16.482 / 16.561
Computer Architecture and Design

Instructor: Dr. Michael Geiger

Fall 2013

Lecture 1:

Course overview

Introduction to computer architecture

Intro to MIPS instruction set

Lecture outline

- Course overview
  - Instructor information
  - Course materials
  - Course policies
  - Resources
  - Course outline
- Introduction to computer architecture

Course staff & meeting times

- Lectures:
  - Th 6:30-9:20, Kitson 310
- Instructor: Dr. Michael Geiger
  - E-mail: Michael_Geiger@uml.edu
  - Phone: 978-934-3618 (x43618 on campus)
  - Office: 118A Perry Hall
  - Office hours: M 1-2:30, W 1-2:30, Th 3-4:30

Course materials

- Textbook:
  - David A. Patterson and John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 5th edition, 2013.
    - ISBN: 9780124077263
- Course tools: TBD, but will likely work with QtSpim simulator (link on web page)

Additional course materials

- Course websites:
  http://mgeiger.eng.uml.edu/compArch/sp14/index.htm
  http://mgeiger.eng.uml.edu/compArch/sp14/schedule.htm
  - Will contain lecture slides, handouts, assignments
- Discussion group through piazza.com:
  - Allow common questions to be answered for everyone
  - All course announcements will be posted here
  - Will use as class mailing list—please enroll ASAP

Course policies

- Prerequisites: 16.265 (Logic Design) and 16.317 (Microprocessors I)
- Academic honesty
  - All assignments are to be done **individually** unless explicitly specified otherwise by the instructor
  - Any copied solutions, whether from another student or an outside source, are subject to penalty
  - You may discuss general topics or help one another with specific errors, but **do not share assignment solutions**
    - Must acknowledge assistance from classmate in submission

Grading and exam dates

- Grading breakdown
  - Homework assignments: 55%
  - Midterm exam: 20%
  - Final exam: 25%
- Exam dates
  - Midterm exam: Thursday, March 13 in class
  - Final exam: Thursday, May 1

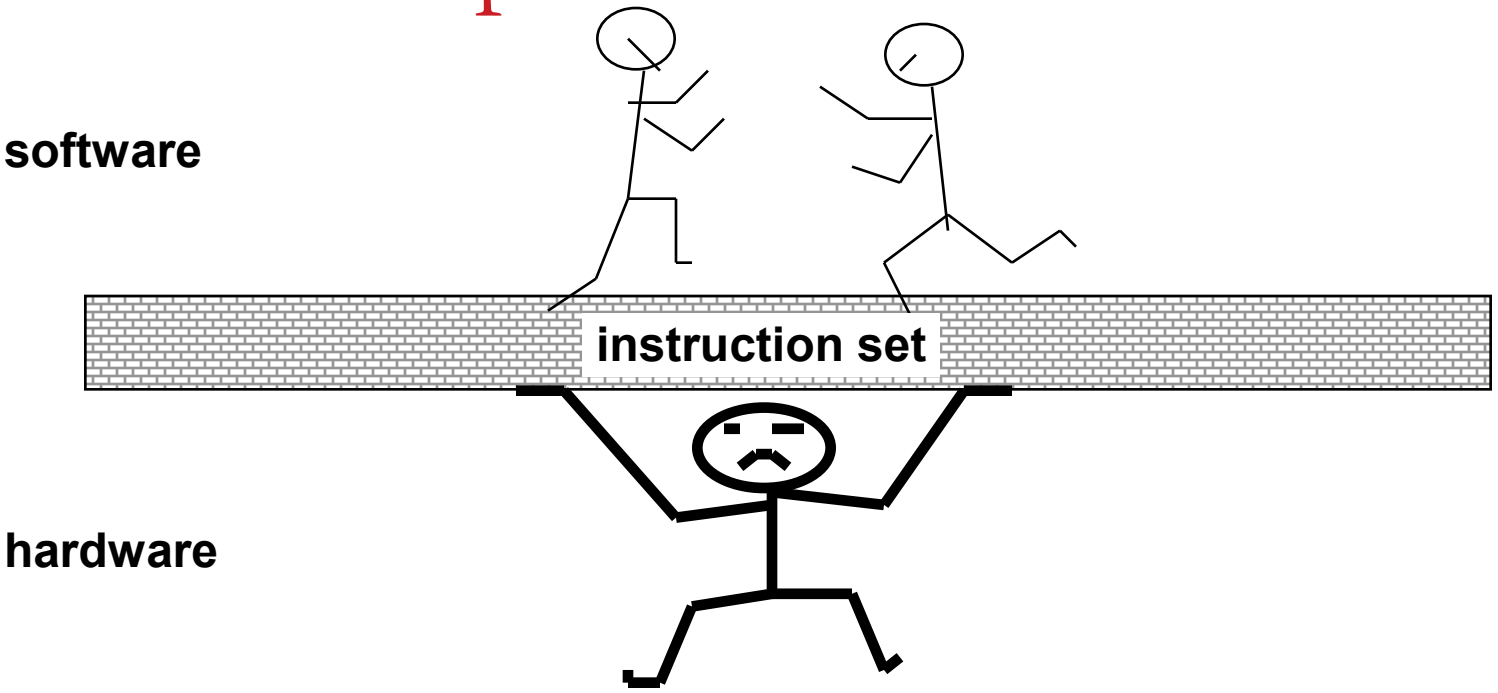
Tentative course outline

- General computer architecture introduction
- Instruction set architecture
- Digital arithmetic
- Datapath/control design
  - Basic datapath
  - Pipelining
  - Multiple issue and instruction scheduling
- Memory hierarchy design
  - Caching
  - Virtual memory
- Storage and I/O
- Multiprocessor systems

What is computer architecture?

- High-level description of
  - Computer hardware
    - Less detail than logic design, more detail than black box
  - Interaction between software and hardware
    - Look at how performance can be affected by different algorithms, code translations, and hardware designs
- Can use to explain
  - General computation
  - A class of computers
  - A specific system

