CS61C
Constants and Making Decisions in C/Assembly Language

Lecture 3

January 27, 1999

Dave Patterson
(http.cs.berkeley.edu/~patterson)

www-inst.eecs.berkeley.edu/~cs61c/schedule.html

Review 1/2

- Big idea in CS&E; compilation to translate from one level of abstraction to lower level
  - Generally single HLL statement produces many assembly instructions
  - Also hides address calculations (byte vs. word addressing)
- Design of an Assembly Language like MIPS shaped by
  1) Desire to keep hardware simple: e.g., each operation has 3 operands
  2) Smaller is faster: e.g., MIPS has 32 registers

Review 2/2

- MIPS assembly language thus far:
  - Instructions: `add, sub, lw, sw`
  - At most one assembly instruction per line
  - Comments start with # to end of line
  - Operands: registers `$s0, $s1, ... ; $t0, $t1, ...`
  - Operands: memory `Memory[0], Memory[4], Memory[8], ... , Memory[4294967292]`

Overview

- Constants (3 minutes)
- C/Assembly `if`, `goto`, `if-else` (7 min)
- C /Assembly Loops: `goto`, `while` (10 min)
- Administrivia,"What's this good for" (5)
- Test for less (6 min)
- C/Assembly case/switch statement (12)
- C/Assembly `for` Example (6 minutes)
- Conclusion (1 minute)

Assembly Constants

- C expressions can have constants:

```
i = i + 10;
```

- How get them using instructions so far?
  - Constants kept in memory with program

```
lw $t0, 0($s0)    # load 10 from memory
add $s3,$s3,$t0   # i = i + 10
```

- So common, have instruction to add constants (called "immediate instructions")

```
addi $s3,$s3,10   # i = i + 10
```


Assembly Constants

- Why include immediate instructions?
  - Design principle: Make the common case fast
  - Why faster?
    - Don't need to access memory
    - 2 instructions v. 1 instruction
- Also,perhaps most popular constant is zero
  - MIPS designers reserved 1 of 32 registers to always have the value 0; called "$zero"
  - Useful in making additional operations from existing instructions; e.g., $s0 ⇐ s1

```
add $s0, $s1, $zero # $s0 = $s1 + 0
```


C Decisions (Control Flow): `if` statements

- 2 kinds of if statements in C
  - `if (`*condition*`)` *statement*
  - `if (`*condition*`)` *statement1* `else` *statement2*
- Following code is same as 2nd `if`

```
if  (condition) go to L1;
    statement2;
    go to L2;
L1: statement1;
L2:
```

- Not as elegant as if-else, but same meaning

MIPS decision instructions (control flow)

- Decision instruction in MIPS:
  - `beq register1, register2, L1`
  - `beq` is "Branch if (registers are) equal" Same meaning as (using C):

```
if  (register1==register2) go to L1
```

- Complementary MIPS decision instruction
  - `bne register1, register2, L1`
  - `bne` is "Branch if (registers are) not equal" Same meaning as (using C):

```
if  (register1!=register2) go to L1
```

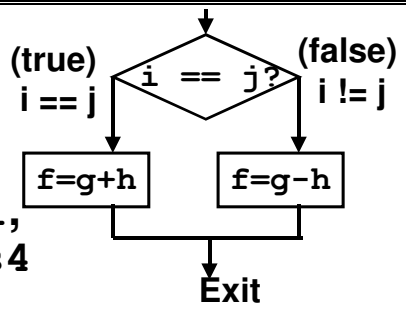
- Called conditional branches

Compiling C if into MIPS Assembly

- Compile by hand

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

Mapping f: $s0, g: $s1, h: $s2, i: $s3, j: $s4



- Start with branch:

```
beq $s3,s4, True # branch to True
                 # if i==j
```

- Follow with false part

```
sub $s0,$s1,$s2 # f=g-h(skip if i==j)
```


Compiling C if into MIPS Assembly

- Next must skip over true part
  - Need instruction that always branches (unconditional branch)
  - MIPS has jump:

```
j Exit # go to Exit
```

