CS311 Lecture: Control Structures Revised 9/17/07

Objectives:

1. To introduce the MIPS unconditional branching instructions - j, jr
2. To introduce the MIPS conditional branching instructions - beq, bne
3. To show how HLL control structures can be realized by branches
4. To show how to implement inequality comparisons using slt, slti

Materials:

1. MIPS ISA Handout (they already have)
2. Handout on Translation of Control Structures

I. Introduction
- ------------

A. In our first intoduction to the execution cycle of a Von Neumann architecture computer, we met the Program Counter (pc) register - which always holds the address of the NEXT instruction to be fetched from memory and executed.

1. In the standard fetch/execute cycle, the pc is updated after fetching an instruction to point to the next successive instruction.

2. In the case of MIPS, this means adding 4 to the pc after each instruction is fetched, since all MIPS instructions are one word (4 bytes) long.

3. Obviously, if this were the only way to update the pc, this would result in executing each instruction in the program once, from top to bottom without any variation - which would not usually be usefui.

B. In HLL's such as C/C++ or Java , we typically have a number of constructs for altering the order of program execution within a procedure - e.g.

```
if (...) ... else ...
switch (...) { case ... case ... case ... default ... }
while (...) ...
do ... while (...)
for (...) ...
goto ...        // C/C++ only - not Java
```

C. In machine language, regardless of the machine, we basically have only two:

1. The equivalent of goto ... - puts a new value into the pc, causing the next instruction to be fetched from that location. This is called in various ISA's an UNCONDITIONAL BRANCH or a JUMP.

2. The equivalent of if (...) goto - puts a new value into the pc if and only if some condition is true. This is called CONDITIONAL BRANCH or simply BRANCH.

3. Actually, we could make do with only the conditional form - we could get the effect of an unconditional branch by using a conditional branch with a condition that we guarantee to be true. However, most architectures, including MIPS, provide both forms, because the unconditional form is simpler and can be made more flexible.

D. In this lecture, we will focus on MIPS instructions for altering the order of program execution. However, similar facilities are found in all ISAs.

II. MIPS Program-Control Instructions

A. Unconditional jumps - Two variants:

1. j address - Destination address is contained in instruction

   a. It looks like this (all values given in decimal)

# of bits	6	26
field name	op	target address
contents	op = 2 for j	

   This general format of instruction is called J format (where J stands for "jump"

   b. The jump instruction specifies the address of an instruction; this address is put in the pc in place of the value computed by adding 4 to the address of the current instruction.

   c. The target address field in the instruction is only 26 bits long, which isn't long enough to specify an address anywhere in memory (for which a 32 bit address is required). Two measures are used to deal with this.

      i. The target address specified in the instruction is multiplied by 4. Multiplication by 4 is done because instructions are words, and therefore must have addresses that are a multiple of 4. Doing this allows a 26 bit address to encode information that corresponds to the lower 28 bits of the pc.

      ii. The target address is loaded into the lower 28 bits of the pc, and the upper 4 bits are left alone. This means that the jump must always be to an address in the same 256 MB chunk of memory as the instruction is in. Since most programs are much smaller than 256 MB, this hasn't proved to be a serious problem - (yet!)

2. jr register - Destination address is contained in register

   a. It looks like this (all values given in decimal)

# of bits	6	5	5	5	5	6
field name	op	rs	rt	rd	shamt	funct
contents	op = 0 for most R type instructons	reg that holds destination	(not used - 0)	(not used - 0)	(not used - 0)	8 for jr

   (Note that this is an R format instruction, though most of the fields go unused.)

b. The contents of the specified register is placed in the pc, becoming the address of the next instruction. Multiplication by 4 is NOT done (not needed).

c. With this format, any location in memory could be the destination, so it could be used in the case where j could not reach a desired destination - but this is rarely, if ever, an issue.

d. More commonly, the jr is used for two cases, both of which require that the destination address of the branch be computed during program execution, rather than being hardwired into the code.

i. In implementing switch instructions - discussed below

ii. In implementing return from a procedure - discussed in the next series of lectures.

3. As was true with the branch instructions, jumps are typically followed by a nop, because the next instruction is fetched at the same time it is discovered that the instruction begin executed is a jump.

B. We talked earlier about the two conditional branch instructions

1. The two instructions are

a. beq - branch if the two registers are equal

b. bne - branch if the two registers are not equal

2. Recall that conditional branches are I format instructions that specify two registers to be compared and a 16 bit signed offset which is added to the PC if the branch is taken.

a. Recall that the offset is first multiplied by 4 (because all instruction addresses are a multiple of 4) and then added to the value currently in the pc, which is by this time the address of the NEXT instruction to be executed.

b. The offset can range from -32678 to +32767. After multiplication by 4, and adding to the address of the next instruction, this means that conditional branches can "reach" to an instruction in the range

address of branch instruction - 131068 ..
address of branch instruction + 131072

4. Recall that a conditional branch instruction is typically followed by a nop.

III. Translating HLL Control Structures

A. We are now ready to see how some familiar HLL control structures can be translated into assembly/machine language.

1. To keep our focus on the control structures, we'll write the HLL statements in terms of CPU registers - actually they would be written in terms of HLL variables which have to be mapped/load into registers, of course.

