameter $a1 into $s3
add $a0, $s2, $zero        # first swap parameter is v
add $a1, $s1, $zero        # second swap parameter is j
jal swap
```

Translate sort – before the loops

- Save registers when sort begins

```
addi $sp,$sp,–20        # make room on stack for 5 regs
sw $ra,16($sp)          # save $ra on stack
sw $s3,12($sp)          # save $s3 on stack
sw $s2, 8($sp)          # save $s2 on stack
sw $s1, 4($sp)          # save $s1 on stack
sw $s0, 0($sp)          # save $s0 on stack
```

Translate sort – after the loops

- Restore registers when sort ends

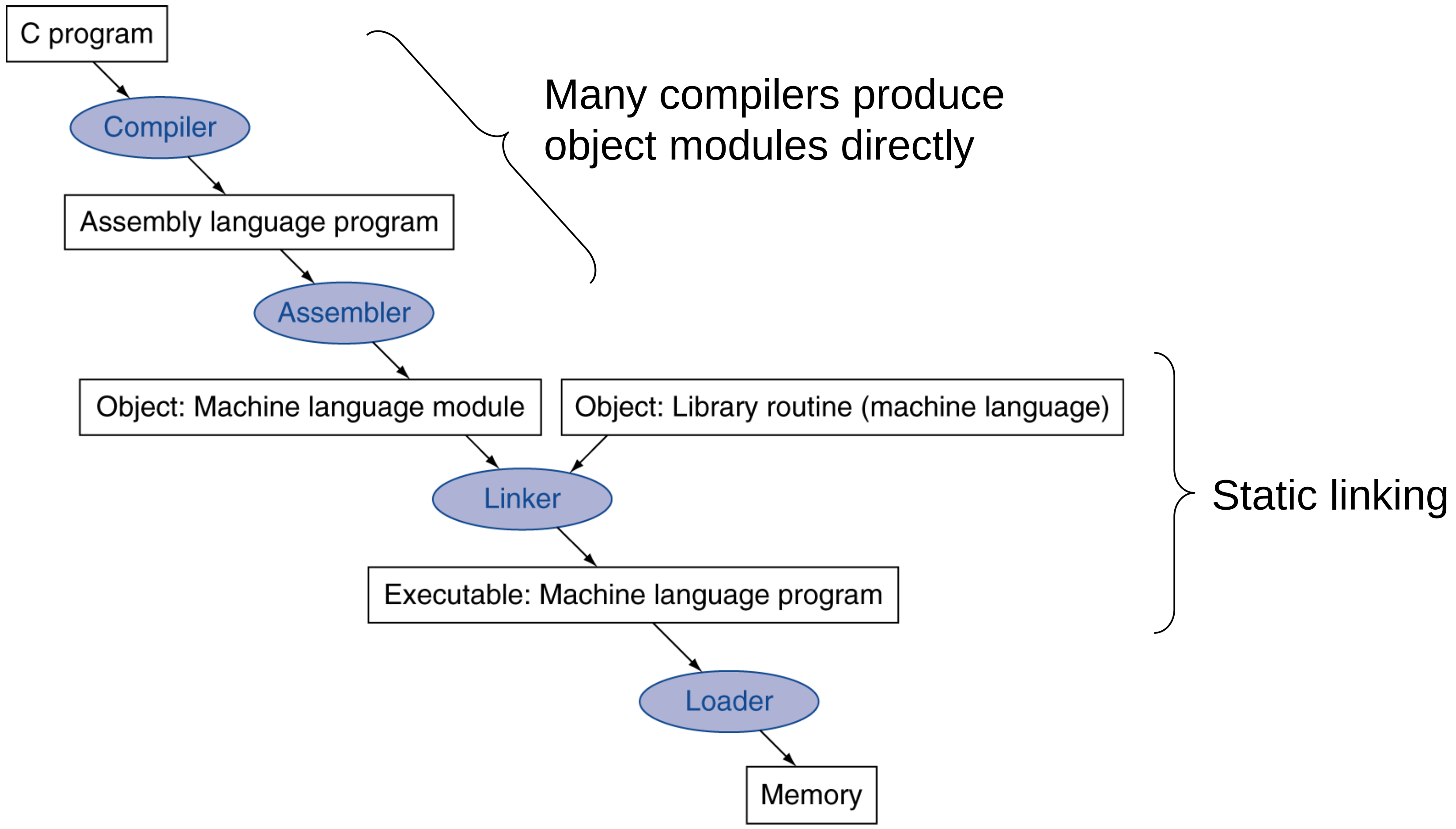
```
lw $s0, 0($sp)          # restore $s0 from stack
lw $s1, 4($sp)          # restore $s1 from stack
lw $s2, 8($sp)          # restore $s2 from stack
lw $s3,12($sp)          # restore $s3 from stack
lw $ra,16($sp)          # restore $ra from stack
addi $sp,$sp, 20        # restore stack pointer
```

Translate Sort