- Next is true part

```
True: add $s0,$s1,$s2   # f=g+h
```

- Followed by exit branch label

```
Exit:
```

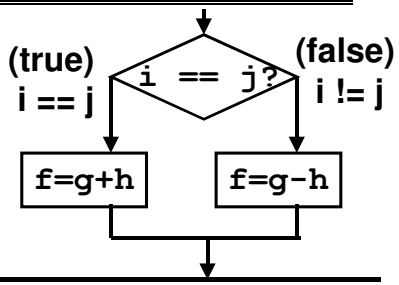

Compiling C `if` into MIPS: Summary

C

- Compile by hand

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

Mapping f: $s0, g: $s1, h: $s2, i: $s3, j: $s4



MIPS

```
      beq $s3,s4, True # branch i==j
      sub $s0,$s1,$s2  # f=g-h(false)
      j Exit           # go to Exit
True: add $s0,$s1,$s2  # f=g+h (true)
Exit:
```

- Note:Compiler supplies labels, branches not found in HLL code; often it flips the condition to branch to false part

Loops in C/Assembly

- Simple loop in C

```
Loop:   g = g + A[i];
        i = i + j;
        if (i != h) goto Loop;
```

- 1st fetch A[i] (g,h,i,j:$s1,$s2,$s3,$s4;base of `A[]`:$s5):

```
Loop: add $t1,$s3,$s3 #$t1= 2*i
      add $t1,$t1,$t1 #$t1= 4*i
      add $t1,$t1,$s5 #$t1=addr A
      lw  $t1,0($t1)  #$t1=A[i]
```

Loops in C /Assembly

- Add A[i] to g and then j to i (g,h,i,j:$s1,$s2,$s3,$s4):

```
add  $s1,$s1,$t1   # g = g+A[i]
add  $s3,$s3,$s4   # i = i + j
```

The final instruction branches back to Loop if `i != h`:

```
bne $s3,$s2,Loop # goto Loop
                 # if i!=h
```


Loops in C/Assembly: Summary

C

```
Loop:  g = g + A[i];
       i = i + j;
       if (i != h) goto Loop;
```

(g,h,i,j:$s1,$s2,$s3,$s4 : base of `A[]`:$s5)

MIPS

```
Loop: add $t1,$s3,$s3 #$t1= 2*i
      add $t1,$t1,$t1 #$t1= 4*i
      add $t1,$t1,$s5 #$t1=addr A
      lw  $t1,0($t1)  #$t1=A[i]
      add $s1,$s1,$t1 #g=g+A[i]
      add $s3,$s3,$s4 #i=i + j
      bne $s3,$s2,Loop# goto Loop
                      # if i!=h
```


While in C/Assembly:

- Although legal C, almost never write loops with `if`, `goto`: use `while`, `for` loops
- Syntax: `while`(condition) statement

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

- 1st load save[i] into a temporary register (i,j,k: $s3,$s4,$s5: base of `save[]`:$s6):

```
Loop: add $t1,$s3,$s3 #$t1 = 2*i
      add $t1,$t1,$t1 #$t1 = 4*i
      add $t1,$t1,$s6 #$t1=Addr
      lw  $t1,0($t1) #$t1=save[i]
```


While in C/Assembly:

- Loop test, exiting if save[i] != k (i,j,k: $s3,$s4,$s5: base of `save[]`:$s6):

```
bne $t1,$s5,Exit#goto Exit
                #if save[i]!=k
```

- The next instruction adds j to i:

```
add  $s3,$s3,$s4   # i = i + j
```

- End of loop branches back to the while test at top of loop. Add the Exit label after:

```
   j    Loop          # goto Loop
Exit:
```

While in C/Assembly: Summary

C

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

(i,j,k: $s3,$s4,$s5: base of save[]:$s6)

MIPS

```
Loop: add $t1,$s3,$s3 #$t1 = 2*i
      add $t1,$t1,$t1 #$t1 = 4*i
      add $t1,$t1,$s6 #$t1=Addr
      lw  $t1,0($t1)  #$t1=save[i]
      bne $t1,$s5,Exit#goto Exit
                      #if save[i]!=k
      add $s3,$s3,$s4  # i = i + j
      j   Loop         # goto Loop

Exit:
```


Patterson Spring 99 ©UCB

Administrivia

- Change in what said last time: teams of 1-2, not 3
  - Much less chance of being "third wheel" if team size is 2; OK to do by yourself
  - 1-2 people for Labs, Exercises, Projects
  - May change and work with different people
- 1st homework: Due today (1/27) 7PM (grace period until 8AM tomorrow)
- 1st project: C spelling checker `philspel` Due Wed. 2/3 7PM (do by yourself)
  `www-inst.EECS/~cs61c/handouts/proj1.pdf`

