Instructor
Paul Pearce

inst.eecs.berkeley.edu/~cs61c

UCB CS61C : Machine Structures

Lecture 9 – Introduction to MIPS Procedures II, Logical Ops, & Instruction Format I

2010-07-06

APPLE ACKNOWLEDGES RECEPTION WOES

Apple announced that there is indeed a problem with iPhone reception, and that it's a software glitch reporting TOO MANY bars. Test your code!

www.apple.com/pr/library/2010/07/02appleletter.html

Pearce, Summer 2010 © UCB

"And in Review…"

- In order to help the conditional branches make decisions concerning inequalities, we introduce a single instruction: "Set on Less Than" called `slt`, `slti`, `sltu`, `sltiu`
- Functions called with `jal`, return with `jr $ra`.
- The stack is your friend: Use it to save anything you need. Just leave it the way you found it!
- Instructions we know so far…

  Arithmetic: `add, addi, sub, addu, addiu, subu`

  Memory: `lw, sw, lb, sb`

  Decision: `beq, bne, slt, slti, sltu, sltiu`

  Unconditional Branches (Jumps): `j, jal, jr`
- Registers we know so far
  - All of them! Today we learn calling CONVENTIONS!

Register Conventions (1/4)

- CalleR: the calling function
- CalleE: the function being called
- When callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged.
- Register Conventions: A set of generally accepted rules as to which registers will be unchanged after a procedure call (`jal`) and which may be changed.

Pearce, Summer 2010 © UCB

Register Conventions (2/4) – saved

- `$0`: No Change. Always `0`.
- `$s0-$s7`: Restore if you change. Very important, that's why they're called saved registers. If the callee changes these in any way, it must restore the original values before returning.
- `$sp`: Restore if you change. The stack pointer must point to the same place before and after the `jal` call, or else the caller won't be able to restore values from the stack.
- HINT -- All saved registers start with S!

Pearce, Summer 2010 © UCB

Register Conventions (3/4) – volatile

- `$ra`: Can Change. The `jal` call itself will change this register. Caller needs to save on stack if nested call.
- `$v0-$v1`: Can Change. These will contain the new returned values.
- `$a0-$a3`: Can change. These are volatile argument registers. Caller needs to save if they are needed after the call.
- `$t0-$t9`: Can change. That's why they're called temporary: any procedure may change them at any time. Caller needs to save if they'll need them afterwards.

Pearce, Summer 2010 © UCB

Register Conventions (4/4)

- What do these conventions mean?
  - If function R calls function E, then function R must save any temporary registers that it may be using onto the stack before making a `jal` call.
  - Function E must save any S (saved) registers it intends to use before garbling up their values
- Remember: caller/callee need to save only temporary/saved registers they are using, not all registers.

Pearce, Summer 2010 © UCB

Parents leaving for weekend analogy (1/5)

- Parents (`main`) leaving for weekend
- They (`caller`) give keys to the house to kid (`callee`) with the rules (`calling conventions`):
  - You can trash the temporary room(s), like the den and basement (`registers`) if you want, we don't care about it
  - BUT you'd better leave the rooms (`registers`) that we want to save for the guests untouched. "these rooms better look the same when we return!"
- Who hasn't heard this in their life?

Pearce, Summer 2010 © UCB

Parents leaving for weekend analogy (2/5)

- Kid now "owns" rooms (`registers`)
- Kid wants to use the saved rooms for a wild, wild party (`computation`)
- What does kid (`callee`) do?
  - Kid takes what was in these rooms and puts them in the garage (`memory`)
  - Kid throws the party, trashes everything (except garage, who ever goes in there?)
  - Kid restores the rooms the parents wanted saved after the party by replacing the items from the garage (`memory`) back into those saved rooms

Parents leaving for weekend analogy (3/5)

- Same scenario, except before parents return and kid replaces saved rooms…
- Kid (`callee`) has left valuable stuff (`data`) all over.
  - Kid's friend (`another callee`) wants the house for a party when the kid is away
  - Kid knows that friend might trash the place destroying valuable stuff!
  - Kid remembers rule parents taught and now becomes the "heavy" (`caller`), instructing friend (`callee`) on good rules (`conventions`) of house.

