CSE 431
Computer Architecture
Fall 2015

Chapter 2: Instructions: Language of the Computer

Mary Jane Irwin (www.cse.psu.edu/~mji)

[Adapted from *Computer Organization and Design, 5th Edition*,
Patterson & Hennessy, © 2014, MK]

Lecture Reading and Reminders

- This lecture
  - MIPS ISA review, PH, Chapter 2, Appendix A
- Next lecture
  - MIPS ALU review, Flt pt reps, PH, Chapter 3, Appendix B
- Reminders
  - HW1 posted on Angel, due Sept 3rd by 11:55pm in Angel DropBox
  - Quiz 1 posted on Angel, will close midnight Sept 7th
  - Attend one of the unix + SimpleScalar tutorial sessions in the lab (218 IST) Sept 9th and 10th from 7:30 to 9pm (HW2 will contain a SimpleScalar simulation question)
  - First evening midterm exam scheduled
    - Tuesday, October 6th, 20:15 to 22:15, Location 22 Deike
    - Please let me know ASAP (via email) if you have a conflict

Review: Course Administration

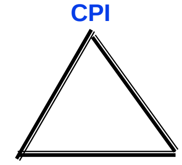
- Instructor: Mary Jane Irwin mji@cse.psu.edu
  348C IST Bldg
  Office Hrs: T 10-11:30am & W 1-2:30pm
- TA: Jing Chen jxc669@psu.edu
  339 IST Bldg (Office Hours)
  Office Hrs: M & F 9-10:30am
- Labs: Accounts on machines in 218 IST (Dells running RedHat Linux)
- URL: Angel
- Text: Required: *Computer Org and Design*, 5rd Ed., Patterson & Hennessy, ©2014
- Slides: Hard copy handed out in class; pdf on Angel after lecture

Review: Grading Information

- Grade determinates
  - First Exam ~27.5%
    - Tuesday, October 6th, 20:15 to 22:15, Location: 22 Deike
  - Second Exam ~27.5%
    - Tuesday, November 17th, 20:15 to 22:15, Location: 22 Deike
  - Homeworks and Final Project (6) ~35%
    - To be submitted on Angel by 23:55 on the due date. No late assignments will be accepted.
  - Class participation & on-line (Angel) quizzes ~10%
- Let me know about exam conflicts ASAP
- Grades will be posted on Angel
  - Must submit email request for change of grade after discussions with the TA (Homeworks/Quizzes) or instructor (Exams)
  - November 30th deadline for filing grade corrections; no requests for grade changes will be accepted after this date

Review: Evaluating ISAs

- Design-time metrics
  - Can it be implemented, at what cost (design, fabrication, test, packaging), with what power, with what reliability?
  - Can it be programmed? Ease of compilation?
- Static Metrics
  - How many bytes does the program occupy in memory?
- Dynamic Metrics
  - How many instructions are executed? How many bytes does the processor fetch to execute the program?
  - How many clocks are required per instruction?
  - How "lean" (fast) a clock is practical?



Best Metric: Time to execute the program!

depends on the instructions set, the processor organization, and compilation techniques.

Below the Program

- High-level language program (in C)

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

one-to-many

C compiler

- Assembly language program (for MIPS)