Patterson Spring 99 ©UCB

Administrivia

- 2nd homework: Due Wed 2/3 7PM
  - Exercises 3.1, 3.2, 3.4, 3.6, 3.9
  - Which instruction set inside? Search WWW
    - Apple iMAC
    - Casio PalmPC
    - Cisco Network Routers
    - HP LaserJet 4000
    - IBM PC
    - Kodak DC260 Digital Camera
    - NASA Mars Rover
    - Nintendo 64
    - Sony Playstation
    - Web TV set top box
- Discussion section moved: Th 4-5 to 247 Cory

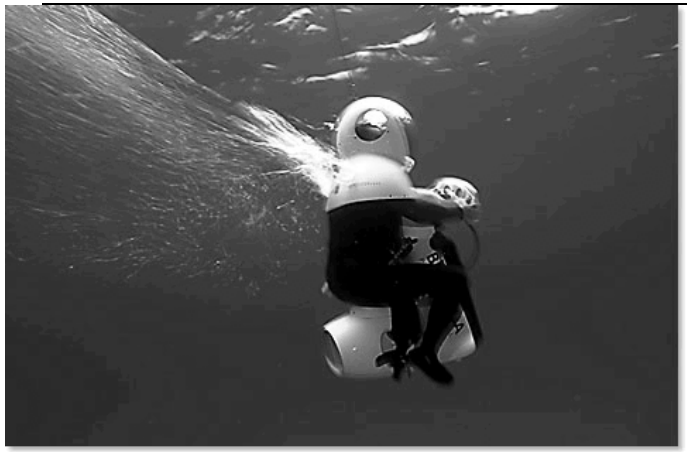
Patterson Spring 99 ©UCB

Administrivia

- Classroom Decorum
  - If need to leave before end of class, sit at end of row near door?
  - Disrupting attention of your classmates
- "And in Conclusion …"
  - Thus far it has been as sign to start shuffling papers : too much noise to hear?
  - As it is only 1 minute before end of class, please wait for 60 seconds before making noise?
- New option on lecture notes: 4/page in pdf format; fast to download and print at home

Patterson Spring 99 ©UCB

"What's This Stuff Good For?"



Breathing Observation Bubble: BOB pipes air from a tank under the handlebars into an acrylic dome, replacing a diver's face mask and breathing apparatus. Wireless technology lets riders talk to other BOBsters darting through the water nearby, as well as to armchair divers above in a boat or back on shore. Saving energy from not having to kick, divers can stay submerged almost an hour with the BOB. Like most modern scuba gear, the BOB features a computer that tells riders when to come up and calculates decompression times for a safe return to the surface. *One Digital Day,* 1998 www.intel.com/onedigitalday

What do applications ("apps") like these mean for reliability requirements of our technology?

Beyond equality tests in MIPS Assembly

- So far =, ≠; what about < or >?
- MIPS instruction "Set Less Than"

```
slt register1,register2,register3
```

- Meaning of `slt` in C

```
if (register2 < register 3)
        register1 = 1;
        else register1 = 0;
```

- Branches then test result in register 1
- Try

```
if (g < h) go to Less
```


If less in C/Assembly

C
```
if (g < h) go to Less
```

MIPS
```
slt $t0,$s0,$s1  # $t0 = 1 if
                 # $s0<$s1 (g<h)
bne $t0,$zero, Less # goto Less
                    # if $t0!=0
        . . .       # (if (g <h))
Less:
```

A branch if `$t0` != 0 branches if g < h.

- **Register `$zero` always 0, so use `bne` comparing register `$t0` to register `$zero`**

- How test if (g >= h)?

Set Less Than Immediate in C/Assembly

- Also immediate version of `slt` to test against constants: `slti`
  - Helpful in for loops

C
```
if (g >= 1) go to Loop
```

MIPS
```
Loop:  . . .
slti $t0,$s0,1     # $t0 = 1 if
                   # $s0<1 (g<1)
beq  $t0,$zero,Loop # goto Loop
                    # if $t0==0
                    # (if (g>=1))
```

C case/switch statement

- Choose among four alternatives depending on whether `k` has the value 0, 1, 2, or 3

```
switch (k) {
 case 0: f=i+j; break; /* k=0*/
 case 1: f=g+h; break; /* k=1*/
 case 2: f=g-h; break; /* k=2*/
 case 3: f=i-j; break; /* k=3*/
}
```


Case/switch via chained if-else, C/Asm.