Parents leaving for weekend analogy (4/5)

- If kid had data in temporary rooms (which were going to be trashed), there are three options:
  - Move items directly to garage (`memory`)
  - Move items to saved rooms whose contents have already been moved to the garage (`memory`)
  - Optimize lifestyle (`code`) so that the amount you've got to shlep stuff back and forth from garage (`memory`) is minimized.
    - Mantra: "Minimize register footprint"
- Otherwise: "Dude, where's my data?!"

Pearce, Summer 2010 © UCB

Parents leaving for weekend analogy (5/5)

- Friend now "owns" rooms (`registers`)
- Friend wants to use the saved rooms for a wild, wild party (`computation`)
- What does friend (`callee`) do?
  - Friend takes what was in these rooms and puts them in the garage (`memory`)
  - Friend throws the party, trashes everything (except garage)
  - Friend restores the rooms the kid wanted saved after the party by replacing the items from the garage (`memory`) back into those saved rooms

Pearce, Summer 2010 © UCB

Administrivia

- Homework 3 due tonight
- Start preparing for midterm!
  - Friday, July 16th. 9:30am-12:30pm. Room: 100 Lewis
- Interested in a review session later this week or early next week?
- First project over:
  - How'd it go?
- Reference slides
  - Don't forget about them!
- Anything else?

Pearce, Summer 2010 © UCB

Bitwise Operations

- So far, we've done arithmetic (**`add`**, **`sub`**,**`addi`**), mem access (**`lw`** and **`sw`**), & branches and jumps.
- All of these instructions view contents of register as a single quantity (e.g., signed or unsigned int)
- New Perspective: View register as 32 raw bits rather than as a single 32-bit number
  - Since registers are composed of 32 bits, wish to access individual bits (or groups of bits) rather than the whole.
- Introduce two new classes of instructions
  - Logical & Shift Ops

Pearce, Summer 2010 © UCB

Logical Operators (1/3)

- Two basic logical operators:
  - AND: outputs 1 only if all inputs are 1
  - OR: outputs 1 if at least one input is 1
- Truth Table: standard table listing all possible combinations of inputs and resultant output

A	B	A AND B	A OR B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Pearce, Summer 2010 © UCB

Logical Operators (2/3)

- Logical Instruction Syntax:

  1 2,3,4

  - where

    1) operation name

    2) register that will receive value

    3) first operand (register)

    4) second operand (register) or immediate (numerical constant)

- In general, can define them to accept > 2 inputs, but in the case of MIPS assembly, these accept exactly 2 inputs and produce 1 output
  - Again, rigid syntax, simpler hardware

Pearce, Summer 2010 © UCB

Logical Operators (3/3)

- Instruction Names:
  - `and, or`: Both of these expect the third argument to be a register
  - `andi, ori`: Both of these expect the third argument to be an immediate
- MIPS Logical Operators are all bitwise, meaning that bit 0 of the output is produced by the respective bit 0's of the inputs, bit 1 by the bit 1's, etc.
  - C: Bitwise AND is `&` (e.g., `z = x & y;`)
  - C: Bitwise OR is `|` (e.g., `z = x | y;`)

Pearce, Summer 2010 © UCB

Uses for Logical Operators (1/3)

- Note that **`and`**ing a bit with `0` produces a `0` at the output while **`and`**ing a bit with `1` produces the original bit.
- This can be used to create a mask.
  - Example:

```
      1011 0110 1010 0100 0011 1101 1001 1010
mask: 0000 0000 0000 0000 0000 1111 1111 1111
```

  - The result of **`and`**ing these:

```
      0000 0000 0000 0000 0000 1101 1001 1010
```

mask last 12 bits

Pearce, Summer 2010 © UCB

Uses for Logical Operators (2/3)