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

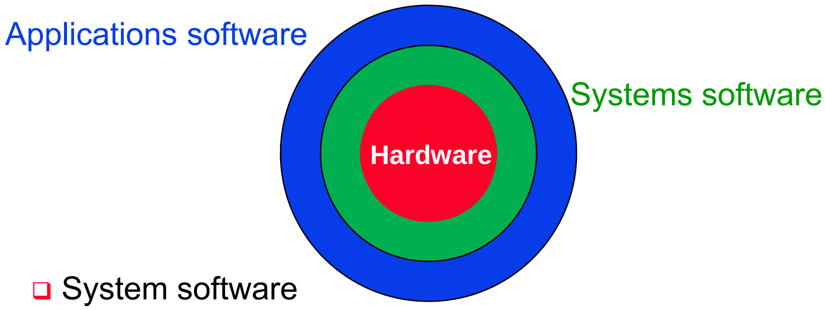
one-to-one

assembler

- Machine (object, binary) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
 . . .
```

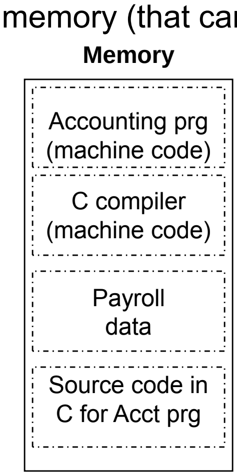
Below the Program, Con't



- System software
  - Compiler – translate programs written in a high-level language (HLL, e.g., C) to machine code – CmpSc 471
  - Operating system – supervising program that interfaces the user's program with the hardware (e.g., Linux, MacOS, Windows) – CmpSc 473
    - Handles basic input and output operations
    - Manages storage (disk) and memory (virtual memory)
    - Schedules tasks and provides for protected sharing of hardware resources (OS memory space)

Two Key Principles of Machine Design

1. Instructions are represented as numbers and, as such, are indistinguishable from data
2. Programs are stored in alterable memory (that can be read or written to) just like data



- Stored-program concept
  - Programs can be shipped as files of binary numbers – binary compatibility
  - Computers can inherit ready-made software provided they are compatible with an existing ISA – this has led the industry to align around a **small** number of ISAs

RISC vs CISC

- RISC = Reduced Instruction Set Computer
  - MIPS, SPARC, PowerPC, ARM (Cortex), etc.
- CISC = Complex Instruction Set Computer
  - X86 is the only surviving example
- Goals in the 1980s – reduce design time, faster/smaller implementation, ISA processor/compiler co-design
- ISAs are measured by how well compilers use them, not by how well or how easily assembly language programmers use them
- There are (or, at least, it's believed there are) many old and useful programs that only exist as machine code, so supporting old ISAs has economic value

MIPS (RISC) Design Principles – Part 1

- Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost
    - fixed size instructions (32-bits (now 64-bits)), small number of instruction formats (three for MIPS), opcode in a fixed location (the first 6 bits for MIPS), etc.
- Smaller is faster
  - Smaller ISA reduces design and implementation costs (and power?), chip sizes, etc.
  - Faster
    - limited instruction set and formats, load-store architecture
      - http://www.arm.com/products/processors/instruction-set-architectures/index.php
    - limited number of registers in the register file (RF)
    - limited number of memory addressing modes
      - Memory address = register value + constant

MIPS-32 ISA

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special

Registers

Register File (RF)
R0 - R31
PC
HI
LO

3 Instruction Formats: all 32 bits wide

op	**rs**	**rt**	**rd**	**sa**	**funct**	R format
op	**rs**	**rt**	**immediate**			I format
op	**jump target**					J format

The Fetch/Execute Cycle

- Memory stores both instruction and data (object code and data bits … just bits)

1. Instruction is fetched from memory at the address indicated by the Program Counter (PC)
2. Control unit decodes the instruction, generates signals to other components so the instruction can be executed
   1. Data is read from the RF or, if necessary, from memory
   2. Datapath executes the instruction as directed by the Control
   3. Data is written to the RF of, if necessary, to memory
3. Control updates the PC which specifies the next instruction to fetch and then execute

MIPS Arithmetic Instructions

- MIPS assembly language arithmetic statement