- Could be done like chain of if-else

C

```
if(k==0) f=i+j;
 else if(k==1) f=g+h;
   else if(k==2) f=g-h;
    else if(k==3) f=i-j;
```

MIPS

```
     bne  $s5,$zero, L1 # branch k!=0
     add  $s0,$s3,$s4 #k=0 so f=i+j
     j    Exit # end of case so Exit
L1:  subi $t0,$s5,1    # $t0=k-1
     bne  $t1,$zero,L2 # branch k!=1
     add  $s0,$s1,$s2 #k=1 so f=g+h
     j    Exit # end of case so Exit
L2:  subi $t0,$s5,2    # $t0=k-2
     bne  $t1,$zero,L3 # branch k!=2
     sub  $s0,$s1,$s2 #k=2 so f=g-h
     j    Exit # end of case so Exit
L3:  sub  $s0,$s3,$s4 #k=3 so f=i-j
Exit:
```


Case/Switch via Jump Address Table

- Notice that last case must wait for n-1 tests before executing, making it slow
- Alternative tries to go to all cases equally fast: jump address table
  - Idea: encode alternatives as a table of addresses of the cases
    - Table an array of words with addresses corresponding to case labels
  - Program indexes into table and jumps
- MIPS instruction "jump register" (`jr`) unconditionally branches to address in register; use load to get address

Case/Switch via Jump Address Table 1/3

- Use `k` to index a jump address table, and then jump via the value loaded
- 1st test that k matches 1 of cases (0<=k<=3); if not, the code exits

```
slti $t3,$s5,0      #Test if k < 0
bne $t3,$zero,Exit  #if k<0,goto Exit
slti $t3,$s5,4      # Test if k < 4
beq $t3,$zero,Exit  #if k>=4,goto Exit
```

- Multiply k by 4 to index table of words:

```
add $t1,$s5,$s5 # Temp reg $t1 = 2*k
add $t1,$t1,$t1 # Temp reg $t1 = 4*k
```

Case/Switch via Jump Address Table 2/3

- Assume 4 sequential words in memory, base address in $t2, have addresses corresponding to labels L0, L1, L2, L3.

```
add  $t1,$t1,$t2# $t1=addr JumpTable[k]
lw   $t1,0($t1) # $t1 = JumpTable[k]
```

- Now jump using address in register $t1:

```
jr   $t1    # jump based on reg. $t1
```


Case/Switch via Jump Address Table 3/3

- Cases jumped to by `jr`:

```
L0: add  $s0,$s3,$s4 # k=0 so f = i + j
    j    Exit    # end case, goto Exit
L1: add  $s0,$s1,$s2 # k=1 so f = g + h
    j    Exit    # end case, goto Exit
L2: sub  $s0,$s1,$s2 # k=2 so f = g - h
    j    Exit    # end case, goto Exit
L3: sub  $s0,$s3,$s4 # k=3 so f = i - j
Exit:       # end of switch statement
```


Jump Address Table: Summary

```
slti $t3,$s5,0     #Test if k < 0
bne $t3,$zero,Exit #if k<0,goto Exit
slti $t3,$s5,4     # Test if k < 4
beq $t3,$zero,Exit #if k>=4,goto Exit
add $t1,$s5,$s5 # Temp reg $t1 = 2*k
add $t1,$t1,$t1 # Temp reg $t1 = 4*k
add $t1,$t1,$t2 # $t1=addr JumpTable[k]
lw  $t1,0($t1)  # $t1 = JumpTable[k]
jr  $t1         # jump based on $t1
L0: add  $s0,$s3,$s4 # k=0 so f = i + j
    j    Exit    # end case, goto Exit
L1: add  $s0,$s1,$s2 # k=1 so f = g + h
    j    Exit    # end case, goto Exit
L2: sub  $s0,$s1,$s2 # k=2 so f = g - h
    j    Exit    # end case, goto Exit
L3: sub  $s0,$s3,$s4 # k=3 so f = i - j
Exit:       # end of switch statement
```


If there is time, do it yourself:

- Compile this MIPS code:

```
sum = 0;
for (i=0;i<10;i=i+1)
    sum = sum + A[i];
```

- `sum`:`$s3`, `i`:`$s4`, base address of `A`:`$s5`

"And in Conclusion …" 1/1

- Constants so common have special version of arithmetic, registers
  - `addi`, `subi;` register `$zero` (always 0)
  - Principle: Making common case fast
- HLL decisions (if, case) and loops (while, for) use same assembly instructions
  - Conditional branches: `beq`, `bne` in MIPS
  - Unconditional branches: `j`, `jr` in MIPS
  - Relative test: `slt`, `slti` in MIPS
  - Case/Switch: either jump table + `jr`
    or simply chained if-else
  - Next: procedures, functions

Patterson Spring 99 ©UCB