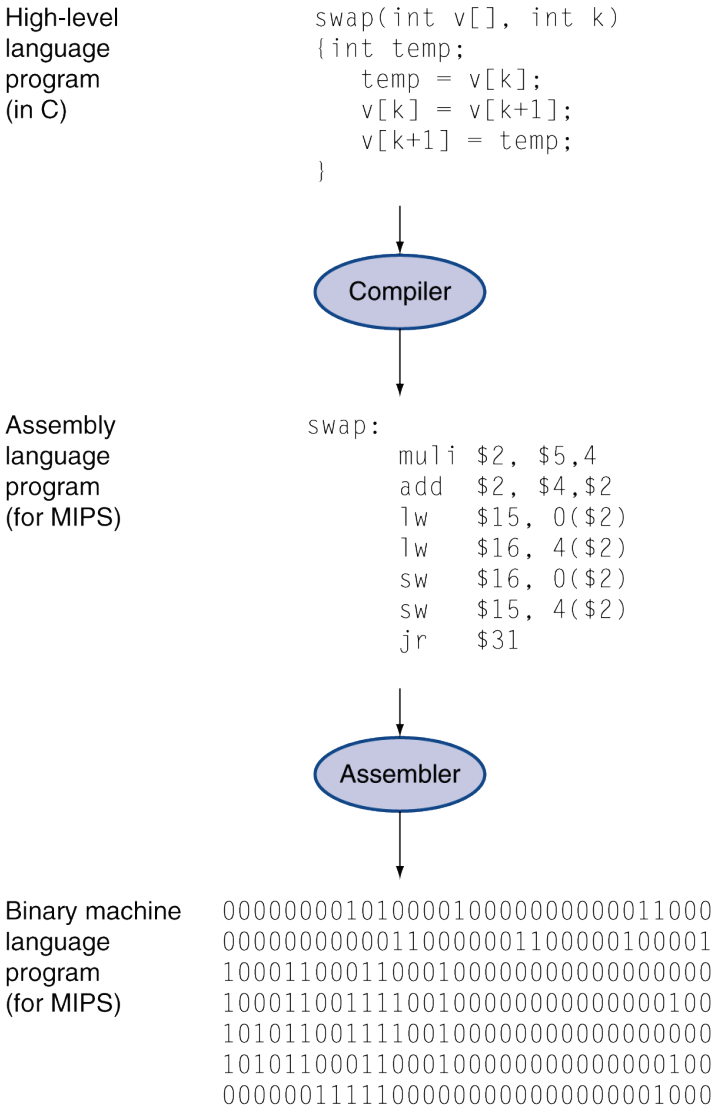
What is computer architecture?



- Classical view: instruction set architecture (ISA)
  - Boundary between hardware and software
  - Provides abstraction at both high level and low level
- More modern view: ISA + hardware design
  - Can talk about processor architecture, system architecture

Role of the ISA

- User writes high-level language (HLL) program
- Compiler converts HLL program into assembly for the particular *instruction set architecture* (ISA)
- Assembler converts assembly into machine language (bits) for that ISA
- Resulting machine language program is loaded into memory and run



ISA design goals

- The ultimate goals of the ISA designer are
  1. To create an ISA that allows for fast hardware implementations
  2. To simplify choices for the compiler
  3. To ensure the longevity of the ISA by anticipating future technology trends
- Often tradeoffs (particularly between 1 & 2)
- Example ISAs: X86, PowerPC, SPARC, ARM, MIPS, IA-64
  - May have multiple hardware implementations of the same ISA
    - Example: i386, i486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium IV

ISA design

- Think about a HLL statement like

```
X[i] = i * 2;
```

- ISA defines how such statements are translated to machine code
  - What information is needed?

ISA design (continued)

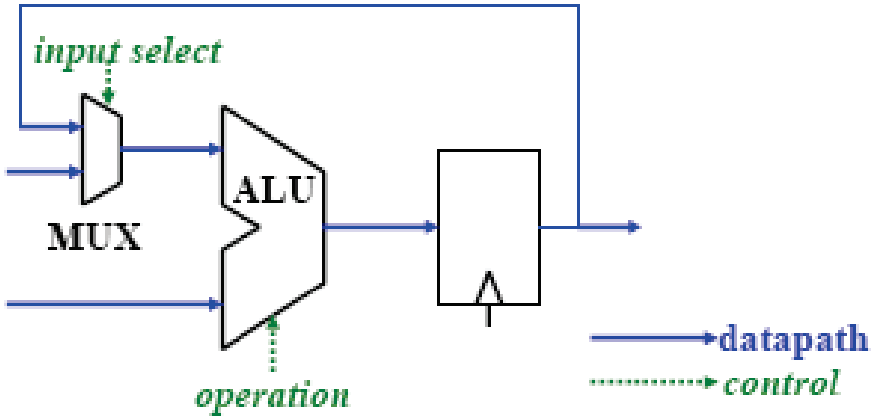
- Questions every ISA designer must answer
  - How will the processor implement this statement?
    - What operations are available?
    - How many operands does each instruction use?
    - How do we reference the operands?
  - Where are X[i] and i located?
    - What types of operands are supported?
    - How big are those operands
  - Instruction format issues
    - How many bits per instruction?
    - What does each bit or set of bits represent?
    - Are all instructions the same length?

Design goal: fast hardware

- From ISA perspective, must understand how processor executes instruction
  1. Fetch the instruction from memory
  2. Decode the instruction
  3. Determine addresses for operands
  4. Fetch operands
  5. Execute instruction
  6. Store result (and go back to step 1 …)
- Steps 1, 2, and 5 involve operation issues
  - What types of operations are supported?
- Steps 2-6 involve operand issues
  - Operand size, number, location
- Steps 1-3 involve instruction format issues
  - How many bits in instruction, what does each field mean?

Designing fast hardware

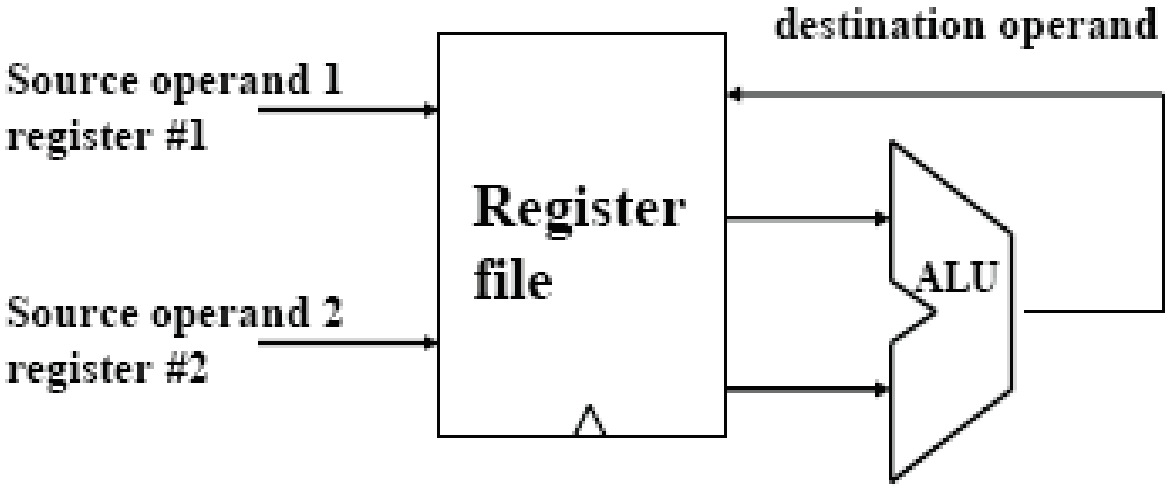
- To build a fast computer, we need
  1. Fast fetching and decoding of instructions
  2. Fast operand access
  3. Fast operation execution
- Two broad areas of hardware that we must optimize to ensure good performance
  - *Datapaths* pass data to different units for computation
  - *Control* determines flow of data through datapath and operation of each functional unit



Designing fast hardware

- Fast instruction fetch and decode
  - We'll address this (from an ISA perspective) today
- Fast operand access
  - ISA classes: where do we store operands?
  - Addressing modes: how do we specify operand locations?
    - We'll also discuss this today
  - We know registers can be used for fast accesses
  - We'll talk about increasing memory speeds later
- Fast execution of simple operations
  - Optimize common case
  - Dealing with multi-cycle operations: one of our first challenges

Making operations fast



- More complex operations take longer → keep things simple!
- Many programs contain mostly simple operations
  - add, and, load, branch …
- Optimize the common case
  - Make these simple operations work well!
  - Can execute them in a single cycle (with a fast clock, too)
    - If you're wondering how, wait until we talk about pipelining ...