```
add  $t0, $s1, $s2
sub  $t0, $s1, $s2
```

- Each arithmetic instruction performs one operation
- Each specifies exactly three operands that are all contained in the datapath's RF (`$t0,$s1,$s2`)

  destination ← source1 op source2

- Instruction Format (R format)

hexidecimal (4-bits per hex digit (0 to f)

0	17	18	8	0	0x22
alu	$s1	$s2	$t0	unused	sub

MIPS Instruction Fields

- MIPS fields are given names to make them easier to refer to

op	rs	rt	rd	shamt	funct

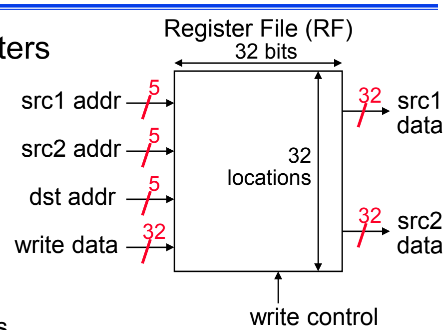
op	6-bits	opcode that specifies the operation
rs	5-bits	register file address of the first source operand
rt	5-bits	register file address of the second source operand
rd	5-bits	register file address of the result's destination
shamt	5-bits	shift amount (for shift instructions)
funct	6-bits	function code augmenting the opcode

MIPS (RISC) Design Principles – Part 2

- Make the common case fast
  - Find the biggest impact on performance
    - E.g., accessing registers is fast, memory is slow
  - Which are the "common cases"? Are they the same for all programs? Will they be the same in the future?
    - arithmetic operands in the RF (load-store machine)
    - allow instructions to contain immediate operands (small constants), otherwise have to bring the constants in from memory, store them in the RF, and access them from there
- Good design demands good compromises
  - Evaluate the many options, determine their impact on performance (IPC?, IC?, clock rate?), make a reasonable choice that doesn't limit future extensions
    - three instruction formats, as similar as possible
    - only two branch instructions (`beq`, `bne`) with a way to do many more with the `slt` "set up" instruction

MIPS Register File (RF)

- Holds thirty-two 32-bit registers
  - Two read ports and
  - One write port



- Registers are
  - Faster than main memory
    - But RFs with more locations are slower (e.g., a 64 word file could be as much as 50% slower than a 32 word file)
    - Increasing number of read/write ports impacts speed quadratically
  - Improves code density (a register is named with fewer bits than a memory location)
  - Easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack

Aside: MIPS Register Convention

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

Number System(s) Review

- Before discussing more instructions, some refreshers
  - Unsigned (binary (positive)) integers
    - Zero-extension (widen an unsigned integer without changing its value) – left extend with zeros
  - Signed (two's complement, binary) integers
    - Sign-extension (widen a signed integer without changing its value)
  - Addresses look like unsigned integers, but some operations on unsigned integers don't make sense for addresses
    - MIPS `add` instruction operates on signed integers
    - MIPS `addu` instruction operates on unsigned integers
    - The only difference is the treatment of overflow
- Later …
  - Floating-point numbers
  - Character data (ASCII)
  - Packed data for instruction-level parallelism

Unsigned Binary Representation

Hex	Binary	Decimal
0x00000000	0…0000	0
0x00000001	0…0001	1
0x00000002	0…0010	2
0x00000003	0…0011	3
0x00000004	0…0100	4
0x00000005	0…0101	5
0x00000006	0…0110	6
0x00000007	0…0111	7
0x00000008	0…1000	8
0x00000009	0…1001	9
	…	
0xFFFFFFFC	1…1100	2^{32} - 4
0xFFFFFFFD	1…1101	2^{32} - 3
0xFFFFFFFE	1…1110	2^{32} - 2
0xFFFFFFFF	1…1111	2^{32} - 1

$X = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \ldots + x_2 2^2 + x_1 2^1 + x_0 2^0$

2^{31} 2^{30} 2^{29} . . . 2^3 2^2 2^1 2^0 bit weight

31 30 29 . . . 3 2 1 0 bit position

1 1 1 . . . 1 1 1 1 bit

1 0 0 0 . . . 0 0 0 0 - 1

2^{32} - 1

With n bits, the range is 0 to (2^n – 1)

Signed Binary Representation

- Two's complement negation
- Sign extend – replicate the sign bit to the left

complement all the bits

0101 1011

and add a 1 and add a 1

0110 1010

complement all the bits

	2'sc binary	decimal
-2^3 =	1000	-8
$-(2^3 - 1)$ =	1001	-7
	1010	-6
	1011	-5
	1100	-4
	1101	-3
	1110	-2
	1111	-1
	0000	0
	0001	1
	0010	2
	0011	3
	0100	4
	0101	5
	0110	6
$2^3 - 1$ =	0111	7

MIPS Memory Access Instructions

- MIPS has two basic data transfer instructions for accessing memory

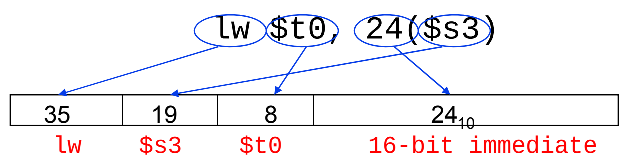
```
lw   $t0, 4($s3)  #load word from memory
sw   $t0, 8($s3)  #store word to memory
```

- The data is loaded into (`lw`) or stored from (`sw`) a register in the register file – a 5 bit address

- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the sign-extended offset value
  - The offset is a 16-bit 2's complement number, so access is limited to memory locations within a region of $\pm 2^{13}$ (8,192) words or $\pm 2^{15}$ (32,768) bytes of the address in the base register

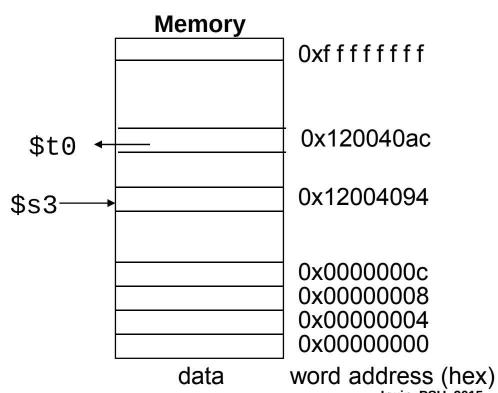
Machine Language - Load Instruction

- Load/Store Instruction Format (I format):



24_{10} + $s3 =

. . . 0001 1000
+ . . . 1001 0100
. . . 1010 1100 =
0x120040ac



Byte Addresses

- Since 8-bit bytes are so useful, most architectures support addressing individual bytes in memory
  - Alignment restriction - the memory address of a word must be on natural word boundaries (a multiple of 4 in MIPS-32)
- Big Endian: leftmost byte is word address

  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: rightmost byte is word address

  Intel 80x86, DEC Vax, DEC Alpha, ARM

	7	**6**	**5**	**4**	
	msb			**lsb**	**Word addr 4**
	4	**5**	**6**	**7**	
	3	**2**	**1**	**0**	***little endian byte 0***
	msb			**lsb**	**Word addr 0**
big endian byte 0	**0**	**1**	**2**	**3**	

Aside: Loading and Storing Bytes

- MIPS provides special instructions to move bytes

