CS 61c: Great Ideas in Computer Architecture

Introduction to Assembly Language

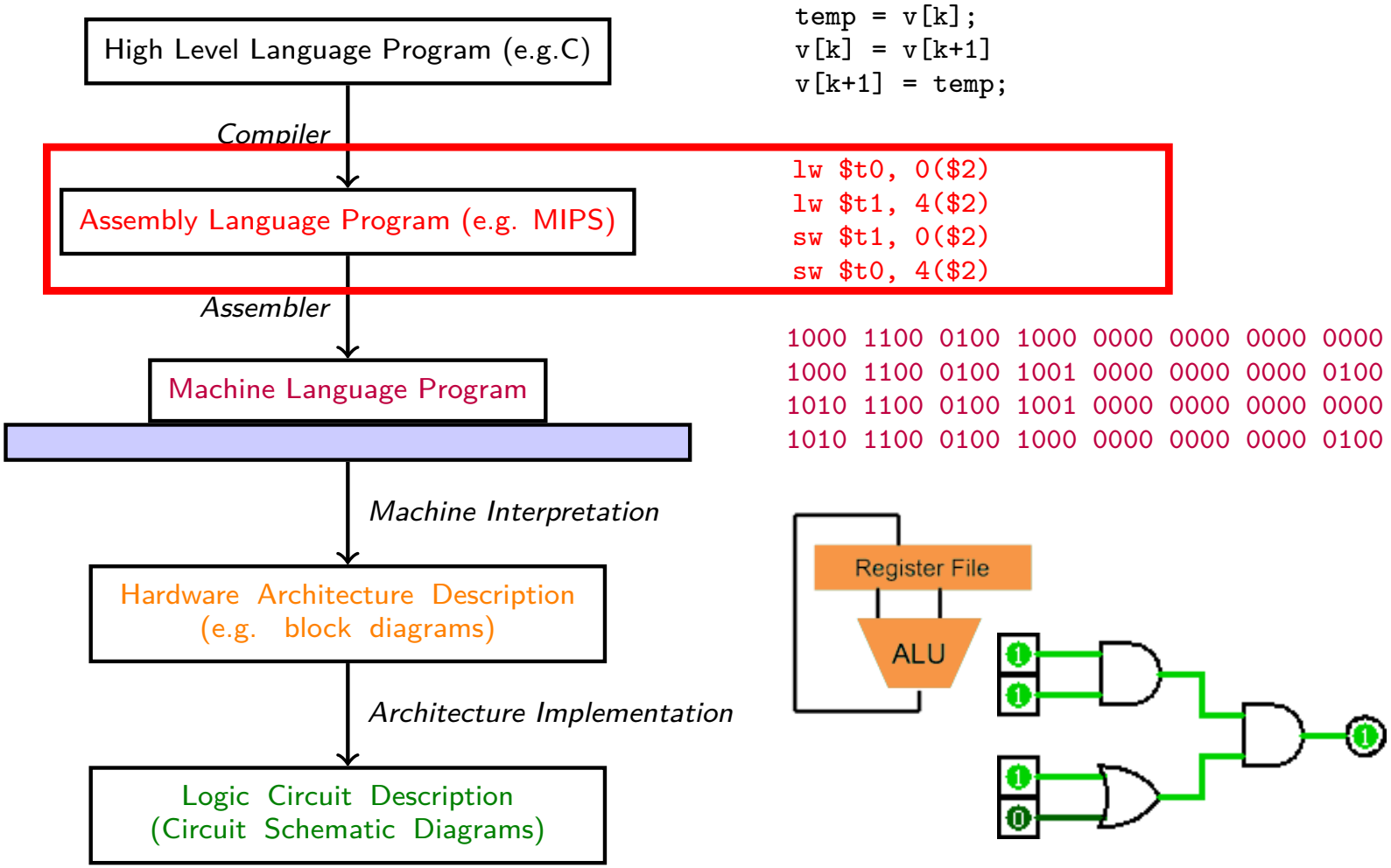
Instructor: Alan Christopher

June 30, 2014

Review

- C Memory Layout
  - Local variables disappear because the stack changes
  - Global variables don't disappear because they are in static data
  - Dynamic memory available using `malloc()` and `free()`
- Memory Management
  - K&R: first-fit, last-fit, best-fit for `malloc()`
- Many common memory problems

Great Idea #1: Levels of Representation/Interpretation



Outline

Machine Language I

- **instruction**: a "word" that a computer understands
- **ISA (instruction set architecture)**: Vocabulary of all the "words" a computer understands
- When to use the same ISA, when to use different?
  - e.g. iPhone and iPad use the same
  - e.g. iPhone and Macbook use different

Machine Language II

- Single ISA
  - Leverage common compilers, operating systems, etc
  - BUT relatively easy to retarget these for different ISAs
- Multiple ISAs
  - Specialized instructions for specialized applications
  - Different tradeoffs in resources used (e.g. functionality, memory demands, complexity, power consumption, etc)
  - Competition and innovation is good, especially in emerging environments (e.g. mobile devices)

Why Study Assembly?