```
sort:
addi $sp,$sp, –20            # make room on stack for 5 registers
sw $ra, 16($sp)              # save $ra on stack
sw $s3,12($sp)               # save $s3 on stack
sw $s2, 8($sp)               # save $s2 on stack
sw $s1, 4($sp)               # save $s1 on stack
sw $s0, 0($sp)               # save $s0 on stack

add $s2, $a0, $zero          # copy parameter $a0 into $s2
add $s3, $a1, $zero          # copy parameter $a1 into $s3

add $s0, $zero, $zero        # i = s0 = 0
for1:
slt $t0, $s0, $a1            # reg $t0 = 0 if $s0 >= $a1 (i>=n)
beq $t0, $zero,exit1         # go to exit1 if $s0□  $a1 (i>=n)

addi $s1, $s0, –1            # j = i – 1
for2:
slti $t0, $s1, 0             # reg $t0 = 1 if $s1 < 0 (j<0)
bne $t0, $zero,exit2         # go to exit2 if $s1<0 (j<0)
sll $t1, $s1,2               # reg $t1 = j * 4
add $t2, $a0,$t1             # reg $t2 = v + (j * 4)
lw $t3, 0($t2)               # reg $t3 = v[j]
lw $t4, 4($t2)               # reg $t4 = v[j + 1]
slt $t0, $t4, $t3            # reg $t0 = 0 if $t4 >= $t3
beq $t0, $zero,exit2         # go to exit2 if $t4 >= $t3
```

```
add $a0, $s2, $zero          # first swap parameter is v
add $a1, $s1, $zero          # second swap parameter is j

jal swap

addi $s1, $s1, –1            # j -= 1
j for2                       # jump to test of inner loop

exit2:
addi $s0, $s0, 1             # i += 1
j for1                       # jump to test of outer loop

exit1:
lw $s0, 0($sp)               # restore $s0 from stack
lw $s1, 4($sp)               # restore $s1 from stack
lw $s2, 8($sp)               # restore $s2 from stack
lw $s3,12($sp)               # restore $s3 from stack
lw $ra,16($sp)               # restore $ra from stack
addi $sp,$sp, 20             # restore stack pointer
jr $ra                       # return to calling routine
```

Translation and Startup


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
Many compilers produce object modules directly
Static linking

Step 1: Compiling

- A compiler transforms a high-level language into assembly instructions and pseudoinstructions
  - High level languages use fewer lines of code
  - Modern programmer productivity is higher
  - Modern compilers are better at optimizing assembly language

Assembler Pseudoinstructions

- Pseudoinstructions: common variation of assembly language instructions

```
move $t0, $t1        →  add $t0, $zero, $t1

blt $t0, $t1, L      →  slt $at, $t0, $t1
                        bne $at, $zero, L
```

Step 2: Assembly

- resolve labels on instructions and data:
  - relative to PC for instructions
  - relative to some register for data
  - either two-pass or use backpatch to resolve external references and PC-relative spans
- expand any macros and pseudoinstructions
- handle any assembler directives: data layout
- translate instructions to binary
- create object file:
  - headers
  - code segment (called text in Unix)
  - data segment
  - relocation information: instruction/data words to relocate
  - symbol table: unresolved references + visible symbols
  - debugging information

Step 3: Linking

- Standard library routines are often not recompiled
- The linker will use the already compiled version

1. Place code and data modules symbolically in memory.
2. Determine the addresses of data and instruction labels.
3. Patch both the internal and external references.

- The result is an executable file with no unresolved references.

Step 4: Loading

- reads executable
- loads code and data segments
- initializes registers, stack, and arguments
- jumps to program’s start-up routine to initiate execution

Dynamic Linking

- Static Linking Advantage
  - fastest method to call library routines
- Static Linking Disadvantages
  - Routines become part of the executable code
    - Updating is more difficult
    - The whole library must be loaded

- Dynamic linking postpones loading and linking library routines until the program is run.
  - Very slow the first time the routine is called

Summary - MIPS

- 32 general purpose registers
  - Software conventions assign properties to registers
- 32-bit wide instructions
  - 3 formats: R-type, I-type, J-type
  - Machine code: binary or hexadecimal
- Reduced Instruction Set
  - Operations: Arithmetic, Logical, Data Transfer, Conditional

Summary – Design Issues

1. *Simplicity favors regularity.* Regularity motivates many features of the MIPS instruction set: keeping all instructions a single size, always requiring three register operands in arithmetic instructions, and keeping the register fields in the same place in each instruction format.
2. *Smaller is faster.* The desire for speed is the reason that MIPS has 32 registers rather than many more.

Summary – Design Issues

3. *Make the common case fast.* Examples of making the common MIPS case fast include PC-relative addressing for conditional branches and immediate addressing for constant operands.
4. *Good design demands good compromises.* One MIPS example was the compromise between providing for larger addresses and constants in instructions and keeping all instructions the same length.