```
lb   $t0, 1($s3)  #load byte from memory
sb   $t0, 6($s3)  #store byte to  memory
```

sb	$s3	$t0	6
0x28	19	8	16 bit offset

- What 8 bits get loaded and stored?
  - load byte places the byte from memory in the rightmost 8 bits of the destination register
    - what happens to the other bits in the register?
  - store byte takes the byte from the rightmost 8 bits of a register and writes it to a byte in memory
    - what happens to the other bits in the memory word?

MIPS Immediate Instructions

- Small constants are used often in typical code

- Possible approaches?
  1. put “typical constants” in memory and load them into the RF
  2. create hard-wired registers (like $zero) for constants like 1
  3. have special instructions that contain constants !

```
addi  $sp, $sp, 4     #$sp = $sp + 4
slti $t0, $s2, 15     #$t0 = 1 if $s2<15
                      # otherwise $t0 = 0
```

- Machine format (I format):

slti	$s2	$t0	15
0x0a	18	8	0x0f

- The constant is kept inside the instruction itself!
  - Immediate format limits values to the range $+2^{15}–1$ to -2^{15}

Aside: How About Larger Constants?

- We'd also like to be able to load a 32 bit constant into a register, for this we must use two instructions
- a new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```

16	0	8	1010101010101010_2

- Then must get the lower order bits right, use

```
ori $t0, $t0, 1010101010101010
```

1010101010101010	0000000000000000
0000000000000000	1010101010101010
1010101010101010	1010101010101010

MIPS Shift Operations

- Need operations to pack and unpack 8-bit characters into 32-bit words
- Shifts move all the bits in a word left or right

```
sll $t2, $s0, 8    #$t2 = $s0 << 8 bits
srl $t2, $s0, 8    #$t2 = $s0 >> 8 bits
```

- Instruction Format (R format)

0		16	10	8	0x00
alu	unused	$s0	$t2	shamt	sll

- Such shifts are called logical because they fill with zeros
  - Notice that a 5-bit shamt field is enough to shift a 32-bit value $2^5 - 1$ or 31 bit positions

MIPS Logical Operations

- There are a number of bit-wise logical operations in the MIPS ISA

```
and $t0, $t1, $t2 #$t0 = $t1 & $t2
or  $t0, $t1, $t2 #$t0 = $t1 | $t2
nor $t0, $t1, $t2 #$t0 = not($t1 | $t2)
```

- Instruction Format (R format)

0	9	10	8	0	0x24
alu	$t1	$t2	$t0	unused	and

```
andi $t0, $t1, 0xFF00   #$t0 = $t1 & ff00
ori  $t0, $t1, 0xFF00   #$t0 = $t1 | ff00
```

- Instruction Format (I format)

0x0d	9	8	0xff00

MIPS Control Flow Instructions

- MIPS conditional branch instructions:

```
bne $s0, $s1, Lbl #go to Lbl if $s0≠$s1
beq $s0, $s1, Lbl #go to Lbl if $s0=$s1
```

- Ex:

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

        bne $s0, $s1, Lbl1
        add $s3, $s0, $s1
Lbl1:   ...
```

COMMON CASE FAST

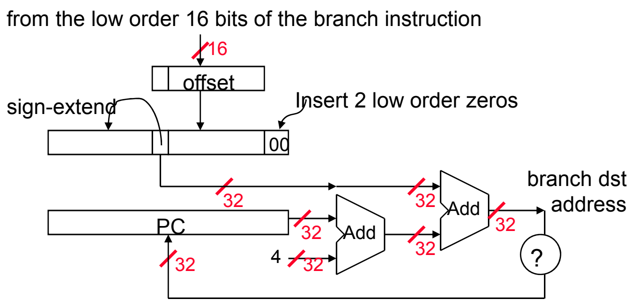
- Instruction Format (I format):

0x05	16	17	16 bit offset
bne	$s0	$s1	16-bit value

- How is the branch destination address specified?