- Understand computers at a deeper level
  - Learn to write more compact and efficient code
  - Can sometimes optimize better by hand than a compiler can (*sometimes*)
- More sensible for minimal applications
  - e.g. embedded computers
  - Eliminating OS, compilers, etc, reduces size and power consumption
  - Embedded computers outnumber PCs (by a *lot*)

Reduced Instruction Set Computing

- The early trend was to add more and more instructions to do elaborate operations – known as *Complex Instruction Set Computing* (CISC)
- Opposite philosophy emerged later: *Reduced Instruction Set Computing* (RISC)
  - Simpler (and smaller) instruction set makes it easier to build fast hardware
  - Let software do the complicated operations by composing simpler ones

Reduced Instruction Set Computing

- The early trend was to add more and more instructions to do elaborate operations – known as *Complex Instruction Set Computing* (CISC)
- Opposite philosophy emerged later: *Reduced Instruction Set Computing* (RISC)
  - Simpler (and smaller) instruction set makes it easier to build fast hardware
  - Let software do the complicated operations by composing simpler ones
- KISS!

Common RISC Simplifications

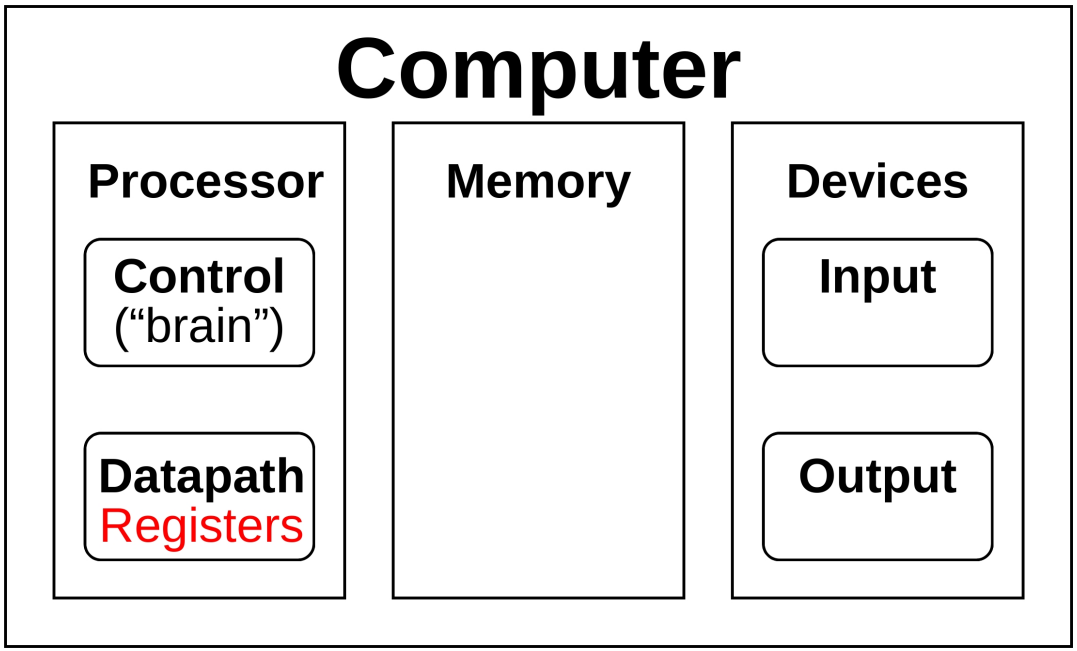
- **Fixed instruction length:**
  Simplifies fetching instructions from memory
- **Simplified addressing modes:**
  Simplifies fetching operands from memory
- **Fewer, simpler instructions in the ISA:**
  Simplifies instruction execution
- **Minimize memory access instructions (load/store):**
  Simplifies hardware for memory accesses
- **Let compiler do heavy lifting**
  Breaks complex statements into multiple assembly instructions

Mainstream ISAs

- Intel 80x86
  - Commonly used in PCs and laptops
  - Found in Core i3, Core i5, Core i7, etc.
- Advanced RISC Machine (ARM)
  - Smart phone-like devices: iPhone, iPad, iPod, etc
  - The most popular RISC by number of chips (20x more common than x86)
- MIPS (what we'll be learning)
  - Networking equipment, PS2, PSP
  - Very similar to ARM

Five Components of a Computer

- **Important:** Computers are subdivided into 5 parts
  - Control
  - Datapath
  - Memory
  - Input
  - Output