Specifying operands

- Most common arithmetic instructions have three operands
  - 2 source operands, 1 destination operand
  - e.g. A = B + C → add A, B, C
- ISA classes for specifying operands:
  - *Accumulator*
    - Uses a single register (fast memory location close to processor)
    - Requires only one address per instruction (ADD addr)
  - *Stack*
    - Requires no explicit memory addresses (ADD)
  - *Memory-Memory*
    - All 3 operands may be in memory (ADD addr1, addr2, addr3)
  - *Load-Store*
    - All arithmetic operations use only registers (ADD r1, r2, r3)
    - Only load and store instructions reference memory

Making operand access fast

- Operands (generally) in one of two places
  - Memory
  - Registers
- Which would we prefer to use? Why?
- Advantages of registers as operands
  - Instructions are shorter
    - Fewer possible locations to specify → fewer bits
  - Fast implementation
    - Fast to access and easy to reuse values
- We'll talk about fast memory later …
- Where else can operands be encoded?
  - Directly in the instruction: ADDI R1, R2, 3
  - Called immediate operands

RISC approach

- Fixed-length instructions that have only a few formats
  - Simplify instruction fetch and decode
  - Sacrifice code density
    - Some bits are wasted for some instruction types
    - Requires more memory
- Load-store architecture
  - Allows fast implementation of simple instructions
  - Easier to pipeline
  - Sacrifice code density
    - More instructions than register-memory and memory-memory ISAs
- Limited number of addressing modes
  - Simplify effective address (EA) calculation to speed up memory access
- Few (if any) complex arithmetic functions
  - Build these from simpler instructions

MIPS: A "Typical" RISC ISA

- 32-bit fixed format instruction (3 formats)
- Registers
  - 32 32-bit integer GPRs (R1-R31, R0 always = 0)
  - 32 32-bit floating-point GPRs (F0-F31)
    - For double-precision FP, registers paired
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store: base + displacement
- Simple branch conditions
- Delayed branch

Operands

- Example: load-store instructions of form:

  LOAD R2, C
  ADD R3, R1, R2
  STORE R3, A

- Three general classes of operands
  - Immediate operands: encoded directly in instruction
    - Example: ADDI R3, R1, 25
  - Register operands: register number encoded in instruction, register holds data
    - Example: R1, R2, and R3 in above code sequence
  - Memory operands: address encoded in instruction, data stored in memory
    - Example: A and C in above code sequence
    - In load-store processor, can't directly operate on these data

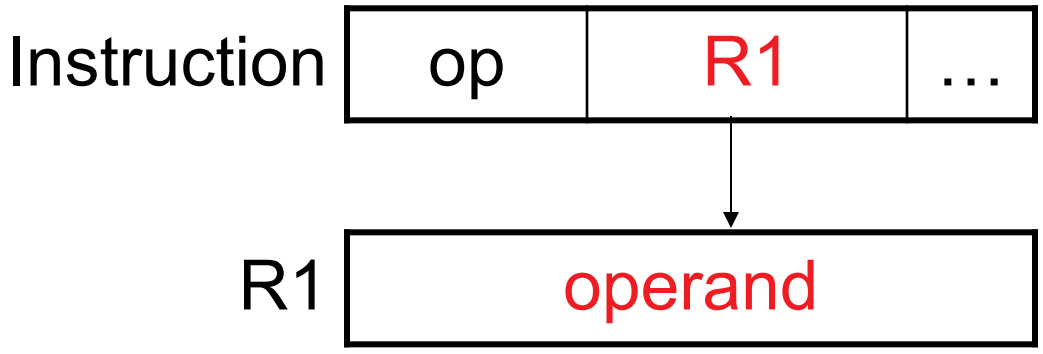
Memory operands

- Specifying addresses is not straightforward
  - Length of address usually = length of instruction
  - Obviously don't want to dedicate that much space to an entire address
  - What are some ways we might get around this?
- Addressing modes: different ways of specifying operand location
  - Already discussed two
    - Immediate: operand encoded directly in instruction
    - Register direct: operand in register, register # in instruction
  - For values in memory, have to calculate effective address

Immediate, register direct modes

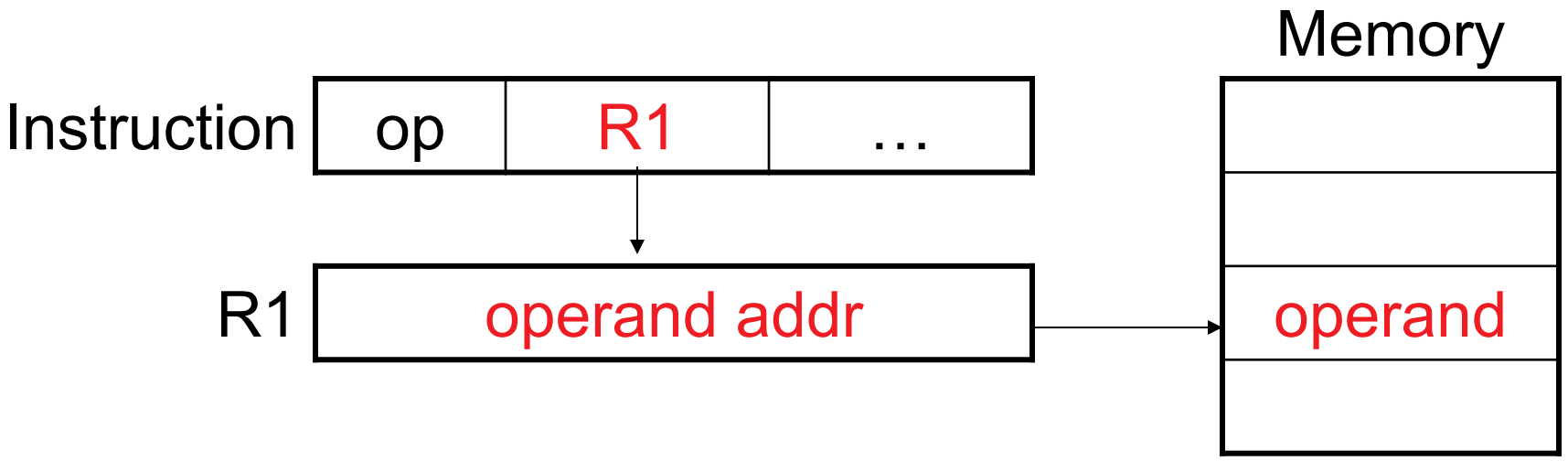
- *Immediate:* instruction contains the operand
  - Example: addi R3, R1, **100**

- *Register direct:* register contains the operand
  - Example: add R3, **R1**, R2