Specifying Branch Destinations

- Use a register (like in lw and sw) added to the 16-bit offset
  - which register? Instruction Address Register (the PC)
    - its use is automatically implied by instruction
    - PC gets updated (PC+4) during the fetch cycle so that it is holding the address of the next instruction when the branch executes



  - limits the branch distance to -2^{15} to $+2^{15}-1$ (word) instructions from the (instruction after the) branch instruction, but most branches are local anyway

In Support of Branch Instructions

- We have beq, bne, but what about other kinds of branches (e.g., branch-if-less-than)? For this, we need yet another instruction, `slt`
- Set on less than instruction:

```
slt $t0, $s0, $s1     # if $s0 < $s1    then
                      # $t0 = 1         else
                      # $t0 = 0
```

- Instruction format (R format):

0	16	17	8		0x2a

- Alternate versions of `slt`

```
slti $t0, $s0, 25     # if $s0 < 25 then $t0=1 ...
sltu $t0, $s0, $s1    # if $s0 < $s1 then $t0=1 ...
sltiu $t0, $s0, 25    # if $s0 < 25 then $t0=1 ...
```


Aside: More Branch Instructions

- Can use `slt`, beq, bne, and the fixed value of 0 in register `$zero` to create other conditions
  - less than `blt $s1, $s2, Label`

```
slt  $at, $s1, $s2       #$at set to 1 if
bne  $at, $zero, Label   #$s1 < $s2
```

  - less than or equal to `ble $s1, $s2, Label`
  - greater than `bgt $s1, $s2, Label`
  - great than or equal to `bge $s1, $s2, Label`

- Such branches are included in the instruction set as pseudo instructions - recognized (and expanded) by the assembler
  - Its why the assembler needs a reserved register (`$at`)

Aside: Branching Far Away

- What if the branch destination is further away than can be captured in 16 bits?

- The assembler comes to the rescue – it inserts an unconditional jump to the branch target and inverts the condition

```
            beq  $s0, $s1, L1
```

becomes

```
            bne  $s0, $s1, L2
            j    L1
      L2:
```


Bounds Check Shortcut

- Treating signed numbers as if they were unsigned gives a low cost way of checking if $0 \leq x < y$ (index out of bounds for arrays)

```
sltu $t0, $s1, $t2      # $t0 = 0 if
                        # $s1 > $t2 (max)
                        # or $s1 < 0 (min)
 beq $t0, $zero, IOOB   # go to IOOB if
                        # $t0 = 0
```

- The key is that negative integers in two's complement look like large numbers in unsigned notation. Thus, an unsigned comparison of $x < y$ also checks if x is negative as well as if x is less than y.

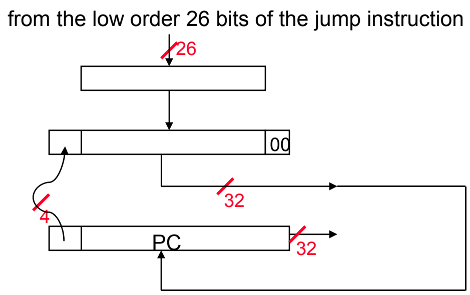
Other Control Flow Instructions

- MIPS also has an unconditional branch instruction or jump instruction:

```
j   label        #go to label
```

- Instruction Format (J Format):

0x02	26-bit address



Instructions for Accessing Procedures

- MIPS procedure call instruction:

```
jal  ProcedureAddress    #jump and link
```

- Saves PC+4 in register $ra to have a link to the next instruction for the procedure return
- Machine format (J format):

0x03	26 bit address

- Then can do procedure return with a jump register instr

```
jr   $ra                 #return
```

- Instruction format (R format):

0	31				0x08

Six Steps in Execution of a Procedure

- Recall the distinction from HHLs
  - parameters – names used when the function is written
  - arguments – values provided when the function is called
- Low-level languages like assembler will associate parameters with registers and memory locations, and arguments with the contents of those registers and memory locations

1. The main routine (caller) evaluates the function argument expressions and places argument values where the procedure (callee) can access them
   - `$a0` - `$a3`: four argument registers
   - Save previous values in those registers if necessary
   - Additional compiler-assigned space on the run-time stack

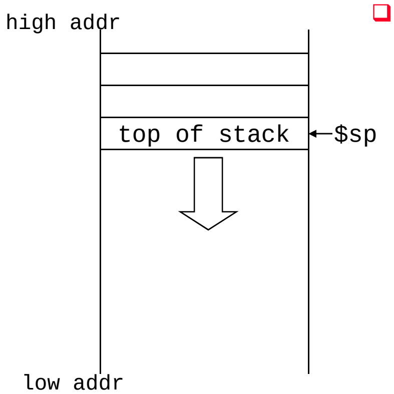
Six Steps in Execution of a Procedure, con't

2. Caller transfers control to the callee
   - `jal` instruction, writes return address (PC + 4) to `$ra`
3. Callee acquires the more storage resources if needed
   - More registers, temporary space on the run-time stack, heap
4. Callee performs the desired task and places the result value in a place where the caller can access it
   - `$v0` - `$v1`: two value registers
   - Additional compiler-assigned space on the run-time stack
5. Callee prepares to return control to the caller
   - Restores previous register values (if necessary), releases temporary space on the run-time stack (adjust `$sp`)
6. Callee returns control to the caller
   - `$jr` instruction using `$ra`