- Registers are part of the *datapath*

Hardware Operands

- In high-level langauges, number of variables limited only by available memory
- ISAs have a fixed, small number of operands, called registers
  - Special memory built directly into hardware
  - **Benefit:** Registers are *extremely* fast (less than 1 ns to access)
  - **Drawback:** Operations can only be performed on this fixed number of registers

MIPS Registers I

- MIPS has 32 registers
  - Each register is 32 bits wide and holds a *word*
- Tradeoff between speed and availability
  - Smaller number means faster hardware, but insufficient to hold data for typical C programs
- *Registers have no type:* The operation being performed determines how register contents are treated

MIPS Registers II

- Registers are denoted by '$', can be referenced by number (e.g. $23), or by name:
  - Registers that hold programmer variables
    $s0-$s7 ↔ $16-$23
  - Registers that hold temporary variables
    $t0-$t7 ↔ $8-$15
    $t8-$t9 ↔ $24-$25
  - We'll cover the other 14 registers later
- In general using register names makes code more readable than using register numbers.

Outline

Administrivia

- Project 1 spec updated
  - All changes tracked on the project spec in the change log
- Project 1 skeleton updated
  - Minor bug fix to avoid crashing when non-existent path provided
  - Just pull from GitHub again to get most up to date version
- Piazza #salmonella tag
  - Staff will apply to posts that are "half baked", as per the course syllabus

Outline

MIPS Instructions I

- Instruction syntax is rigid:

  `op dst, src1, src2`

  - 1 operator, 3 operands
    - `op` = operation name ("operator")
    - `dst` = register getting result ("destination")
    - `src1` = first register for operation ("source 1")
    - `src2` = second register for operation ("source 2")
- Keeps hardware simple via regularity

MIPS Instructions II

- One operation per instruction, at most one instruction per line
- Assembly instructions are related to C operations (=, +, −, *, /, &, |, etc.)
  - Must be, since C decomposes into assembly!
  - A single line of C may break up to several lines of MIPS

MIPS Instructions Example

- Your very first instructions!
  (assume here that the variables `a`, `b`, `c` are assigned to registers `$s1`, `$s2`, `$s3`, respectively)

MIPS Instructions Example

- Your very first instructions!
  (assume here that the variables `a`, `b`, `c` are assigned to registers `$s1`, `$s2`, `$s3`, respectively)
- Integer Addition (add)
  - C: `a = b + c`
  - MIPS: `add $s1, $s2, $s3`

MIPS Instructions Example

- Your very first instructions!
  (assume here that the variables `a`, `b`, `c` are assigned to registers `$s1`, `$s2`, `$s3`, respectively)
- Integer Addition (add)
  - C: `a = b + c`
  - MIPS: `add $s1, $s2, $s3`
- Integer Subtraction (sub)
  - C: `a = b - c`
  - MIPS: `sub $s1, $s2, $s3`

MIPS Instructions Example

- Suppose a↔$s0, b↔$s1, c↔$s2, d↔$s3, e↔$s4. Convert the following C statement to MIPS:

```
a = (b + c) - (d + e);
```

MIPS Instructions Example

- Suppose a↔$s0, b↔$s1, c↔$s2, d↔$s3, e↔$s4. Convert the following C statement to MIPS:

```
a = (b + c) - (d + e);
```

```
add $t1, $s3, $s4
add $t2, $s1, $s2
sub $s0, $t2, $t1
```

- Notice: order of instructions must follow order of operations in C

Comments in MIPS

- Comments in MIPS follow hash marks (#) until the end of line
  - Improves readability and helps you keep track of variables/registers
  - MIPS is in NO way self-documenting, make good use of comments

```
add $t1, $s3, $s4 # $t1=d+e
add $t2, $s1, $s2 # $t2=b+c
sub $s0, $t2, $t1 # a=(b+c)-(d+e)
```

The Zero Register

- Zero appears so often in code and is so useful that we give it its own register
- Register zero (`$0` or `$zero`) always has the value 0, and cannot be changed
  - Any instruction which writes to `$0` has no effect
- Example uses:
  - `add $s2, $0,  $0 # c=0`
  - `add $s0, $s1, $0 # a=b`

