inst.eecs.berkeley.edu/~cs61c

CS61CL : Machine Structures

Lecture #4 – MIPS I: Registers, Memory, Decisions

2009-06-30

Jeremy Huddleston

Administrivia

- Final exam @ 9am instead of 9:30
- hw1, hw2, hw3 will be graded this week
  - Your grade will be emailed by autograder

Review

- R-type instructions
  - add $s0 $s1 $s2
  - sub $t0 $t1 $t2
- Immediates
  - addi $s0 $s1 24
- Flow Control
  - j Label
  - beq $t0 $t1 Label
  - bne $s0 $s1 Label
  - slt $t0 $t1 $t2
- Memory
  - BYTE addressed
    - 1 word = 4 bytes = 32bits
  - lw $t0 4($s1) *# 4 + $s1 must be divisible by 4*
  - sb $t1 0($s2)

Review

- We can implement < using
  ```
  slt $t0 $s0 $s1
  bne $t0 $0 True
  ```
- How do we implement >, ≤ and ≥ ?
- We could add 3 more instructions, but:
  - MIPS goal: Simpler is Better
- Can we implement ≥ in one or more instructions using just `slt` and branches? `(a ≥ b) is !(a < b)`
  - What about >? `(a > b) is (b < a)`
  - What about ≤? `(a ≤ b) is !(b < a)`

C functions

```
main() {
  int i,j,k,m;
  ...
  i = mult(j,k); ...
  m = mult(i,i); ...
}

/* really dumb mult function */

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

What information must compiler/programmer keep track of?

What instructions can accomplish this?

Function Call Bookkeeping

- Registers play a major role in keeping track of information for function calls.
- Register conventions:
  - Return address `$ra`
  - Arguments `$a0, $a1, $a2, $a3`
  - Return value `$v0, $v1`
  - Local variables `$s0, $s1, … , $s7`
- The stack is also used; more later.

Instruction Support for Functions (1/4)

- Syntax for `jal` (jump and link) is same as for `j` (jump):
  ```
  jal label
  ```
- `jal` should really be called `laj` for "link and jump":
  - Step 1 (link): Save address of *next* instruction into `$ra`
    - Why next instruction? Why not current one?
  - Step 2 (jump): Jump to the given label

Instruction Support for Functions (2/4)

- Syntax for `jr` (jump register):
  ```
  jr register
  ```
- Instead of providing a label to jump to, the `jr` instruction provides a register which contains an address to jump to.
- Very useful for function calls:
  - `jal` stores return address in register `($ra)`
  - `jr $ra` jumps back to that address

Instruction Support for Functions (3/4)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

address (shown in decimal)

MIPS

```
1000
1004
1008
1012
2000
2004
```

In MIPS, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.

Instruction Support for Functions (4/4)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
   return x+y;
}
```

address (shown in decimal)

MIPS

```
1000 add  $a0,$s0,$zero   # x = a
1004 add  $a1,$s1,$zero   # y = b
1008 jal  sum      #$ra = 1012, jump to sum
1012 ...
2000 sum: add $v0,$a0,$a1
2004 jr   $ra             #nstruction
```


Nested Procedures

```
int sumSquare(int x, int y) {
   return mult(x,x)+ y;
}
```

- Something called `sumSquare`, now `sumSquare` is calling `mult`.
- So there's a value in `$ra` that `sumSquare` wants to jump back to, but this will be overwritten by the call to `mult`.
- Need to save `sumSquare` return address before call to `mult`.