- The caller continues execution after the function call

Aside: Spilling Registers

- What if the callee needs to use more registers than allocated to argument and return values?
  - callee uses a stack – a last-in-first-out queue

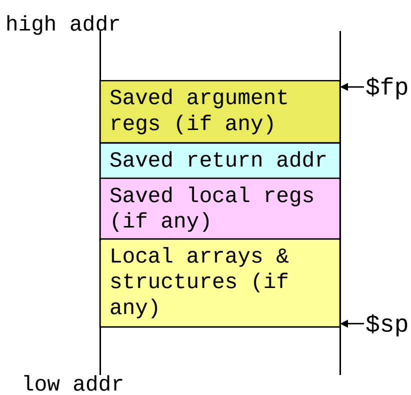


- One of the general registers, $sp ($29), is used to address the stack (which "grows" from high address to low address)
  - add data onto the stack – push

    $sp = $sp – 4
    data on stack at new $sp
  - remove data from the stack – pop

    data from stack at $sp
    $sp = $sp + 4

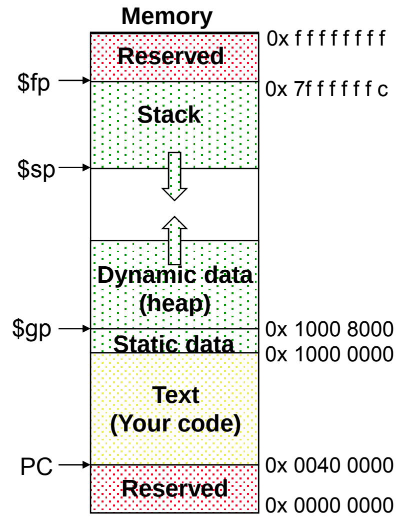
Aside: Allocating Space on the Stack



- The segment of the stack containing a procedure's saved registers and local variables is its procedure frame (aka activation record)
  - The frame pointer ($fp) points to the first word of the frame of a procedure – providing a stable "base" register for the procedure
    - $fp is initialized using $sp on a call and $sp is restored using $fp on a return
    - $fp is unchanged during the procedure's execution
    - $sp could change even without calling another procedure, if we need more space on the stack

Aside: Allocating Space on the Heap

- Static data segment for constants and other staticially-allocated variables (e.g., globally defined arrays)
  - `$gp` = global pointer; never changes
- Dynamic data segment (aka heap) for structures that grow and shrink (e.g., linked lists)
  - In C, allocate space on the heap with `malloc()` and deallocate it with `free()`



For Later: Atomic Exchange Instructions

- Hardware support for synchronization mechanisms
  - Avoid data races where the results of the program can change depending on the relative ordering of events
  - Two memory accesses from different threads/cores to the same memory (cache) location, and at least one is a write – which goes first?

- Atomic exchange (atomic swap, atomic read/write)
  - Interchange a value in a register with a value in memory **atomically**, i.e., as one indivisible operation
  - Logically requires both a memory read and a memory write in a single, uninterruptable instruction. An alternative is to have a pair of specially configured instructions where no other access to the location is allowed between the read and the write.

```
ll   $t1, 0($s1)     #load linked
sc   $t0, 0($s1)     #store conditional
```

For Later: Atomic Exchange with `ll` and `sc`

- If the contents of the memory location specified by the `ll` is changed before the `sc` to the same address occurs, the `sc` fails (returns 0), otherwise it succeeds (returns 1)

```
try:  add $t0, $zero, $s4   #$t0=$s4 (exchange value)
      ll  $t1, 0($s1)       #load memory value to $t1
      sc  $t0, 0($s1)       #try to store exchange
                            #value in $t0 to memory,
                            #if fail then $t0 will be 0,
                            #if ok then $t0 will be 1
      beq $t0, $zero, try   #try again on failure
      add $s4, $zero, $t1   #copy new value in $s4
```

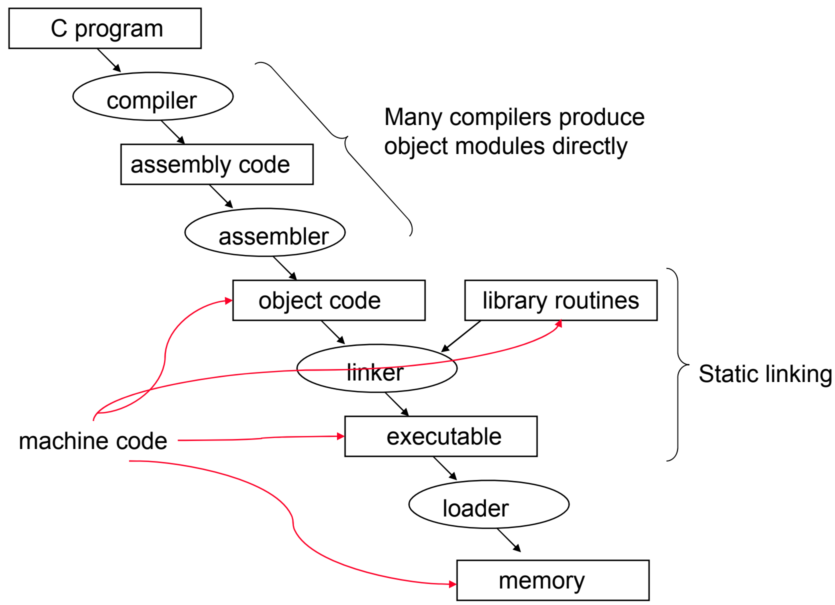
- If the value in memory changes between the `ll` and the `sc` instructions, then `sc` returns a 0 in `$t0` causing the code sequence to try again.