Immediates

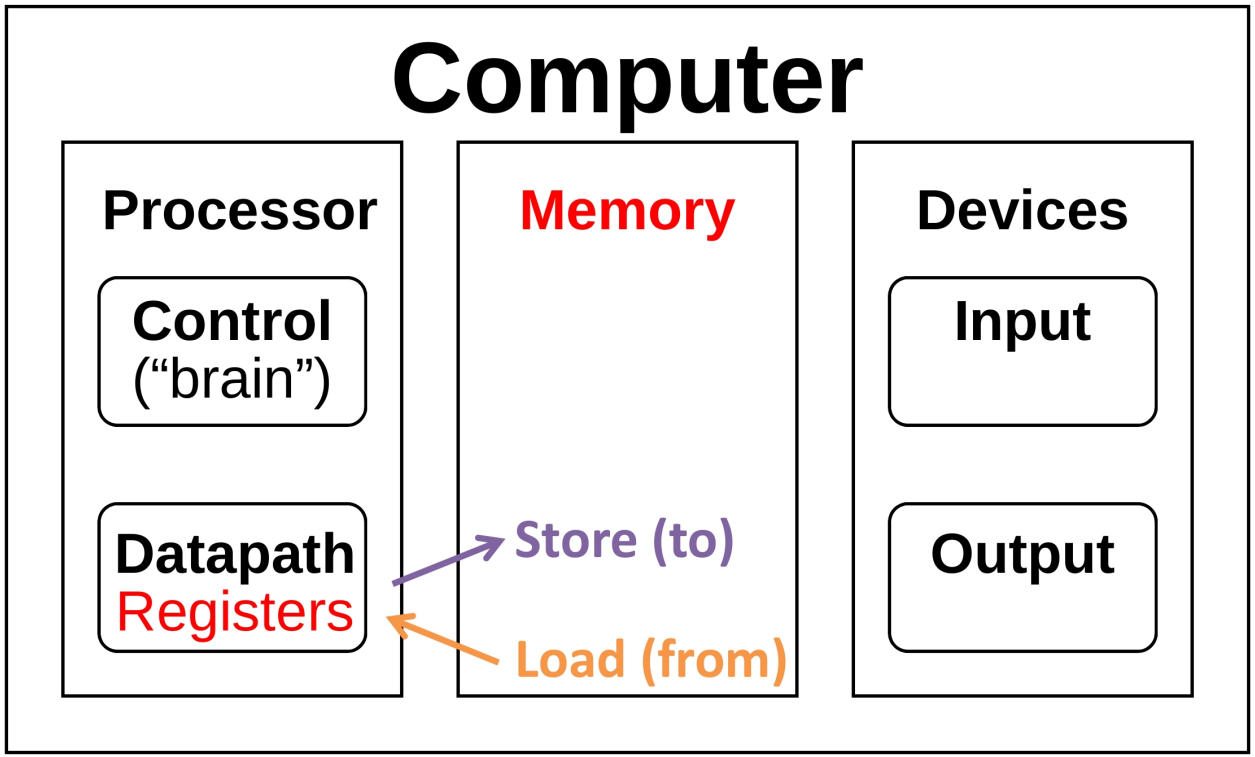
- Numerical constants are called *immediates*
- Separate instruction synstax for immediates

  `opi dst, src, imm`

  - Operation names end with 'i', replace second source register with an immediate
- Example uses:
  - `addi $s0, $s1, 5 # a=b+5`
  - `addi $s2, $s2, 1 # c++`
- Why no `subi` instruction?

Five Components of a Computer

- Data transfer instructions move data between registers (datapath) and memory
  - Allows us to fetch and store operands in memory



Data Transfer

- C variables (can) map onto registers; What about large data structures like arrays?
  - Use memory! But how?
- MIPS instructions only operate on registers

Data Transfer

- C variables (can) map onto registers; What about large data structures like arrays?
  - Use memory! But how?
- MIPS instructions only operate on registers
- Specialized *data transfer instructions* move data between registers and memory
  - Store: register TO memory
  - Load: register FROM memory

Data Transfer

- Instruction syntax for data transfer:

  `op reg, offset(base_addr)`

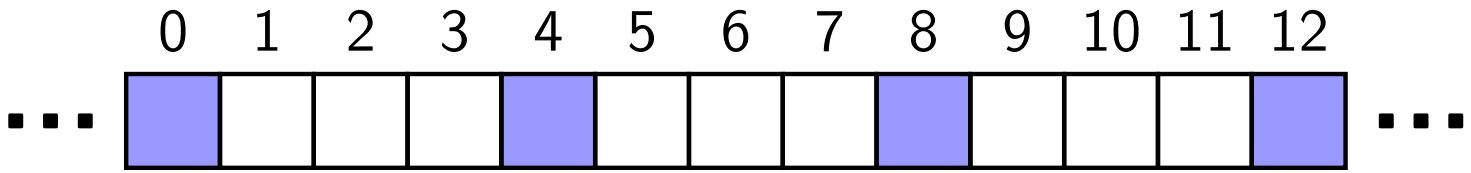
  - `op` = operation name ("operator")
  - `reg` = register for operation source or destination
  - `base_addr` = register with pointer to memory ("base address")
  - `offset` = address offset (immediate) in bytes ("offset")
- Accesses memory at address `base_addr + offset`
- **Reminder:** A register holds a word of raw data (no type) – be sure to use registers and offsets that point to valid memory addresses

Memory is Byte-Addressed

- What was the smallest data type in C?