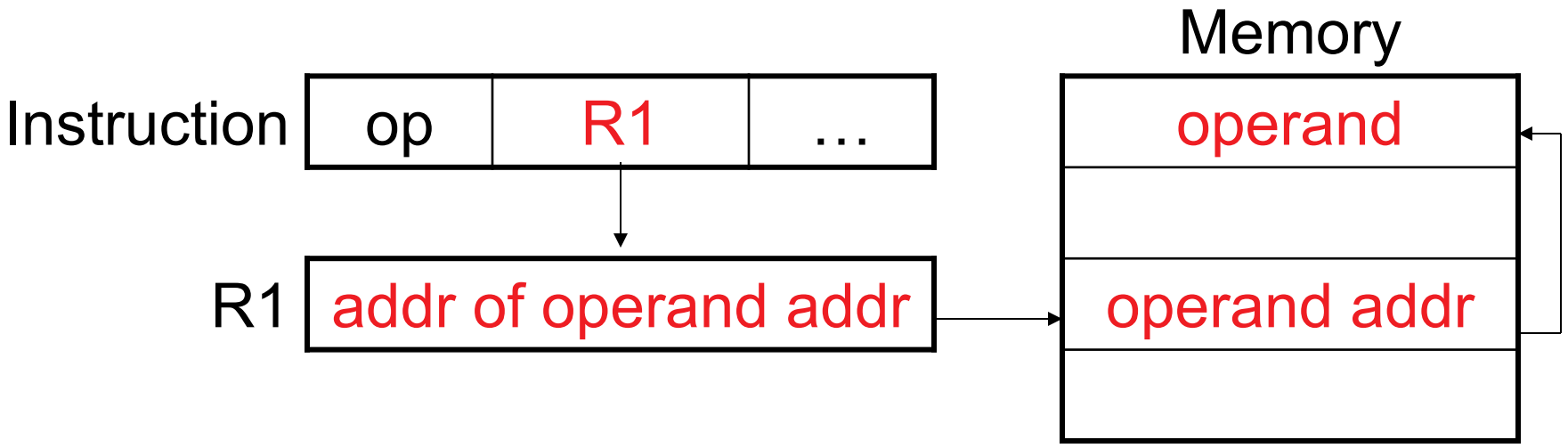
Register indirect mode

- *Register indirect:* register contains address of operand
  - Example: lw R2, **(R1)**



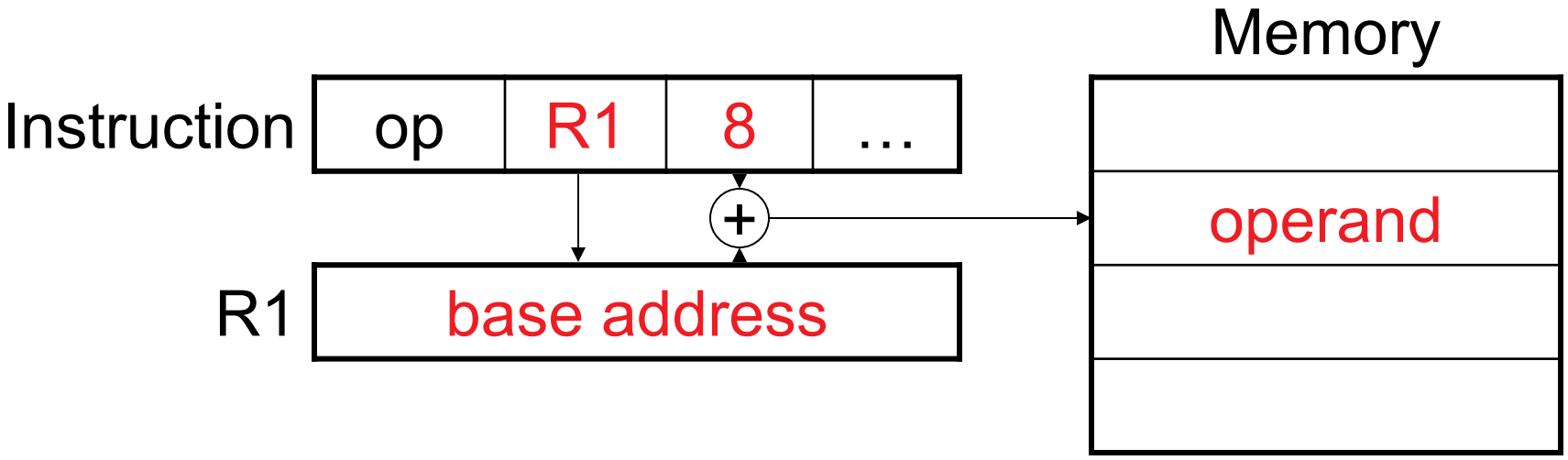
Memory indirect mode

- *Memory indirect:* memory contains address of operand
  - Allows for more efficient pointer implementations



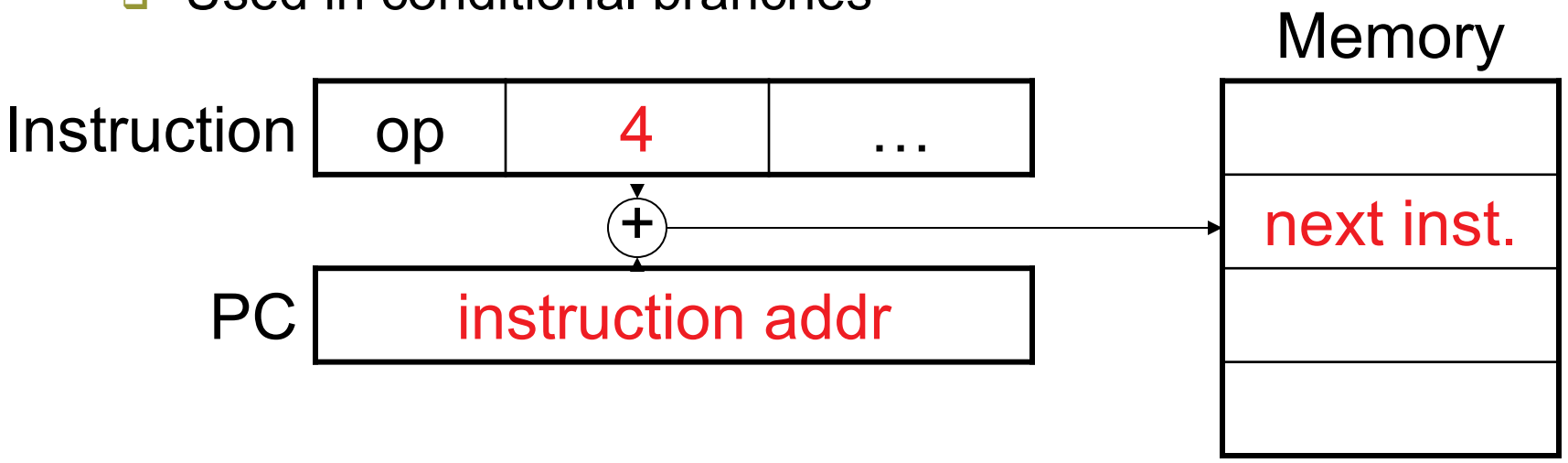
Base + displacement mode

- *Base + displacement*: Operand address = register + constant
  - Example: lw R3, **8(R1)**