MIPS Instruction Classes Distribution

- Frequency of MIPS instruction classes for SPEC2006

Instruction Class	Frequency	
	SPECint	SPECfp
Arithmetic	16%	48%
Data transfer	35%	36%
Logical	12%	4%
Cond. Branch	34%	8%
Jump	2%	0%

The C Code Translation Hierarchy



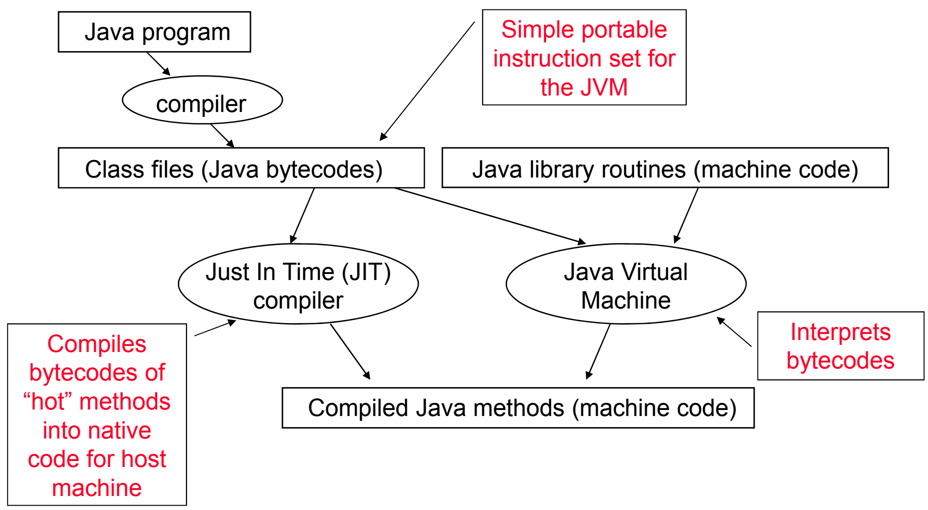
Compiler Benefits

- Comparing performance for bubble (exchange) sort
  - To sort 100,000 words with the array initialized to random values on a Pentium 4 with a 3.06 GHz clock rate, a 533 MHz system bus, with 2 GB of DDR SDRAM, using Linux version 2.4.20

gcc opt	Relative performance	Clock cycles (M)	Instr count (M)	CPI
None	1.00	158,615	114,938	1.38
O1 (medium)	2.37	66,990	37,470	1.79
O2 (full)	2.38	66,521	39,993	1.66
O3 (proc mig)	2.41	65,747	44,993	1.46

- The unoptimized code has the best CPI, the O1 version has the lowest instruction count, but the O3 version is the fastest. Why?

The Java Code Translation Hierarchy



Sorting in C versus Java

- Comparing performance for two sort algorithms in C and Java
  - The JVM/JIT is Sun/Hotspot version 1.3.1/1.3.1

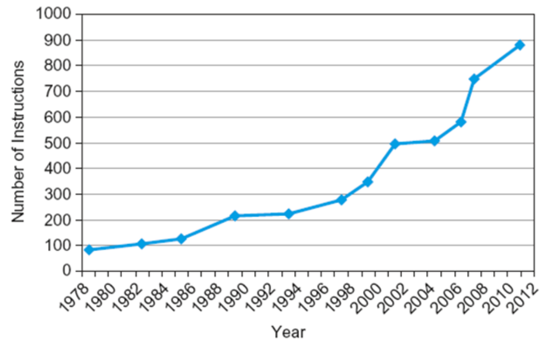
	Method	Opt	Bubble	Quick	Speedup quick vs bubble
			Relative performance		
C	Compiler	None	1.00	1.00	2468
C	Compiler	O1	2.37	1.50	1562
C	Compiler	O2	2.38	1.50	1555
C	Compiler	O3	2.41	1.91	1955
Java	Interpreter		0.12	0.05	1050
Java	JIT compiler		2.13	0.29	338

- Observations?