Using the Stack (1/2)

- So we have a register `$sp` which always points to the last used space in the stack.
- To use stack, we decrement this pointer by the amount of space we need and then fill it with info.
- So, how do we compile this?

```
int sumSquare(int x, int y) {
  return mult(x,x)+ y;
}
```


Using the Stack (2/2)

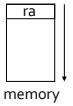
- Hand-compile

```
int sumSquare(int x, int y) {
   return mult(x,x)+ y;
}
sumSquare:
"push"  addi $sp,$sp,-8  # space on stack
        sw $ra, 4($sp)   # save ret addr
        sw $a1, 0($sp)   # save y

        add $a1,$a0,$zero   # mult(x,x)
        jal mult            # call mult

        lw $a1, 0($sp)   # restore y
        add $v0,$v0,$a1  # mult()+y

        lw $ra, 4($sp)   # get ret addr
"pop"   addi $sp,$sp,8   # restore stack
        jr $ra
```


Basic Structure of a Function

```
entry_label:
addi $sp,$sp, -framesize
sw $ra, framesize-4($sp)  # save $ra
save other regs if need be
```

Body ... (call other functions...)



Epilogue

```
restore other regs if need be
lw $ra, framesize-4($sp)  # restore $ra
addi $sp,$sp, framesize
jr $ra
```


Rules for Procedures

- Called with a `jal` instruction, returns with a `jr $ra`
- Accepts up to 4 arguments in `$a0`, `$a1`, `$a2` and `$a3`
- Return value is always in `$v0` (and if necessary in `$v1`)
- Must follow register conventions

So what are they?

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.

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.

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.

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 clobbering their values and restore the contents before returning
- Remember: caller/callee need to save only temporary/saved registers they are using, not all registers.

Register Conventsions Summary

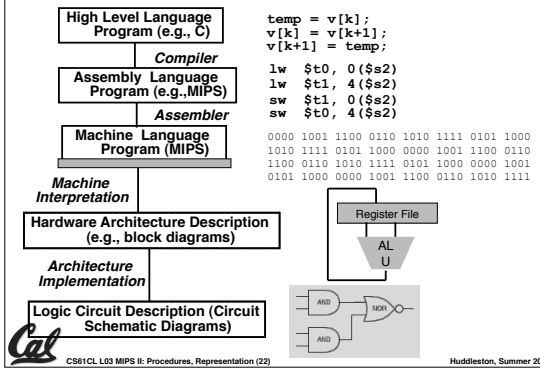
			Preserved?
The constant 0	$0	$zero	n/a
Used by Assembler	$1	$at	n/a
Return Values	$2-$3	$v0-$v1	no
Arguments	$4-$7	$a0-$a3	no
Temporary	$8-$15	$t0-$t7	no
Saved	$16-$23	$s0-$s7	yes
More Temporary	$24-$25	$t8-$t9	no
Used by Kernel	$26-27	$k0-$k1	n/a
Global Pointer	$28	$gp	yes
Stack Pointer	$29	$sp	yes
Frame Pointer	$30	$fp	yes
Return Address	$31	$ra	no

(From COD green insert)
Use names for registers -- code is clearer!

"New" Registers

- `$at`: may be used by the assembler at any time; unsafe to use
- `$k0-$k1`: may be used by the OS at any time; unsafe to use
- `$gp`, `$fp`: don't worry about them
- Note: Feel free to read up on `$gp` and `$fp` in Appendix A, but you can write perfectly good MIPS code without them.

61C Levels of Representation (abstractions)



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

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

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.

R-Format Instructions (1/5)

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

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.

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.

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

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 (p. 264 P&H)

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

R-Format Example (1/2)

- MIPS Instruction:

```
add   $8,$9,$10
```

`opcode` = 0 (look up in table in book)
`funct` = 32 (look up in table in book)
`rd` = 8 (destination)
`rs` = 9 (first *operand*)
`rt` = 10 (second *operand*)
`shamt` = 0 (not a shift)

R-Format Example (2/2)

- MIPS Instruction:

```
add   $8,$9,$10
```

Decimal number per field representation:

0	9	10	8	0	32

Binary number per field representation:

000000	01001	01010	01000	00000	100000

hex representation: $012A\ 4020_{hex}$

decimal representation: $19,546,144_{ten}$

Called a Machine Language Instruction

I-Format Instructions (1/4)

- What about instructions with immediates?
  - 5-bit field only represents numbers up to the value 31: immediates may be much larger than this
  - Ideally, MIPS would have only one instruction format (for simplicity): unfortunately, we need to compromise
- Define new instruction format that is partially consistent with R-format:
  - First notice that, if instruction has immediate, then it uses at most 2 registers.