PC-relative mode

- Need some way to specify instruction addresses, too
- *PC-relative:* Address of next instruction = PC + const
  - PC = program counter
    - The memory address of the current instruction
  - Used in conditional branches



Instruction formats

- *Instruction formats* define what each bit or set of bits means
- Different instructions need to specify different information
  - Diffferent numbers of operands, types of operands, etc.
- What are the pros and cons if we have …
  - Many instruction formats
    + Can tailor format to specific instructions
    - Complicate decoding
    - May use more bits
  - Few fixed, regular formats
    + Simpler decoding
    - Less flexibility in encoding
- One of our hardware design goals: fast decoding

Instruction length

- What are the pros and cons if we use …
  - Variable-length instructions (CISC)

    + No bits wasted on unused fields/operands

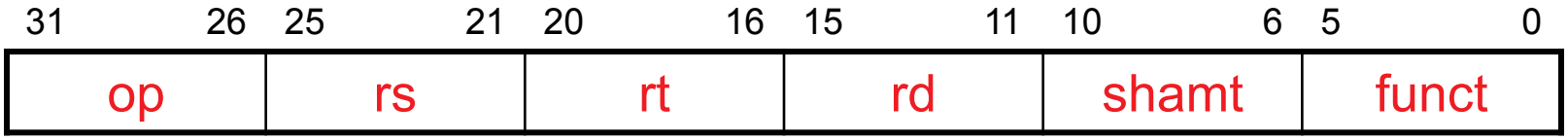
    - Complex fetch and decoding
  - Fixed-length instructions (RISC)

    + Simple fetch and decoding

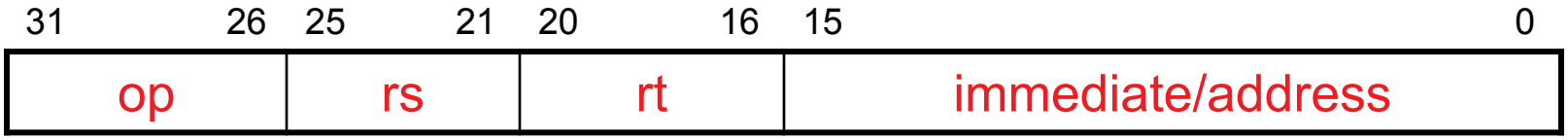
    - Instruction encoding is less compact

MIPS instruction formats

- All fixed length (32-bit) instructions
- Register instructions: R-type



- Immediate instructions: I-type



- Jump instructions: J-type

MIPS instruction formats (cont.)

- Notation from the previous slide
  - **op** is a 6-bit operation code (opcode)
  - **rs** is a 5-bit source register specifier
  - **rt** is a 5-bit (source or destination) register specifier or branch condition
  - **rd** is a 5-bit destination register specifier
  - **shamt** is a 5-bit shift amount
  - **funct** is a 6-bit function field
  - **immediate** is a 16-bit immediate, branch displacement, or memory address displacement
  - **target** is a 26-bit jump target address
- Simplifications
  - Fixed length (32 bits)
  - Limited number of field types
  - Many fields located in same location in different formats

MIPS instruction fields

- Assume a MIPS instruction is represented by the hexadecimal value 0xDEADBEEF
- List the values for each instruction field, assuming that the instruction is
  - An R-type instruction
  - An I-type instruction
  - A J-type instruction

MIPS instruction fields

- List the values for each instruction field, assuming that 0xDEADBEEF is
  - An R-type instruction
  - An I-type instruction
  - A J-type instruction
- 0xDEADBEEF =

  1101 1110 1010 1101 1011 1110 1110 1111

MIPS addressing modes

- MIPS implements several of the addressing modes discussed earlier
- To address operands
  - Immediate addressing
    - Example: addi $t0, $t1, **150**
  - Register addressing
    - Example: sub **$t0, $t1, $t2**
  - Base addressing (base + displacement)
    - Example: lw $t0, **16($t1)**
- To transfer control to a different instruction
  - PC-relative addressing
    - Used in conditional branches
  - Pseudo-direct addressing
    - Concatenates 26-bit address (from J-type instruction) shifted left by 2 bits with the 4 upper bits of the PC

MIPS integer registers

Name	Register number	Usage
`$zero`	0	Constant value 0
`$v0-$v1`	2-3	Values for results and expression evaluation
`$a0-$a3`	4-7	Function arguments
`$t0-$t7`	8-15	Temporary registers
`$s0-$s7`	16-23	Callee save registers
`$t8-$t9`	24-25	Temporary registers
`$gp`	28	Global pointer
`$sp`	29	Stack pointer
`$fp`	30	Frame pointer
`$ra`	31	Return address

- List gives mnemonics used in assembly code
  - Can also directly reference by number ($0, $1, etc.)
- Conventions
  - $s0-$s7 are preserved on a function call (callee save)
  - Register 1 ($at) reserved for assembler
  - Registers 26-27 ($k0-$k1) reserved for operating system

Computations in MIPS

- All computations occur on full 32 bit words
- Computations use *signed* (2's complement) or *unsigned* operands (positive numbers)
  - Example: 1111 1111 1111 1111 1111 1111 1111 1111 is –1 as a signed number and 4,294,967,294 as an unsigned number
- Operands are in registers or are *immediates*
- Immediate (constant) values are only 16 bits
  - 32-bit instruction must also hold opcode, source and destination register numbers
  - Value is *sign extended* before usage to 32 bits
  - Example: 1000 0000 0000 0000

    becomes

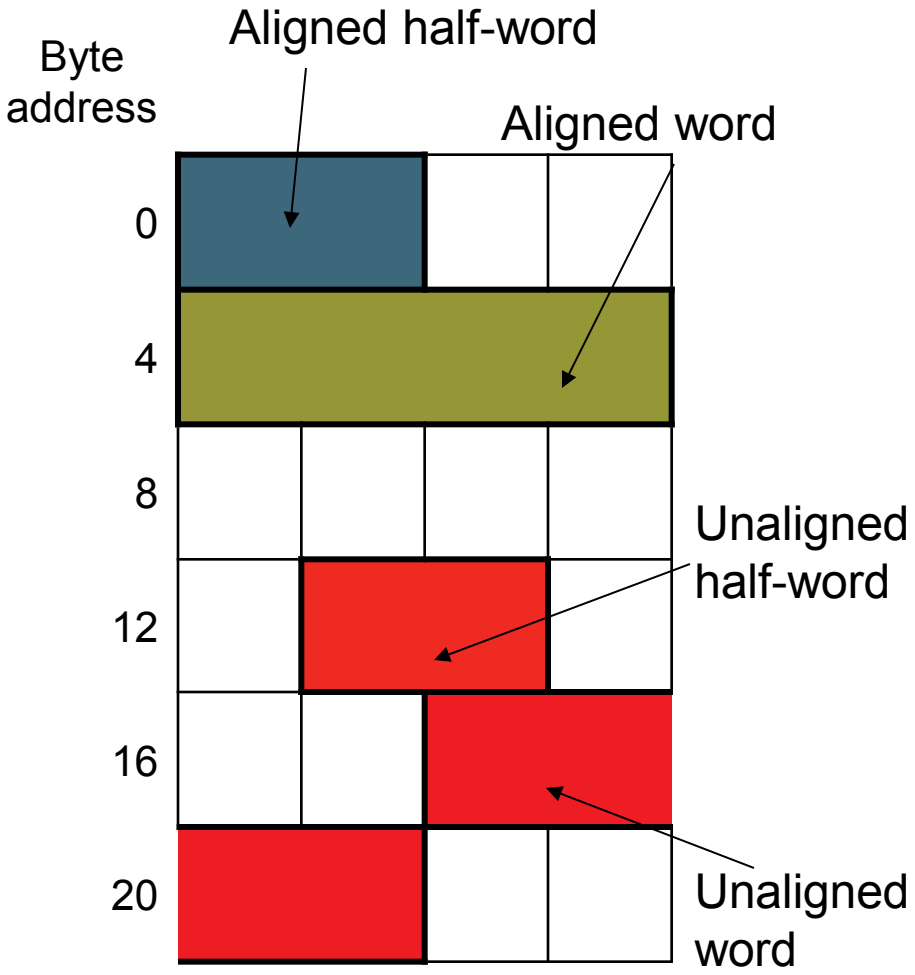
    1111 1111 1111 1111 1000 0000 0000 0000