Memory is Byte-Addressed

- What was the smallest data type in C?
  - A `char`, which was a *byte* (8 bits)
  - Everything in multiples of 8 bits (e.g. 1 word = 4 bytes)
- Memory addresses are indexed by *bytes*, not words
- Word addresses are 4 bytes apart
  - Word addr is the same as left-most byte
  - Addrs must be multiples of 4 to be *word-aligned*
- Pointer arithmetic not done for you in assembly
  - Must take data size into account manually



Data Transfer Instructions

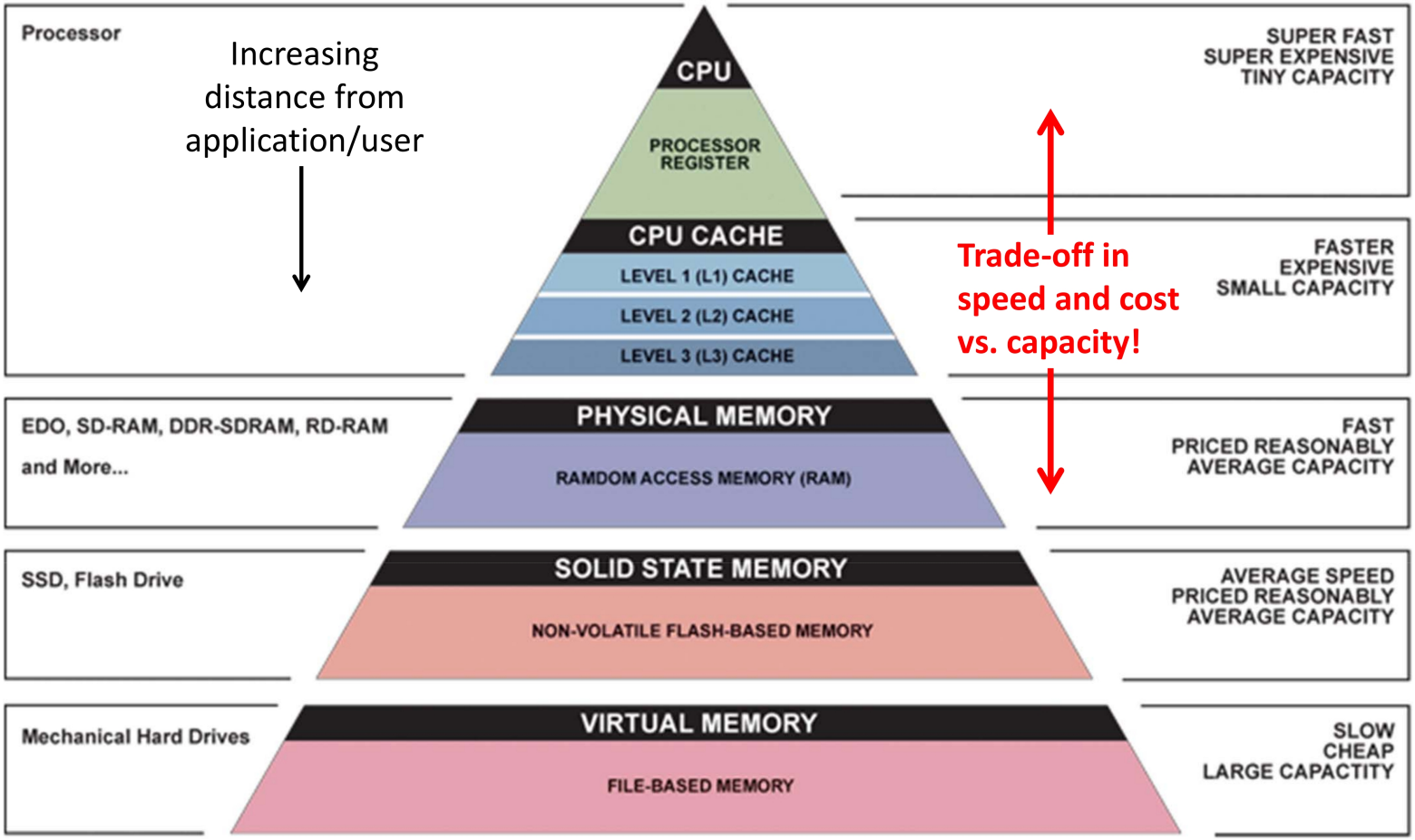
- Load Word (`lw`)
  - Takes data at address `base_addr + offset` FROM memory and places it into a register
- Store Word (`sw`)
  - Takes data in register and stores it TO memory at address `base_addr + offset`
- Example Usage:

```
# addr of int B[] -> $s3, a -> $s0
lw  $t0,12($s3)  # $t0=B[3]
add $t0,$s0,$t0  # $t0=B[3]+a
sw  $t0,40($s3)  # B[10]=B[3]+a
```