- The second bitstring in the example is called a mask. It is used to isolate the rightmost 12 bits of the first bitstring by masking out the rest of the string (e.g. setting to all `0`s).
- Thus, the **`and`** operator can be used to set certain portions of a bitstring to `0`s, while leaving the rest alone.
  - In particular, if the first bitstring in the above example were in **`$t0`**, then the following instruction would mask it:

```
andi   $t0,$t0,0xFFF
```

Pearce, Summer 2010 © UCB

Uses for Logical Operators (3/3)

- Similarly, note that **`or`**ing a bit with **`1`** produces a **`1`** at the output while `or`ing a bit with **`0`** produces the original bit.
- Often used to force certain bits to **`1`**s.
  - For example, if **`$t0`** contains **`0x12345678`**, then after this instruction:

```
ori  $t0, $t0, 0xFFFF
```

… **`$t0`** will contain **`0x1234FFFF`**

    - (i.e., the high-order 16 bits are untouched, while the low-order 16 bits are forced to **`1`**s).

Pearce, Summer 2010 © UCB

61C Levels of Representation (abstractions)

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g.,MIPS)

```
lw   $t0, 0($s2)
lw   $t1, 4($s2)
sw   $t1, 0($s2)
sw   $t0, 4($s2)
```

Assembler

Machine Language Program (MIPS)