MIPS instruction categories

- Data transfer instructions
  - Example: lw, sb
  - Always I-type
- Computational instructions (arithmetic/logical)
  - Examples: add, and, sll
  - Can be R-type or I-type
- Control instructions
  - Example: beq, jr
  - Any of the three formats (R-type, I-type, J-type)

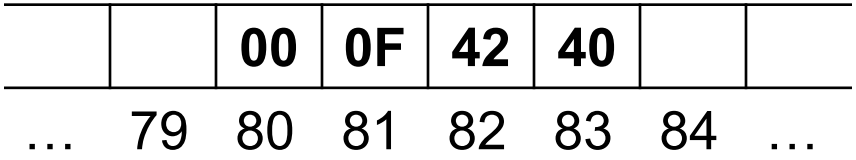
MIPS data transfer instructions

- Memory operands can be bytes, half-words (2 bytes), or words (4 bytes [32 bits])
  - opcode determines the operand size
- Half-word and word addresses must be aligned
  - Divisible by number of bytes being accessed
  - Bit 0 must be zero for half-word accesses
  - Bits 0 and 1 must be zero for word accesses

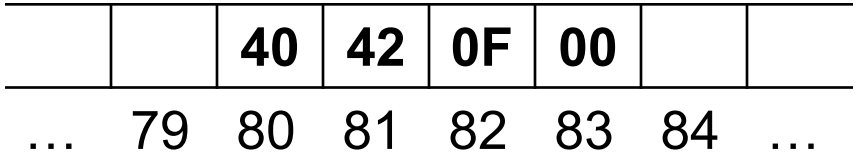


Byte order ("endianness")

- In a multi-byte operand, how are the bytes ordered in memory?
- Assume the value 1,000,000 (0xF4240) is stored at address 80
  - In a *big-endian* machine, the most significant byte (the "big" end) is at address 80



  - In a *little-endian* machine, it's the other way around



Big-endian vs. little-endian

- Big-endian systems
  - MIPS, Sparc, Motorola 68000
- Little-endian systems
  - Most Intel processors, Alpha, VAX
- Neither one is “better;” it’s simply a matter of preference …
- … but there are compatibility issues that arise when transferring data from one to the other

MIPS data transfer instructions (cont.)

- For all cases, calculate effective address first
  - MIPS doesn't use segmented memory model like x86
  - Flat memory model → EA = address being accessed
- lb, lh, lw
  - Get data from addressed memory location
  - Sign extend if lb or lh, load into rt
- lbu, lhu, lwu
  - Get data from addressed memory location
  - Zero extend if lb or lh, load into rt
- sb, sh, sw
  - Store data from rt (partial if sb or sh) into addressed location

Data transfer examples

- Say memory holds the word 0xABCD1234, starting at address 0x1000, $t0 holds the value 0x1000, and $s0 holds 0xDEADBEEF
- What are the results of the following instructions?
  - lh $t1, 2($t0)
  - lb $t2, 1($t0)
  - lbu $t3, 0($t0)
  - sh $s0, 0($t0)
  - sb $s0, 3($t0)

Solutions to examples

- If mem[0x1000] = 0xABCD1234, $t0 holds the value 0x1000, and $s0 holds 0xDEADBEEF
  - lh $t1, 2($t0)
    - $t1 = mem[0x1002] = 0x00001234
  - lb $t2, 1($t0)
    - $t2 = mem[0x1001] = 0xFFFFFFCD
  - lbu $t3, 0($t0)
    - $t3 = mem[0x1000] = 0x000000AB
  - sh $s0, 0($t0)
    - Change 16 bits at address 0x1000
    - mem[0x1000] = 0xBEEF1234
  - sb $s0, 3($t0)
    - Change 8 bits at address 0x1003
    - mem[0x1000] = 0xABCD12EF

More data transfer examples

- If **A** is an array of words and the starting address of **A** (memory address of **A[0]**) is in $s4, perform the operation:

  A[3] = A[0] + A[1] - A[2]

- Solution:

```
lw $s1, 0($s4)        # A[0] into $s1
lw $s2, 4($s4)        # A[1] into $s2
add $s1, $s1, $s2     # A[0]+A[1] into $s1
lw $s2, 8($s4)        # A[2] into $s2
sub $s1, $s1, $s2     # A[0]+A[1]-A[2] into $s1
sw $s1, 12($s4)       # store result in A[3]
```

MIPS computational instructions

- Arithmetic
  - Signed: add, sub, mult, div
  - Unsigned: addu, subu, multu, divu
  - Immediate: addi, addiu
    - Immediates are sign-extended (why?)
- Logical
  - and, or, nor, xor
  - andi, ori, xori
    - Immediates are zero-extended (why?)
- Shift (logical and arithmetic)
  - srl, sll – shift right (left) logical
    - Shift the value in rs by shamt digits to right or left
    - Fill empty positions with 0s
    - Store the result in rd
  - sra – shift right arithmetic
    - Same as above, but sign-extend the high-order bits

Signed vs. unsigned computation

- What's the difference between add and addu?
  - Result looks exactly the same
  - addu ignores *overflow*
    - add used for most normal computation
    - addu used for memory addresses
    - C language ignores overflow → compiler generates unsigned computation
  - 4-bit example:

    (-8) + (-8) = (-16) → can't represent 16 with 4 bits

```
  1000
+ 1000
-----
 10000
```

MIPS computational instructions (cont.)