I-Format Instructions (2/4)

- Define "fields" of the following number of bits each: 6 + 5 + 5 + 16 = 32 bits

6	5	5	16

- Again, each field has a name:

opcode	rs	rt	immediate

- Key Concept: Only one field is inconsistent with R-format. Most importantly, `opcode` is still in same location.

I-Format Instructions (3/4)

- What do these fields mean?
  - `opcode`: same as before except that, since there's no `funct` field, `opcode` uniquely specifies an instruction in I-format
  - This also answers question of why R-format has two 6-bit fields to identify instruction instead of a single 12-bit field: in order to be consistent as possible with other formats while leaving as much space as possible for immediate field.
  - `rs`: specifies a register operand (if there is one)
  - `rt`: specifies register which will receive result of computation (this is why it's called the *target* register "`rt`") or other operand for some instructions.

I-Format Instructions (4/4)

- The Immediate Field:
  - `addi`, `slti`, `sltiu`, the immediate is sign-extended to 32 bits. Thus, it's treated as a signed integer.
  - 16 bits → can be used to represent immediate up to 2^{16} different values
  - This is large enough to handle the offset in a typical `lw` or `sw`, plus a vast majority of values that will be used in the `slti` instruction.
  - If immediate is larger, must be split into multiple instructions (more on this in the bonus slides)

I-Format Example (1/2)

- **MIPS Instruction:**

```
addi   $21,$22, 50
```

`opcode` = 8 (look up in table in book)

`rs` = 22 (register containing operand)

`rt` = 21 (target register)

`immediate` = 50 (the immediate)

I-Format Example (2/2)

- **MIPS Instruction:**

```
addi   $21,$22,-50
```

Decimal/field representation:

8	22	21	-50

Binary/field representation:

001000	10110	10101	1111111111001110

hexadecimal representation: 22D5 $\texttt{FFCE}_{hex}$

decimal representation: $584,449,998_{ten}$

Branches: PC-Relative Addressing (1/5)

- Use I-Format

opcode	rs	rt	immediate

- `opcode` specifies `beq` or `bne`
- `rs` and `rt` specify registers to compare
- What can immediate specify?
  - `immediate` is only 16 bits
  - PC (Program Counter) has byte address of current instruction being executed; 32-bit pointer to memory
  - So `immediate` cannot specify entire address to branch to.

Branches: PC-Relative Addressing (2/5)

- How do we typically use branches?
  - Answer: `if-else`, `while`, `for`
  - Loops are generally small: usually up to 50 instructions
  - Function calls and unconditional jumps are done using jump instructions (`j` and `jal`), not the branches.
- Conclusion: may want to branch to anywhere in memory, but a branch often changes PC by a small amount

Branches: PC-Relative Addressing (3/5)

- Solution to branches in a 32-bit instruction: PC-Relative Addressing
- Let the 16-bit immediate field be an integer to be *added* to the PC if we take the branch.
- Now we can branch $\pm 2^{15}$ bytes from the PC, which should be enough to cover almost any loop.
- Any ideas to further optimize this?

Branches: PC-Relative Addressing (4/5)

- Note: Instructions are words, so they're word aligned (byte address is always a multiple of 4, which means it ends with `00` in binary).
  - So the number of bytes to add to the PC will always be a multiple of 4.
  - So specify the `immediate` in words.
- Now, we can branch $\pm 2^{15}$ words from the PC (or $\pm 2^{17}$ bytes), so we can handle loops 4 times as large.