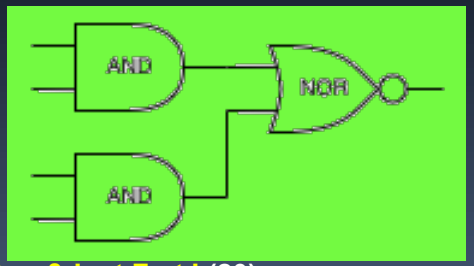
```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Pearce, Summer 2010 © UCB

Overview – Instruction Representation

- Big idea: stored program
  - consequences of stored program
- Instructions as numbers
- Instruction encoding
- MIPS instruction format for Add instructions
- MIPS instruction format for Immediate, Data transfer instructions and jumps NEXT TIME

Pearce, Summer 2010 © UCB

Big Idea: Stored-Program Concept

- Computers built on 2 key principles:
  - Instructions are represented as bit patterns - can think of these as numbers.
  - Therefore, entire programs can be stored in memory to be read or written just like data.
- Simplifies SW/HW of computer systems:
  - Memory technology for data also used for programs

Pearce, Summer 2010 © UCB

Consequence #1: Everything Addressed

- Since all instructions and data are stored in memory, everything has a memory address: instructions, data words
  - both branches and jumps use these
- C pointers are just memory addresses: they can point to anything in memory
  - Unconstrained use of addresses can lead to nasty bugs; up to you in C; limits in Java
- One register keeps address of instruction being executed: **"Program Counter" (PC)**
  - Basically a pointer to memory: Intel calls it Instruction Address Pointer, a better name

Pearce, Summer 2010 © UCB

Consequence #2: Everything Addressed

- Programs are distributed in binary form
  - Programs bound to specific instruction set
  - Different version for (old) Macintoshes and PCs
- New machines want to run old programs ("binaries") as well as programs compiled to new instructions
- Leads to "backward compatible" instruction set evolving over time
- Selection of Intel 8086 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set (Pentium 4); could still run program from 1981 PC today

Pearce, Summer 2010 © UCB

Instructions as Numbers (1/2)

- Currently all data we work with is in words (32-bit blocks):
  - Each register is a word.
  - `lw` and `sw` both access memory one word at a time.
- So how do we represent instructions?
  - Remember: Computer only understands 1s and 0s, so "`add $t0,$0,$0`" is meaningless.
  - MIPS wants simplicity: since data is in words, make instructions be words too

Pearce, Summer 2010 © UCB

Instructions as Numbers (2/2)

- One word is 32 bits, so divide instruction word into "fields".
- Each field tells processor something about instruction.
- We could define different fields for each instruction, but MIPS is based on simplicity, so define 3 basic types of instruction formats:
  - R-format
  - I-format
  - J-format

Pearce, Summer 2010 © UCB

Instruction Formats

- I-format: used for instructions with immediates, `lw` and `sw` (since offset counts as an immediate), and branches (`beq` and `bne`),
  - (but not the shift instructions; later)
- J-format: used for `j` and `jal`
- R-format: used for all other instructions
- It will soon become clear why the instructions have been partitioned in this way.

Pearce, Summer 2010 © UCB

R-Format Instructions (1/5)

- Define "fields" of the following number of bits each: 6 + 5 + 5 + 5 + 5 + 6 = 32

31 0

6	5	5	5	5	6

- For simplicity, each field has a name:

opcode	rs	rt	rd	shamt	funct

- Important: On these slides and in book, each field is viewed as a 5- or 6-bit unsigned integer, not as part of a 32-bit integer.
  - Consequence: 5-bit fields can represent any number 0-31, while 6-bit fields can represent any number 0-63.

Pearce, Summer 2010 © UCB

R-Format Instructions (2/5)

- What do these field integer values tell us?
  - `opcode`: partially specifies what instruction it is
    - Note: This number is equal to `0` for all R-Format instructions.
  - `funct`: combined with `opcode`, this number exactly specifies the instruction
- Question: Why aren't `opcode` and `funct` a single 12-bit field?
  - We'll answer this later.

Pearce, Summer 2010 © UCB

R-Format Instructions (3/5)

- More fields:
  - `rs` (Source Register): *generally* used to specify register containing first operand
  - `rt` (Target Register): *generally* used to specify register containing second operand (note that name is misleading)
  - `rd` (Destination Register): *generally* used to specify register which will receive result of computation

Pearce, Summer 2010 © UCB

R-Format Instructions (4/5)

- Notes about register fields:
  - Each register field is exactly 5 bits, which means that it can specify any unsigned integer in the range 0-31. Each of these fields specifies one of the 32 registers by number.
  - The word "generally" was used because there are exceptions that we'll see later. E.g.,
    - **`mult`** and **`div`** have nothing important in the **`rd`** field since the dest registers are **`hi`** and **`lo`**
    - **`mfhi`** and **`mflo`** have nothing important in the **`rs`** and **`rt`** fields since the source is determined by the instruction (see COD)

R-Format Instructions (5/5)

- Final field:
  - `shamt`: This field contains the amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is useless, so this field is only 5 bits (so it can represent the numbers 0-31).
  - This field is set to 0 in all but the shift instructions.
- For a detailed description of field usage for each instruction, see green insert in COD (You will be given a copy for all exams)

Pearce, Summer 2010 © UCB

Peer Instruction

```
r: ...      # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
   ...      ### PUSH REGISTER(S) TO STACK?
   jal e    # Call e
   ...      # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
   jr $ra   # Return to caller of r