- Set less than
  - Used to evaluate conditions
    - Set rd to 1 if condition is met, set to 0 otherwise
  - slt, sltu
    - Condition is rs < rt
  - slti, sltiu
    - Condition is rs < immediate
    - Immediate is *sign-extended*
- Load upper immediate (lui)
  - Shift immediate 16 bits left, append 16 zeros to right, put 32-bit result into rd

Examples of arithmetic instructions

Instruction	Meaning
add $s1, $s2, $s3	$s1 = $s2 + $s3 3 registers; signed addition
sub $s1, $s2, $s3	$s1 = $s2 - $s3 3 registers; signed subtraction
addu $s1, $s2, $s3	$s1 = $s2 + $s3 3 registers; unsigned addition
addi $s1, $s2, 50	$s1 = $s2 + 50 2 registers and immediate; signed
addiu $s1, $s2, 50	$s1 = $s2 + 50 2 registers and immediate; unsigned

Examples of logical instructions

Instruction	Meaning
and $s1, $s2, $s3	$s1 = $s2 & $s3 3 registers; logical AND
or $s1, $s2, $s3	$s1 = $s2 | $s3 3 registers; logical OR
xor $s1, $s2, $s3	$s1 = $s2 ⊕ $s3 3 registers; logical XOR
nor $s1, $s2, $s3	$s1 = ~($s2 + $s3) 3 registers; logical NOR

Computational instruction examples

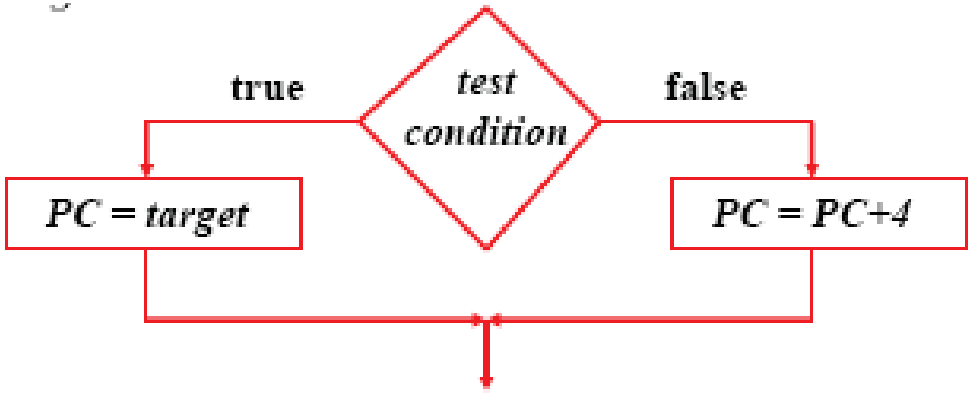
- Say $t0 = 0x00000001, $t1 = 0x00000004, $t2 = 0xFFFFFFFF
- What are the results of the following instructions?
  - sub $t3, $t1, $t0
  - addi $t4, $t1, 0xFFFF
  - andi $t5, $t2, 0xFFFF
  - sll $t6, $t0, 5
  - slt $t7, $t0, $t1
  - lui $t8, 0x1234

Solutions to examples

- If $t0 = 0x00000001, $t1 = 0x00000004, $t2 = 0xFFFFFFFF
  - sub $t3, $t1, $t0
    - $t3 = 0x00000004 – 0x00000001 = **0x00000003**
  - addi $t4, $t1, 0xFFFF
    - $t4 = 0x00000004 + 0xFFFFFFFF = **0x00000003**
  - andi $t5, $t2, 0xFFFF
    - $t5 = 0xFFFFFFFF AND 0x0000FFFF = **0x0000FFFF**
  - sll $t6, $t0, 5
    - $t6 = 0x00000001 << 5 = **0x00000020**
  - slt $t7, $t0, $t1
    - $t7 = 1 if ($t0 < $t1)
    - $t0 = 0x00000001, $t1 = 0x00000004 → **$t7 = 1**
  - lui $t8, 0x1234
    - $t8 = 0x1234 << 16 = **0x12340000**

MIPS control instructions

- Branch instructions test a condition
  - Equality or inequality of rs and rt
    - beq, bne
    - Often coupled with slt, sltu, slti, sltiu
  - Value of rs relative to rt
    - Pseudoinstructions: blt, bgt, ble, bge
- Target address → add sign extended immediate to the PC
  - Since all instructions are words, immediate is shifted left two bits before being sign extended



Pseudoinstructions

- Assembler recognizes certain "instructions" that aren't actually part of MIPS ISA
  - Common operations that can be implemented using other relatively simple operations
  - Easier to read and write assembly in terms of these *pseudoinstructions*
  - Example: MIPS only has beq, bne instructions, but assembler recognizes bgt, bge, blt, ble
- Assembler converts pseudoinstruction into actual instruction(s)
  - If extra register needed, use $at
  - Example: bgt $t0, $t1, label → slt $at, $t1, $t0
    bne $at, $zero, label

MIPS control instructions (cont.)

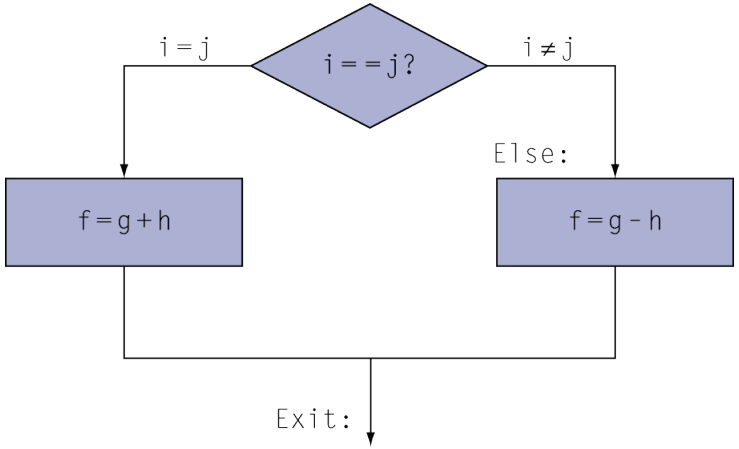
- *Jump* instructions unconditionally branch to the address formed by either
  - Shifting left the 26-bit target two bits and combining it with the 4 high-order PC bits
    - j
  - The contents of register $rs
    - jr
- *Branch-and-link* and *jump-and-link* instructions also save the address of the next instruction into $ra
  - jal
  - Used for subroutine calls
  - jr $ra used to return from a subroutine

Compiling If Statements

- C code:

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

  - f, g, … in $s0, $s1, …



- Compiled MIPS code:

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

Assembler calculates addresses

Compiling Loop Statements

- C code:

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

  - i in $s3, k in $s5, address of save in $s6

- Compiled MIPS code:

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

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

Register Usage

- $a0 – $a3: arguments (reg's 4 – 7)
- $v0, $v1: result values (reg's 2 and 3)
- $t0 – $t9: temporaries
  - Can be overwritten by callee
- $s0 – $s7: saved
  - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)

Procedure Call Instructions

- Procedure call: jump and link

  `jal ProcedureLabel`

  - Address of following instruction put in $ra
  - Jumps to target address
- Procedure return: jump register

  `jr $ra`

  - Copies $ra to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Leaf Procedure Example

- C code:

```
int leaf_example (int g, h, i, 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

- MIPS code:

```
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
```

Save $s0 on stack

Procedure body

Result

Restore $s0

Return

Non-Leaf Procedures

- Procedures that call other procedures
- 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

- C code:

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

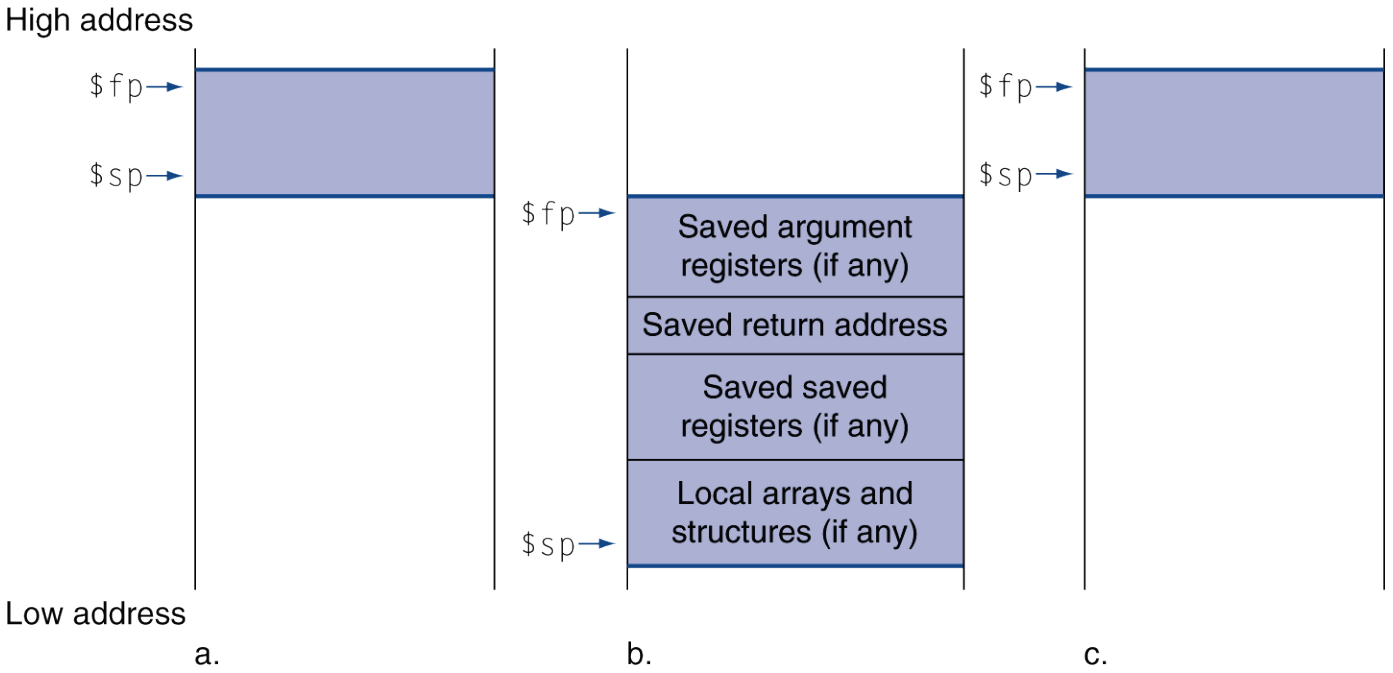
  - Argument n in $a0
  - Result in $v0

Non-Leaf Procedure Example

- MIPS code:

```
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
   mul  $v0, $a0, $v0     # multiply to get result
   jr   $ra               # and return
```

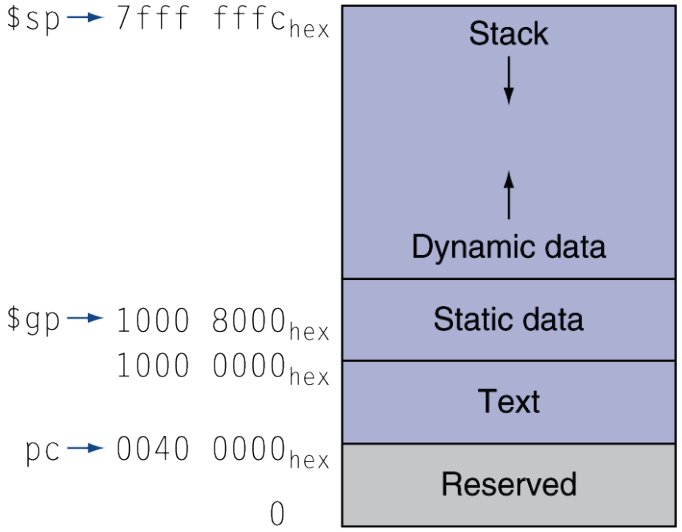
Local Data on the Stack



- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

Memory Layout

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



String Copy Example

- C code (naïve):
  - Null-terminated string

```
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

- MIPS code:

```
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
    lbu  $t2, 0($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
```

C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

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

  - v in $a0, k in $a1, temp in $t0

The Procedure Swap

```
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
```

The Sort Procedure in C

- Non-leaf (calls swap)

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

  - v in $a0, k in $a1, i in $s0, j in $s1

The Procedure Body

```
          move $s2, $a0            # save $a0 into $s2
          move $s3, $a1            # save $a1 into $s3
          move $s0, $zero          # i = 0
for1tst:  slt  $t0, $s0, $s3       # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
          beq  $t0, $zero, exit1   # go to exit1 if $s0 ≥ $s3 (i ≥ n)
          addi $s1, $s0, -1        # j = i - 1
for2tst:  slti $t0, $s1, 0         # $t0 = 1 if $s1 < 0 (j < 0)
          bne  $t0, $zero, exit2   # go to exit2 if $s1 < 0 (j < 0)
          sll  $t1, $s1, 2         # $t1 = j * 4
          add  $t2, $s2, $t1       # $t2 = v + (j * 4)
          lw   $t3, 0($t2)         # $t3 = v[j]
          lw   $t4, 4($t2)         # $t4 = v[j + 1]
          slt  $t0, $t4, $t3       # $t0 = 0 if $t4 ≥ $t3
          beq  $t0, $zero, exit2   # go to exit2 if $t4 ≥ $t3
          move $a0, $s2            # 1st param of swap is v (old $a0)
          move $a1, $s1            # 2nd param of swap is j
          jal  swap                # call swap procedure
          addi $s1, $s1, -1        # j -= 1
          j    for2tst             # jump to test of inner loop
exit2:    addi $s0, $s0, 1         # i += 1
          j    for1tst             # jump to test of outer loop
```

Move params

Outer loop

Inner loop

Pass params & call

Inner loop

Outer loop

The Full Procedure

```
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
         …                      # procedure body
         …
         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
```

Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

Synchronization in MIPS

- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in rt
  - Fails if location is changed
    - Returns 0 in rt
- Example: atomic swap (to test/set lock variable)

```
try: add $t0,$zero,$s4 ;copy exchange value
     ll  $t1,0($s1)    ;load linked
     sc  $t0,0($s1)    ;store conditional
     beq $t0,$zero,try ;branch store fails
     add $s4,$zero,$t1 ;put load value in $s4
```

Final notes

- Next time:
  - More on MIPS instruction set (if needed)
  - Digital arithmetic
- Announcements/reminders:
  - Sign up for the course discussion group on Piazza!
  - HW 1 to be posted; due 1/30