Branches: PC-Relative Addressing (5/5)

- Branch Calculation:
  - If we don't take the branch:

    PC = PC + 4 *# (address of next instruction)*
  - If we do take the branch:

    PC = (PC + 4) + (`immediate` * 4)
  - Observations
    - `Immediate` field specifies the number of words to jump, which is simply the number of instructions to jump.
    - `Immediate` field can be positive or negative.
    - Due to hardware, add `immediate` to (PC+4), not to PC; will be clearer why later in course

Branch Example (1/3)

- MIPS Code:

```
Loop:  beq   $9,$0,End
       addu  $8,$8,$10
       addiu $9,$9,-1
       j     Loop
 End:
```

- `beq` branch is I-Format:

`opcode` = 4 (look up in table)

`rs` = 9 (first operand)

`rt` = 0 (second operand)

`immediate` = ???

Branch Example (2/3)

- MIPS Code:

```
Loop:  beq   $9,$0,End
       addu  $8,$8,$10
       addiu $9,$9,-1
       j     Loop
 End:
```

- `immediate` Field:
  - Number of instructions to add to (or subtract from) the PC, starting at the instruction *following* the branch.
  - In this case, `immediate` = 3

Branch Example (3/3)

- MIPS Code:

```
Loop:  beq   $9,$0,End
       addu  $8,$8,$10
       addiu $9,$9,-1
       j     Loop
End:
```

decimal representation:

4	9	0	3

binary representation:

000100	01001	00000	0000000000000011

J-Format Instructions (1/5)

- For branches, we assumed that we won't want to branch too far, so we can specify *change* in PC.
- For general jumps (`j` and `jal`), we may jump to *anywhere* in memory.
- Ideally, we could specify a 32-bit memory address to jump to.
- Unfortunately, we can't fit both a 6-bit opcode and a 32-bit address into a single 32-bit word, so we compromise.

J-Format Instructions (2/5)

- Define two "fields" of these bit widths:

6 bits	26 bits

- As usual, each field has a name:

opcode	target address

- Key Concepts
  - Keep `opcode` field identical to R-format and I-format for consistency.
  - Collapse all other fields to make room for large target address.

J-Format Instructions (3/5)

- For now, we can specify 26 bits of the 32-bit bit address.
- Optimization:
  - Note that, just like with branches, jumps will only jump to word aligned addresses, so last two bits are always `00` (in binary).
  - So let's just take this for granted and not even specify them.

J-Format Instructions (4/5)

- Now specify 28 bits of a 32-bit address
- Where do we get the other 4 bits?
  - By definition, take the 4 highest order bits from the PC.
  - Technically, this means that we cannot jump to *anywhere* in memory, but it's adequate 99.9999…% of the time, since programs aren't that long
    - only if straddle a 256 MB boundary
  - If we absolutely need to specify a 32-bit address, we can always put it in a register and use the `jr` instruction.

J-Format Instructions (5/5)

- Summary:
  - New PC = { PC[31..28], target address, 00 }
- Understand where each part came from!
- Note: { , , } means concatenation
  { 4 bits , 26 bits , 2 bits } = 32 bit address
  - { 1010, 11111111111111111111111111, 00 }
    = 10101111111111111111111111111100
  - Note: Book uses ||

"And in Conclusion…"

- 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!

In conclusion

- MIPS Machine Language Instruction:
  32 bits representing a single instruction

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	target address				

- Branches use PC-relative addressing, Jumps use absolute addressing.
- Disassembly is simple and starts by decoding `opcode` field. (more in a week)

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

Instruction Support for Functions (4/6)

- Single instruction to jump and save return address: jump and link (`jal`)
- Before:

```
1008 addi $ra,$zero,1016  #$ra=1016
1012 j sum                #goto sum
1016 ...
```

- After:

```
1008 jal sum  # $ra=1012,goto sum
```

- Why have a `jal`?
  - Make the common case fast: function calls very common.
  - Don't have to know where code is in memory with `jal`!

Instruction Support for Functions (3/6)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

- Question: Why use `jr` here? Why not use `j`?
- Answer: `sum` might be called by many places, so we can't return to a fixed place. The calling proc to `sum` must be able to say "return here" somehow.