Registers vs. Memory

- What if more variables than registers?
  - Keep most frequently used in registers and move the rest to memory (called *spilling to memory*)
- Why not all variables in memory?
  - Smaller is faster: registers 100-500 times faster
  - Registers are more versatile
    - In 1 arithmetic instruction: read 2 operands, perform 1 operation, 1 write
    - In 1 data transfer instruction: 1 read/write, no operation

Great Idea #3: Principle of Locality/Memory Hierarchy



Question: Which of the following is TRUE?

(blue) `add $t0,$t1,4($t2)` is valid MIPS
(green) We can address 8 GiB with MIPS 32-bit words
(purple) `offset` must be a multiple of 4 for
`lw $t0, offset($s0)` to be valid
(yellow) If MIPS halved the number of registers available, code would NOT be twice as fast

Question: Which of the following is TRUE?

(blue) `add $t0,$t1,4($t2)` is valid MIPS
(green) We can address 8 GiB with MIPS 32-bit words
(purple) `offset` must be a multiple of 4 for
`lw $t0, offset($s0)` to be valid
(yellow) If MIPS halved the number of registers available, code would NOT be twice as fast

Chars and Strings

- **Recall:** A string is just an array of characters and a `char` in C uses 8-bit ASCII
- How to retrieve from memory?

Chars and Strings

- **Recall:** A string is just an array of characters and a `char` in C uses 8-bit ASCII
- How to retrieve from memory?
- Method 1: Move words in and out of memory using bit-masking and shifting

```
lw   $s0, 0($s1)
andi $s0, $s0, 0xFF # lowest byte
```

Chars and Strings

- **Recall:** A string is just an array of characters and a `char` in C uses 8-bit ASCII
- How to retrieve from memory?
- Method 1: Move words in and out of memory using bit-masking and shifting

```
lw   $s0, 0($s1)
andi $s0, $s0, 0xFF # lowest byte
```

- Method 2: Load/store byte instructions

```
lb $s0, 0($s1)
sb $s0, 1($s1) # Addr != 0 (mod 4), but OK
```

Byte Instructions

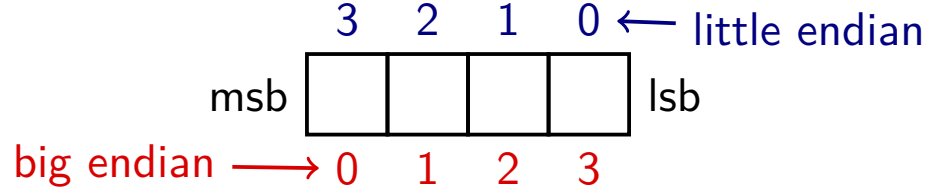
- `lb`/`sb` utilize the *least significant* byte of the register
  - On `sb`, upper 24 bits are ignored
  - On `lb`, upper 24 bits are filled via sign extension
- For example, let `0($s0) = 0x00000180`:

```
lb $s1,1($s0) # $s1=0x00000001
lb $s2,0($s0) # $s2=0xFFFFFF80
sb $s2,2($s0) # 0($s0)=0x00800180
```

- Normally you don't want to sign extend `chars`
  - Use `lbu` (load byte unsigned) instead

Endianness

- Big Endian: Most-significant byte at least address of word
  - word address = address of most significant byte
- Little Endian: Least-significant byte at least address of word
  - word address = address of least significant byte



- MIPS is bi-endian (can be implemented either way)
  - Using MARS simulator in lab, which is little endian
- Why is this confusing?
  - Data stored in reverse order that you write it out
  - 0x01020304 stored as `04 03 02 01`
- BUT, it's natural from a programming perspective
  - Moving to a larger address moves you to a more significant ("larger") byte.