e: ...      # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
   jr $ra   # Return to r
```

What does `r` have to push on the stack before "`jal e`"?

a) 1 of ($s0,$sp,$v0,$t0,$a0,$ra)
b) 2 of ($s0,$sp,$v0,$t0,$a0,$ra)
c) 3 of ($s0,$sp,$v0,$t0,$a0,$ra)
d) 4 of ($s0,$sp,$v0,$t0,$a0,$ra)
e) 5 of ($s0,$sp,$v0,$t0,$a0,$ra)

Pearce, Summer 2010 © UCB

"And in Conclusion…"

- Register Conventions: Each register has a purpose and limits to its usage. Learn these and follow them, even if you're writing all the code yourself.
- Logical and Shift Instructions
  - Operate on bits individually, unlike arithmetic, which operate on entire word.
  - Use to isolate fields, either by masking or by shifting back and forth.
  - Use shift left logical, `sll`, for multiplication by powers of 2
  - Use shift right logical, `srl`, for division by powers of 2 of unsigned numbers (`unsigned int`)
  - Use shift right arithmetic, `sra`, for division by powers of 2 of signed numbers (`int`)
- New Instructions:
  `and, andi, or, ori, sll, srl, sra`

ALSO in conclusion…

- Simplifying MIPS: Define instructions to be same size as data word (one word) so that they can use the same memory (compiler can use lw and sw).
- Computer actually stores programs as a series of these 32-bit numbers.
- MIPS Machine Language Instruction: 32 bits representing a single instruction

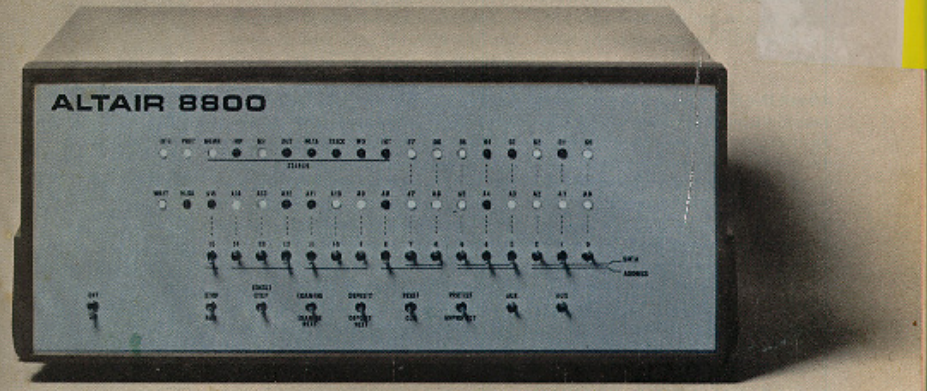
Pearce, Summer 2010 © UCB

Popular Electronics
JANUARY, 1975

PE TESTED

EXCLUSIVE!

ALTAIR 8800
The most powerful minicomputer project ever presented—can be built for under $400



BY H. EDWARD ROBERTS AND WILLIAM YATES

THE era of the computer in every home—a favorite topic among science-fiction writers—has arrived! It's made possible by the POPULAR ELECTRONICS/MITS Altair 8800, a full-blown computer that can hold its own against sophisticated minicomputers now on the market. And it doesn't cost several thousand dollars. In fact, it's in a color TV-receiver's price class—under $400 for a complete kit.

The Altair 8800 is not a "demonstrator" or souped-up calculator. It is the most powerful computer ever presented as a construction project in any electronics magazine. In many ways, it represents a revolutionary development in electronic design and thinking.

The Altair 8800 is a parallel 8-bit word/16-bit address computer with an instruction cycle time of 2 μs. Its central processing unit is a new LSI chip that is many times more powerful than previous IC processors. It can accommodate 256 inputs and 256 outputs, all directly addressable, and has 78 basic machine instructions (as compared with 40 in the usual minicomputer). This means that you can write an extensive and detailed program. The basic computer has 256 words of memory, but it can be economically expanded for 65,000 words. Thus, with full expansion, up to 65,000 subroutines can all be going at the same time.