MIPS

```
2000 sum: add $v0,$a0,$a1
2004 jr   $ra     # new instruction
```

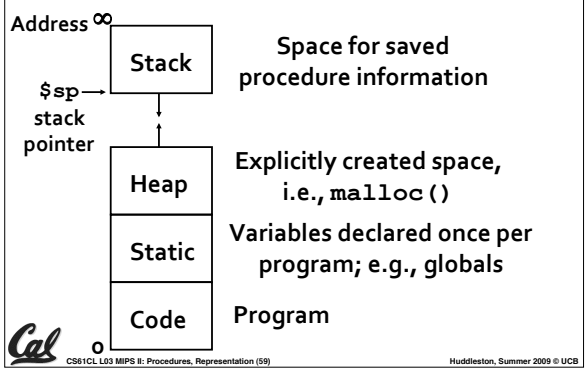

Steps for Making a Procedure Call

1. Save necessary values onto stack.
2. Assign argument(s), if any.
3. `jal` call
4. Restore values from stack.

Nested Procedures (2/2)

- In general, may need to save some other info in addition to `$ra`.
- When a C program is run, there are 3 important memory areas allocated:
  - Static: Variables declared once per program, cease to exist only after execution completes. E.g., C globals
  - Heap: Variables declared dynamically via `malloc`
  - Stack: Space to be used by procedure during execution; this is where we can save register values

C memory Allocation review



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?

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?!"

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

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))))))
```

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));
}
```

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:

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

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  # $a0 = n - 1
  sw $a0, 0($sp)   #  Need $a0 after jal
     jal fib       #  fib(n - 1)
  lw $a0, 0($sp)   #  restore $a0
addi $a0, $a0, -1  # $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, # Place fib(n – 1)
              # somewhere it won't get
              # clobbered
     jal fib # fib(n - 2)
 add $v0, $v0, # $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;
 }
```


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()
                  ...
                __exit:
```

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


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

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;
             }
```


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)

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**

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

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

Consequence #2: Binary Compatibility

- Programs are distributed in binary form
  - Programs bound to specific instruction set
  - Different version for 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

I-Format Problems (0/3)

- Problem 0: Unsigned # sign-extended?
  - `addiu`, `sltiu`, sign-extends immediates to 32 bits. Thus, # is a "signed" integer.
- Rationale
  - `addiu` so that can add w/out overflow
    - See K&R pp. 230, 305
  - `sltiu` suffers so that we can have easy HW
    - Does this mean we'll get wrong answers?
    - Nope, it means assembler has to handle any unsigned immediate $2^{15} \le n < 2^{16}$ (I.e., with a `1` in the 15th bit and `0`s in the upper 2 bytes) as it does for numbers that are too large. ⇒

I-Format Problem (1/3)

- Problem:
  - Chances are that `addi`, `lw`, `sw` and `slti` will use immediates small enough to fit in the immediate field.
  - …but what if it's too big?
  - We need a way to deal with a 32-bit immediate in any I-format instruction.

I-Format Problem (2/3)

- Solution to Problem:
  - Handle it in software + new instruction
  - Don't change the current instructions: instead, add a new instruction to help out
- New instruction:

```
lui   register, immediate
```

  - stands for Load Upper Immediate
  - takes 16-bit immediate and puts these bits in the upper half (high order half) of the register
  - sets lower half to `0`s

I-Format Problems (3/3)

- Solution to Problem (continued):
  - So how does `lui` help us?
  - Example:

```
addiu $t0,$t0, 0xABABCDCD
```

  …becomes

```
lui $at 0xABAB
ori $at, $at, 0xCDCD
addu $t0,$t0,$at
```

  - Now each I-format instruction has only a 16-bit immediate.
  - Wouldn't it be nice if the assembler would this for us automatically? (later)

Decoding Machine Language

- How do we convert 1s and 0s to assembly language and to C code?

  Machine language ⇒ assembly ⇒ C?
- For each 32 bits:
  1. Look at `opcode` to distinguish between R-Format, J-Format, and I-Format.
  2. Use instruction format to determine which fields exist.
  3. Write out MIPS assembly code, converting each field to name, register number/name, or decimal/hex number.
  4. Logically convert this MIPS code into valid C code. Always possible? Unique?

Decoding Example (1/7)

- Here are six machine language instructions in hexadecimal:

00001025_{hex}
$0005402A_{hex}$
11000003_{hex}
00441020_{hex}
$20A5FFFF_{hex}$
08100001_{hex}

- Let the first instruction be at address $4{,}194{,}304_{ten}$ ($0x00400000_{hex}$).
- Next step: convert hex to binary

Decoding Example (2/7)

- The six machine language instructions in binary:

```
00000000000000000001000000100101
00000000000001010100000000101010
00010001000000000000000000000011
00000000010001000001000000100000
00100000101001011111111111111111
00001000000100000000000000000001
```

- Next step: identify opcode and format

R	0	rs	rt	rd	shamt	funct
I	1, 4-62	rs	rt	immediate		
J	2 or 3	target address				

Decoding Example (3/7)

- Select the opcode (first 6 bits) to determine the format:

Format:

R	00000000000000000001000000100101
R	00000000000001010100000000101010
I	00010001000000000000000000000011
R	00000000010001000001000000100000
I	00100000101001011111111111111111
J	00001000000100000000000000000001

- Look at `opcode`: 0 means R-Format, 2 or 3 mean J-Format, otherwise I-Format.
- Next step: separation of fields

Decoding Example (4/7)

- Fields separated based on format/opcode:

Format:

R	0	0	0	2	0	37
R	0	0	5	8	0	42
I	4	8	0	+3		
R	0	2	4	2	0	32
I	8	5	5	-1		
J	2	1,048,577				

- Next step: translate ("disassemble") to MIPS assembly instructions

Decoding Example (5/7)

- MIPS Assembly (Part 1):

```
    Address:          Assembly
instructions:

      0x00400000     or
$2,$0,$0          0x00400004     slt
$8,$0,$5          0x00400008     beq
$8,$0,3           0x0040000c     add
$2,$2,$4          0x00400010     addi
$5,$5,-1          0x00400014     j
0x00400001
```

- Better solution: translate to more meaningful MIPS instructions (fix the branch/jump and add labels, registers)

Decoding Example (6/7)

- MIPS Assembly (Part 2):

```
          or    $v0,$0,$0
 Loop:    slt
$t0,$0,$a1
 beq
$t0,$0,Exit
 add
$v0,$v0,$a0
 addi  $a1,$a1,-1
 j     Loop                    Exit:
```

- Next step: translate to C code (must be creative!)

Decoding Example (7/7)

Before Hex:

00001025_{hex}
$0005402A_{hex}$
11000003_{hex}
00441020_{hex}
$20A5FFFF_{hex}$
08100001_{hex}

- After C code (Mapping below)
  - $v0: product
  - $a0: multiplicand
  - $a1: multiplier

```
    product = 0;
while (multiplier > 0) {
    product += multiplicand;
    multiplier -= 1;
}
```

```
      or   $v0,$0,$0
Loop: slt  $t0,$0,$a1
      beq  $t0,$0,Exit
      add  $v0,$v0,$a0
      addi $a1,$a1,-1
      j    Loop
     Exit:
```

Demonstrated Big 61C Idea: Instructions are just numbers, code is treated like data

Administrivia

- Midterm is next week! Day and location are still TBA
  - Old midterms online (link at top of page)
  - Lectures and reading materials fair game
  - Fix green sheet errors (if old book)
- Review session also TBA
- Project 2 is due March 5 at 11:59PM
  - That's Wednesday!
  - There was a file update. See spec page.

Review from before: `lui`

- So how does `lui` help us?
  - Example:

```
addi   $t0,$t0, 0xABABCDCD
```

becomes:

```
 lui    $at, 0xABAB
ori    $at, $at, 0xCDCD
add    $t0,$t0,$at
```

  - Now each I-format instruction has only a 16-bit immediate.
- Wouldn't it be nice if the assembler would this for us automatically?
  - If number too big, then just automatically replace `addi` with `lui`, `ori`, `add`

True Assembly Language (1/3)

- Pseudoinstruction: A MIPS instruction that doesn't turn directly into a machine language instruction, but into other MIPS instructions
- What happens with pseudo-instructions?
  - They're broken up by the assembler into several "real" MIPS instructions.
- Some examples follow

Example Pseudoinstructions

- Register Move

```
move   reg2,reg1
```

Expands to:

```
add    reg2,$zero,reg1
```

- Load Immediate

```
li     reg,value
```

If value fits in 16 bits:

```
addi   reg,$zero,value
```

else:

```
lui    reg,upper 16 bits of value
ori    reg,$zero,lower 16 bits
```

Example Pseudoinstructions

- Load Address: How do we get the address of an instruction or global variable into a register?

```
la     reg,label
```

Again if value fits in 16 bits:

```
addi   reg,$zero,label_value
```

else:

```
lui    reg,upper 16 bits of value
ori    reg,$zero,lower 16 bits
```

True Assembly Language (2/3)

- Problem:
  - When breaking up a pseudo-instruction, the assembler may need to use an extra register
  - If it uses any regular register, it'll overwrite whatever the program has put into it.
- Solution:
  - Reserve a register ($1, called `$at` for "assembler temporary") that assembler will use to break up pseudo-instructions.
  - Since the assembler may use this at any time, it's not safe to code with it.

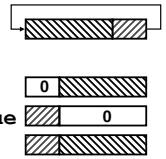
Example Pseudoinstructions

- Rotate Right Instruction

```
ror   reg, value
```

Expands to:

```
srl   $at, reg, value
sll   reg, reg, 32-value
or    reg, reg, $at
```



- "No OPeration" instruction

```
nop
```

Expands to instruction = 0_{ten},

```
sll   $0, $0, 0
```


Example Pseudoinstructions

- Wrong operation for operand

```
addu  reg,reg,value # should be addiu
```

If value fits in 16 bits, `addu` is changed to:

```
addiu reg,reg,value
```

else:

```
lui   $at,upper 16 bits of value
ori   $at,$at,lower 16 bits
addu  reg,reg,$at
```

- How do we avoid confusion about whether we are talking about MIPS assembler with or without pseudoinstructions?

True Assembly Language (3/3)

- MAL (MIPS Assembly Language): the set of instructions that a programmer may use to code in MIPS; this includes pseudoinstructions
- TAL (True Assembly Language): set of instructions that can actually get translated into a single machine language instruction (32-bit binary string)
- A program must be converted from MAL into TAL before translation into 1s & 0s.

Questions on Pseudoinstructions

- Question:
  - How does MIPS assembler / SPIM recognize pseudo-instructions?
- Answer:
  - It looks for officially defined pseudo-instructions, such as `ror` and `move`
  - It looks for special cases where the operand is incorrect for the operation and tries to handle it gracefully

Rewrite TAL as MAL

- TAL:

```
        or    $v0,$0,$0
Loop:   slt
$t0,$0,$a1
 beq
$t0,$0,Exit
 add
$v0,$v0,$a0
 addi  $a1,$a1,-1
 j     Loop
                    Exit:
```

- This time convert to MAL
- It's OK for this exercise to make up MAL instructions

Rewrite TAL as MAL (Answer)

- TAL:

```
        or    $v0,$0,$0
Loop:  slt
$t0,$0,$a1                    beq
$t0,$0,Exit
   add   $v0,$v0,$a0
   addi
$a1,$a1,-1                    j
Loop                  Exit:
```

- MAL:

```
        li    $v0,0
Loop:   ble   $a1,$zero,Exit
        add   $v0,$v0,$a0
        sub   $a1,$a1,1
        j     Loop
Exit:
```


Questions on PC-addressing

- Does the value in branch field change if we move the code?
- What do we do if destination is > 2^{15} instructions away from branch?
- Why do we need different addressing modes (different ways of forming a memory address)? Why not just one?