Technology Break

Computer Decision Making

- In C, we had *control flow*
  - Outcomes of comparative/logical statements determine which blocks of code to execute
- In MIPS, we cannot define blocks of code; all we have are *labels*
  - Defined by text followed by a colon (e.g. `main:`) and refers to the instruction that follows
  - Generate control flow by jumping to labels
  - C has these too, but they are considered bad style

Decision Making Instructions

- Branch If Equal (`beq`)
  - `beq src1, src2, label`
  - If the contents of `src1` equal the contents of `src2`, then go to `label`
- Branch If Not Equal (`bne`)
  - `bne src1, src2, label`
  - If the contents of `src1` does NOT equal the contents of `src2`, then go to `label`
- Jump (j)
  - `j label`
  - jump to `label` no matter what

Translating an If-Else

In C:

```
if (i == j)
     a = b;   /* then */
else
     a = -b; /* else */
```

In English:

- if TRUE, execute the "then" block
- if FALSE, execute the "else" block

MIPS (beq):

```
# i->$s0, j->$s1
# a->$s2, b->$s3
    beq $s0, $s1, then
else:#do not technically need
    sub $s2, $0, $s3
    j end
then:
    add $s2, $s3, $0
end:
```

Loops in MIPS

- There are three types of loops in C:
  - `while`, `do`...`while`, and `for`
  - Each can be rewritten as either of the other two
- You will examine how to write loops in MIPS in discussion
  - do-while loop now, if time
- **Key Concept:** Although there are many ways to write a loop in MIPS, the key to implementing control flow is the conditional branch

Question: Which of the following is FALSE? (Extra practice, try writing out the TRUE statement)

(blue) We can make an unconditional branch from a conditional branch instruction

(green) We can make a while-loop with just `j` (no `beq` or `bne`)

(purple) We can make a for-loop without using `j`

(yellow) An if-else clause written with `beq` can be written in the same number of lines with `bne`

Question: Which of the following is FALSE? (Extra practice, try writing out the TRUE statement)

(blue) We can make an unconditional branch from a conditional branch instruction

(green) We can make a while-loop with just `j` (no `beq` or `bne`)

(purple) We can make a for-loop without using `j`

(yellow) An if-else clause written with `beq` can be written in the same number of lines with `bne`

MIPS Green Sheet

- Contains MIPS instructions and lots of other useful information
  - `http://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf`
  - Hard copy in textbook (will be provided on exams)
- Inspired by the IBM 360 "Green Card" from the late 1960's and 1970's
  - `http://en.wikipedia.org/wiki/Green_card_(IBM/360)`

And in Conclusion

- Computers understand the *instructions* of their *ISA*
- RISC Design Principles
  - Smaller is faster, keep it simple
- MIPS Registers: `$s0-$s7`, `$t0-$t9`, `$0`
- MIPS Instructions
  - Arithmetic: `add`, `sub`, `addi`
  - Data transfer: `lw`, `sw`, `lb`, `sb`, `lbu`
  - Branching: `beq`, `bne`, `j`
- Memory is byte-addressed

Outline

Bonus Slides

We will likely not have time to cover these slides in lecture, but you are still responsible for the material presented within them. They have been put together in such a way as to be easily readable even without a live lecturer presenting them.

C to MIPS Practice

- Let's put all of our new MIPS knowledge to use in an example: "Fast String Copy"
- C code is as follows:

```
/* Copy string from p to q. */
char *p, *q;
while(*q++ = *p++);
```

- What do we know about its structure?
  - Single `while` loop
  - Exit condition is an equality test

C to MIPS Practice

- Start with code skeleton:

```
# copy string p to q
# p->$s0, q->$s1 (pointers)

Loop:                          # $t0 = *p
                               # *q = $t0
                               # p = p + 1
                               # q = q + 1
                               # if *p == 0, go to Exit
        j Loop                 # go to loop
```

C to MIPS Practice

- Fill in code according to comments:

```
# copy string p to q
# p->$s0, q->$s1 (pointers)

Loop: lb   $t0, 0($s0)   # $t0 = *p
      sb   $t0, 0($s1)   # *q = $t0
      addi $s0, $s0, 1   # p = p + 1
      addi $s1, $s1, 1   # q = q + 1
      beq  $t0, $0, Exit # if *p == 0, go to Exit
      j Loop             # go to loop
Exit:
```

C to MIPS Practice

- Finished code:

```
# copy string p to q
# p->$s0, q->$s1 (pointers)

Loop: lb   $t0, 0($s0)    # $t0 = *p
      sb   $t0, 0($s1)    # *q = $t0
      addi $s0, $s0, 1    # p = p + 1
      addi $s1, $s1, 1    # q = q + 1
      beq  $t0, $0, Exit  # if *p == 0, go to Exit
      j Loop              # go to loop
Exit: #N chars in p => N*6 instructions
```

C to MIPS Practice

- Alternate code using bne:

```
# copy string p to q
# p->$s0, q->$s1 (pointers)

Loop: lb   $t0, 0($s0)    # $t0 = *p
      sb   $t0, 0($s1)    # *q = $t0
      addi $s0, $s0, 1    # p = p + 1
      addi $s1, $s1, 1    # q = q + 1
      bne  $t0, $0, Loop # if *p == 0, go to Exit
                          # go to loop
      #N chars in p => N*6 instructions
```