The basic computer is a complete system. The program can be entered via switches located on the front panel, providing a LED readout in binary format. The very-low-cost terminal presented in POPULAR ELECTRONICS last month can also be used.

PROCESSOR DESCRIPTION

Processor: 8 bit parallel
Max. memory: 65,000 words (all directly addressable)
Instruction cycle time: 2 μs (min.)
Inputs and outputs: 256 (all directly addressable)
Number of basic machine instructions: 78 (181 with variants)
Add/substract time: 2 μs
Number of subroutine levels: 65,000
Interrupt structure: 8 hardwire vectored levels plus software levels
Number of auxiliary registers: 8 plus stack pointer, program counter and accumulator
Memory type: semiconductor (dynamic or static RAM, ROM, PROM)
Memory access time: 850 ns static RAM; 420 or 150 ns dynamic Ram

Bonus slides

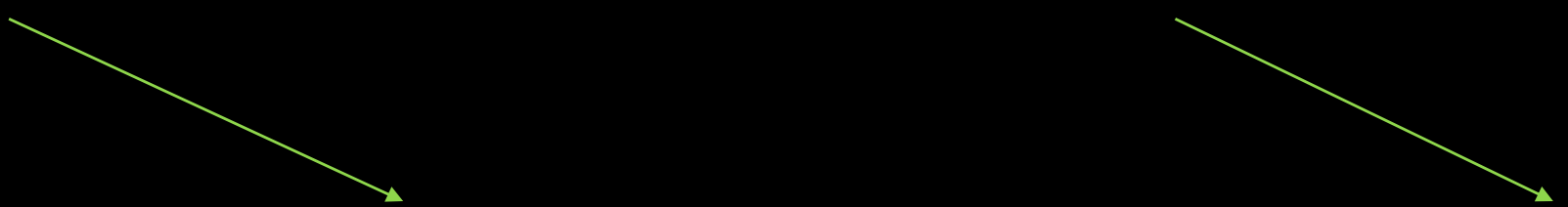
- These are extra slides that used to be included in lecture notes, but have been moved to this, the “bonus” area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

Bonus

Pearce, Summer 2010 © UCB

Shift Instructions (review) (1/4)

- Move (shift) all the bits in a word to the left or right by a number of bits.
  - Example: shift right by 8 bits



  - Example: shift left by 8 bits

Pearce, Summer 2010 © UCB

Shift Instructions (2/4)

- Shift Instruction Syntax:

  1 2,3,4

  …where

  1) operation name
  2) register that will receive value
  3) first operand (register)
  4) shift amount (constant < 32)

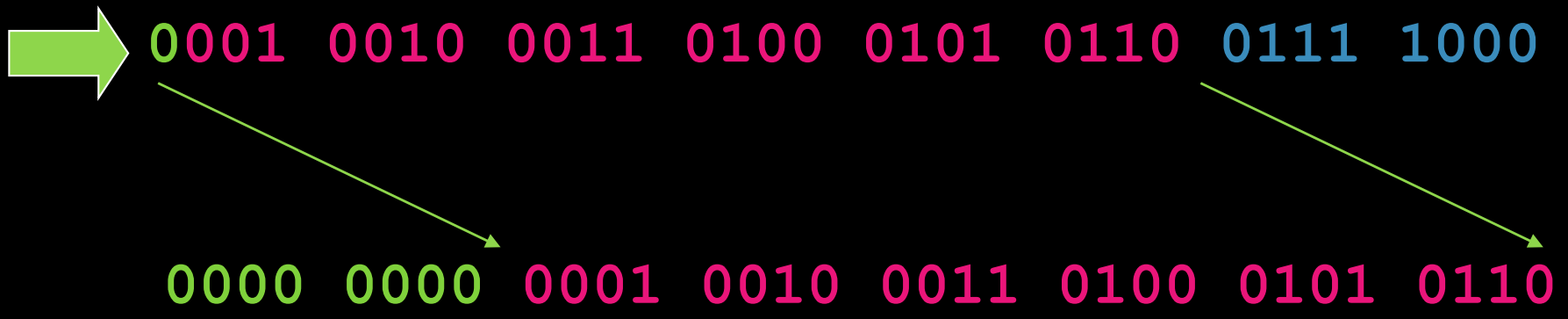
- MIPS shift instructions:

  1. `sll` (shift left logical): shifts left and fills emptied bits with 0s
  2. `srl` (shift right logical): shifts right and fills emptied bits with 0s
  3. `sra` (shift right arithmetic): shifts right and fills emptied bits by sign extending

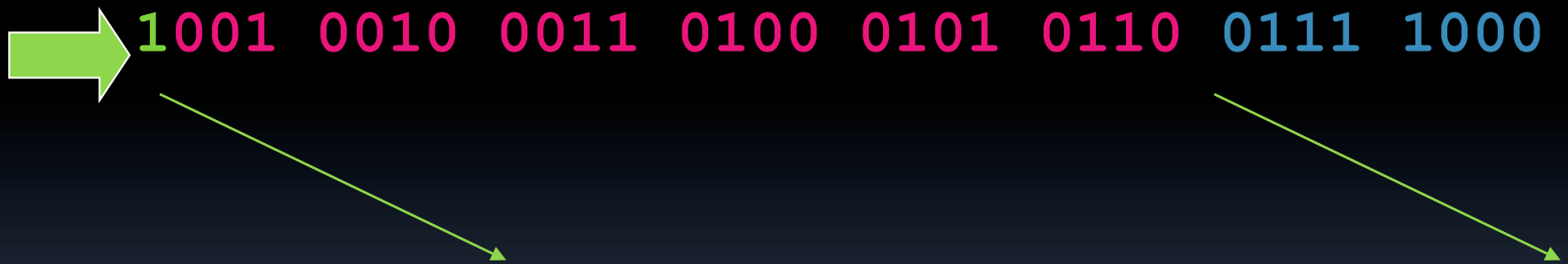
Pearce, Summer 2010 © UCB

Shift Instructions (3/4)

- Example: shift right arithmetic by 8 bits



- Example: shift right arithmetic by 8 bits



Pearce, Summer 2010 © UCB

Shift Instructions (4/4)

- Since shifting may be faster than multiplication, a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:

  `a *= 8;` (in C)

  would compile to:

  `sll $s0,$s0,3` (in MIPS)

- Likewise, shift right to divide by powers of 2 (rounds towards -∞)
  - remember to use `sra`

Pearce, Summer 2010 © UCB

Example: Fibonacci Numbers 1/8

- The Fibonacci numbers are defined as follows: F(n) = F(n – 1) + F(n – 2), F(0) and F(1) are defined to be 1
- In scheme, this could be written:

```
(define (Fib n)
  (cond  ((= n 0) 1)
         ((= n 1) 1)
         (else (+(Fib (- n 1))
                  (Fib (- n 2))))))
```

Pearce, Summer 2010 © UCB

Example: Fibonacci Numbers 2/8

- Rewriting this in C we have:

```
int fib(int n) {
  if(n == 0) { return 1; }
  if(n == 1) { return 1; }
  return (fib(n - 1) + fib(n - 2));
}
```

Pearce, Summer 2010 © UCB

Example: Fibonacci Numbers 3/8

- Now, let's translate this to MIPS!
- You will need space for three words on the stack
- The function will use one `$s` register, `$s0`
- Write the Prologue:

```
fib:
addi $sp, $sp, -12  # Space for three words
sw $ra, 8($sp)      # Save return address
sw $s0, 4($sp)      # Save s0
```

Pearce, Summer 2010 © UCB

Example: Fibonacci Numbers 4/8

- **Now write the Epilogue:**

```
fin:
lw $s0, 4($sp)          # Restore $s0
lw $ra, 8($sp)          # Restore return address
addi $sp, $sp, 12       # Pop the stack frame
jr $ra                  # Return to caller
```

Example: Fibonacci Numbers 5/8

- Finally, write the body. The C code is below. Start by translating the lines indicated in the comments

```
int fib(int n) {
 if(n == 0) { return 1; } /*Translate Me!*/
 if(n == 1) { return 1; } /*Translate Me!*/
 return (fib(n - 1) + fib(n - 2));
 }
```

```
addi $v0, $zero, 1      # $v0 = 1
beq  $a0, $zero, fin    #
addi $t0, $zero, 1      # $t0 = 1
beq  $a0, $t0, fin      #
Continued on next slide.  .  .
```

Example: Fibonacci Numbers 6/8

- Almost there, but be careful, this part is tricky!