Fallacies and Pitfalls

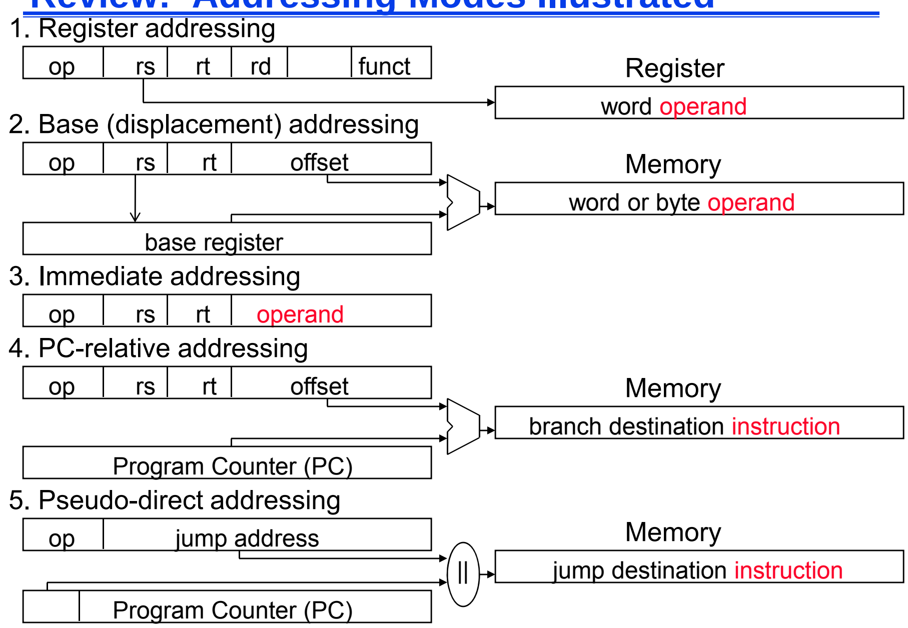
- Fallacies:
  - More powerful instructions mean higher performance
  - Write in assembly language for highest performance
  - Binary compatibility means successful ISAs (e.g., x86) don't change

x86 Instruction Set Growth

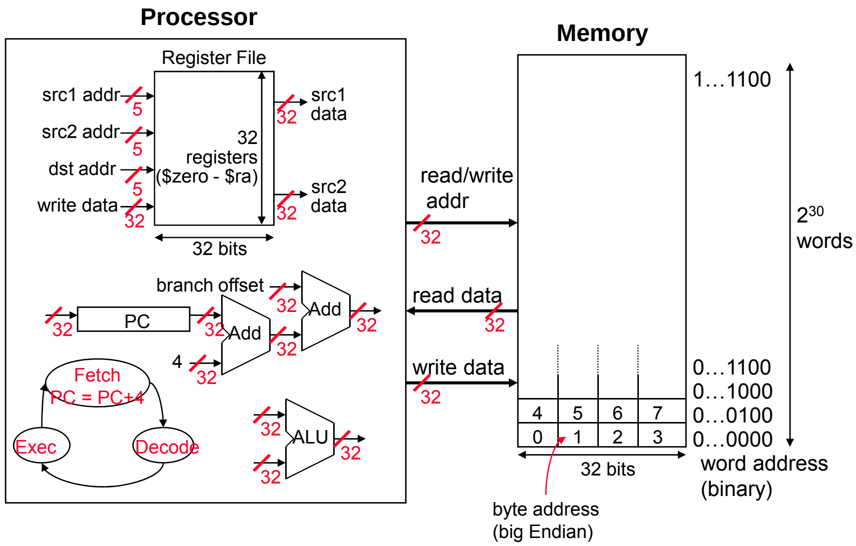


- Pitfalls:
  - Forgetting that sequential word addresses in machines with byte addressing don't differ by one (but by 4 !)
  - Using a pointer to an automatic variable outside its defining procedure

Review: Addressing Modes Illustrated



Review: MIPS Organization So Far



Lecture Reading and Reminders

- This lecture
  - MIPS ISA Review, PH, Chapter 2, Appendix A
- Next lecture
  - MIPS ALU Review, PH, Chapter 3, Appendix B
- Reminders
  - HW1 posted on Angel, due Sep 4 by 11:55pm in Angel DropBox
  - Quiz 1 posted on Angel, will close midnight Sept 8
  - Attend one of the unix + SimpleScalar tutorial sessions in the lab (218 IST) Sept 9^{th} and 10^{th} from 7:30 to 9pm (HW2 will contain a SimpleScalar simulation question)
  - First evening midterm exam scheduled
    - Thursday, October 2^{th} , 20:15 to 22:15, Location 262 Willard
    - Please let me know ASAP (via email) if you have a conflict