MIPS Arithmetic Instructions

- The following commands place results in the special registers HI and LO
  - Access these values with "move from HI" (`mfhi dst`) and "move from LO" (`mflo dst`)
- **Multiplication** (`mult`)
  - `mult src1, src2`
  - `src1 * src2`: lower 32-bits in LO, upper 32-bits in HI
- **Division** (`div`)
  - `div src1, src2`
  - `src1 / src2`: puts quotient in LO, remainder in HI

MIPS Arithmetic Instructions

- Example:

```
# mod using div: $s2 = $s0 mod $s1
mod:
    div  $s0, $s1 # LO = $s0 / $s1
    mfhi $s2      # HI = $s0 mod $s1
```

Arithmetic Overflow

- **Recall:** Overflow occurs when there is a "mistake" in arithmetic due to the limitied precision in computers
  - i.e. not enough bits to represent answer
- MIPS detects overflow (*throws errors*)
  - Arithmetic unsigned instructions ignore overflow

Detects Overflow	Ignores Overflow
`add dst, src1, src2`	`addu dst, src1, src2`
`addi dst, src1, imm`	`addiu dst, src1, imm`
`sub dst, src1, src2`	`subu dst, src1, src2`

Arithmetic Overflow

- Example (Recall that 0x80000000 is the most negative number):

```
# $s0=0x80000000, $s1=0x1
add $t0,$s0,$s0  # overflow (error)
addu $t1,$s0,$s0 # $t1 = 0
addi $t2,$s0,-1  # overflow (error)
addiu $t2,$s0,-1 # $t3=0x7FFFFFFF
sub $t4,$s0,$s1  # overflow (error)
subu $t5,$s0,$s1 # $t5=0x7FFFFFFF
```

MIPS Bitwise Instructions

Note: a → $s1, b → $s2, c → $s3

Instruction	C	MIPS
And	`a = b & c`	`and $s1,$s2,$s3`
And Immediate	`a = b & 0x1`	`andi $s1,$s2,0x1`
Or	`a = b | c`	`or $s1,$s2,$s3`
Or Immediate	`a = b | 0x5`	`ori $s1,$s2,0x5`
Not Or	`a = ~(b | c)`	`not $s1,$s2,$s3`
Exclusive Or	`a = b ^ c`	`xor $s1,$s2,$s3`
Exclusive Or Immediate	`a = b ^ 0xF`	`xori $s1,$s2,0xF`

Shifting Instructions

- In binary, shifting an unsigned number to the left is the same as multiplying by the corresponding power of 2
  - Shifting operations are faster
  - Does not (quite) work with shifting right/division
- *Logical Shift*: Add zeros as you shift
- *Arithmetic Shift*: Sign extend as you shift
  - Only applies when you shift right (preserves sign)
- Can shift by immediate or value in a register

Shifting Instructions

Instruction Name	MIPS
Shift Left Logical	`sll $s1,$s2,1`
Shift Left Logical Variable	`sllv $s1,$s2,$s3`
Shift Right Logical	`srl $s1,$s2,2`
Shift Right Logical Variable	`srlv $s1,$s2,$s3`
Shift Right Arithmetic	`sra $s1,$s2,3`
Shift Right Arithmetic Variable	`srav $s1,$s2,$s3`

- When using immediate, only values 0-31 are accepted
- When using variable, only 5 least significant bits are used (read as unsigned)

Shifting Instructions

```
# sample calls to shift instructions
addi $t0, $0, -256 # $t0=0xFFFFFF00
sll  $s0, $t0, 3   # $s0=0xFFFFF800
srl  $s1, $t0, 8   # $s1=0x00FFFFFF
sra  $s2, $t0, 8   # $s2=0xFFFFFFFF

addi $t1, $0,  -22 # $t1=0xFFFFFFEA
                   # low 5: 0b01010
sllv $s3, $t0, $t1 # $s3=0xFFFC0000
# same as sll $s3, $t0, 10
```

Shifting Instructions

- Another Example:

```
# lb using lw: lb $s1, 1($s0)
lw   $s1, 0($s0)       # get word
andi $s1, $s1, 0xFF00 # get 2nd byte
srl  $s1, $s1, 8       # shift into lowest
```

Shifting Instructions

- Yet Another Example:

```
# sb using sw: sb $s1, 3($s0)
lw   $t0, 0($s0)         # get current word
andi $t0, $t0, 0xFFFFFF # zero top byte
sll  $t1, $s1, 24        # shift into highest
or   $t0, $t0, $t1       # combine
sw   $t0, 0($s0)         # store back
```

Shifting Instructions

- Extra for Experience:
  - Rewrite the two preceding examples to be more general
  - Assume that the byte offset (e.g. 1 and 3 in the examples) is contained in `$s2`
- Hint:
  - The variable shift instructiosn will come in handy
  - Remember, the offset can be negative