```
int fib(int n) {
   . . .
   return (fib(n - 1) + fib(n - 2));
}
```

```
addi $a0, $a0, -1
sw $a0, 0($sp)
jal fib
lw $a0, 0($sp)
addi $a0, $a0, -1
```

```
# $a0 = n - 1
#  Need $a0 after jal
#  fib(n - 1)
#  restore $a0
#  $a0 = n - 2
```

Example: Fibonacci Numbers 7/8

- Remember that $vo is caller saved!

```
int fib(int n) {
   . . .
   return (fib(n - 1) + fib(n - 2));
 }
```

```
add $s0, $v0, $zero     # Place fib(n – 1)
                        # somewhere it won't get
                        # clobbered
jal fib                 # fib(n - 2)
add $v0, $v0, $s0       # $v0 = fib(n-1) + fib(n-2)
To the epilogue and beyond.   .   .
```

Example: Fibonacci Numbers 8/8

- Here's the complete code for reference:

```
fib:  addi $sp, $sp, -12
      sw $ra, 8($sp)
      sw $s0, 4($sp)
      addi $v0, $zero, 1
      beq $a0, $zero, fin
      addi $t0, $zero, 1
      beq $a0, $t0, fin
      addi $a0, $a0, -1
      sw $a0, 0($sp)
      jal fib
      lw $a0, 0($sp)
      addi $a0, $a0, -1
      add $s0, $v0, $zero
      jal fib
      add $v0, $v0, $s0
fin:  lw $s0, 4($sp)
      lw $ra, 8($sp)
      addi $sp, $sp, 12
      jr $ra
```

Bonus Example: Compile This (1/5)

```
main() {
  int i,j,k,m; /* i-m:$s0-$s3 */
  ...
  i = mult(j,k); ...
  m = mult(i,i); ...
}
int mult (int mcand, int mlier){
  int product;
 product = 0;
  while (mlier > 0)  {
   product += mcand;
   mlier -= 1; }
  return product;
  }
```

Pearce, Summer 2010 © UCB

Bonus Example: Compile This (2/5)

```
__start:
   ...
   add $a0,$s1,$0          # arg0 = j
   add $a1,$s2,$0          # arg1 = k
   jal mult                # call mult
   add $s0,$v0,$0          # i = mult()
   ...
   add $a0,$s0,$0          # arg0 = i
   add $a1,$s0,$0          # arg1 = i
   jal mult                # call mult
   add $s3,$v0,$0          # m = mult()
   ...
   j __exit
```

```
main() {
int i,j,k,m; /* i-m:$s0-$s3 */
...
i = mult(j,k); ...
m = mult(i,i); ... }
```

Pearce, Summer 2010 © UCB

Bonus Example: Compile This (3/5)

- Notes:
  - **`main`** function ends with a jump to **`__exit`**, not **`jr $ra`**, so there's no need to save **`$ra`** onto stack
  - all variables used in **`main`** function are saved registers, so there's no need to save these onto stack

Pearce, Summer 2010 © UCB

Bonus Example: Compile This (4/5)

```
mult:
      add   $t0,$0,$0     # prod=0
Loop:
      slt   $t1,$0,$a1    # mlr > 0?
      beq   $t1,$0,Fin    # no=>Fin
      add   $t0,$t0,$a0   # prod+=mc
      addi  $a1,$a1,-1    # mlr-=1
      j     Loop          # goto Loop
Fin:
     add   $v0,$t0,$0     # $v0=prod
     jr    $ra            # return
```

```
int mult (int mcand, int mlier){
int product = 0;
while (mlier > 0)  {
 product += mcand;
 mlier -= 1; }
return product;
}
```

Pearce, Summer 2010 © UCB

Bonus Example: Compile This (5/5)

- Notes:
  - no `jal` calls are made from `mult` and we don't use any saved registers, so we don't need to save anything onto stack
  - temp registers are used for intermediate calculations (could have used s registers, but would have to save the caller's on the stack.)
  - `$a1` is modified directly (instead of copying into a temp register) since we are free to change it
  - result is put into `$v0` before returning (could also have modified `$v0` directly)

Pearce, Summer 2010 © UCB