2. Likewise, we'll specify the target addresses of the branch instructions symbolically - e.g.

```
        bne     $4, $5, L1
        ...
        ...
L1:     some instruction
```

will mean "put a target address into the branch instruction such that when it is multiplied by 4 and added to the address of the next instruction it will cause execution to continue at the instruction labelled L1:

Example: Suppose the bne instruction is at address 0x1000, and the instruction labelled L1 is at 0x100c - then the branch instruction would be encoded as:

bits	31..26	25..21	20..16	15..0
	(6)	(5)	(5)	(16)
field values (decimal)	5	4	5	2
binary	000101	00100	00101	0000000000000010
	= 0001 0100 1000 0101 0000 0000 0000 0010			
hexadecimal	= 0x14850002			

The instruction contains 2 in the offset field because the instruction following the branch is at 0x1004, and 0x1004 + (4 x 2) = 0x100c = desired target.

(This is a computation that the assembler routinely does.)

DISTRIBUTE CONTROL STRUCTURES HANDOUT

B. Simple if (no else)

```
Example: if ($4 == $5)
             do something

         bne $4, $5, L1
         code for do something
L1:      (next instruction after the if)
```

(Note the inversion of the sense of the branch)

C. If .. else

```
   Example:  if ($4 == $5)
                 do something
             else
                 do something else

             bne $4, $5, L1
             code for do something
             j L2
       L1:   code for do something else
       L2:   (next instruction after the if)
       (Note, again, the inversion of the sense of the branch)
```

D. While loop

```
   Example:  while ($4 == $5)
                 do something

       L1:   bne $4, $5, L2
             code for do something
             j L1
       L2:   (next instruction after the while)
```

or: (slightly more efficient - one less instruction in loop body)

```
             j L2
       L1:   code for do something
       L2:   beq $4, $5, L1
```

E. Do .. while loop

```
   Example:  do
                 do something
             while ($4 == $5)

       L1:   code for do something
             beq $4, $5, L1
```

F. For loop

```
   Example:  for ($4 = 0, $4 != $5, $4 ++)
                 do something

             sub $4, $4, $4
       L1:   beq $4, $5, L2
             code for do something
             addi $4, $4, 1
             j L1
       L2:   (next instruction after the for)
```

or: (slightly more efficient - one less instruction in loop body)

```
             sub $4, $4, $4
             j L2
       L1:   code for do something
             addi $4, $4, 1
       L2:   bne $4, $5, L1
```

G. Switch statement - two possible approaches

1. Translate the switch as if it were a series of ifs:

Example:

```
switch(x)
{
    case v1:
        code1;
        break;
    case v2:
        code2;
        break;
    ...
    default:
        coded
}
```

```
        lw $4, x
        li $2, v1
        bne $4, $2, L1
        code1
        j Lfini
L1:     li $2, v2
        bne $4, $2, L2
        code2
        j LFini
L2:     ...
Ln:     coded
LFini:
```

2. The problem with the above is that we could be forced to do as many comparisons as there are cases - and on the average would do half. If the set of case labels forms a dense set, with few or any missing values, then a much more efficient translation is possible using a JUMP TABLE.

Example: We could translate the following switch statement as shown

```
switch(x)
{
    case 0:
        code0;
        break;
    case 1:
        code1;
        break;
    case 2:
        code2;
        break;
    default:
        coded;
}
```

```
lw $4, x
-- if $4 < 0 or $4 > 2 do coded (as above) then j LFini
-- multiply $4 by 4 (shift left two places)
lw $4, JTable($4)
jr $4
```

```
JTable: Lcode0
        Lcode1
        Lcode2

Lcode0: code0
        j LFini
Lcode1: code1
        j LFini
Lcode2: code2
        j LFini

LFini:
```

H. Note that, on occassion, it may be necessary to translate an HLL construct circuitously, because of the limited range of the conditional branch instructions (though this will not happen often on a machine like MIPS because the range of the conditional branch is -32767/+32768 INSTRUCTIONS. It is conceivable that a problem could arise with the initial conditional branch at the start of a very large switch statement.

1. Typical example:

```
if ($4 == 0)
    $4 = $5

        bne     $4, $0, L1
        move    $4, $5          # translated as add $4, $5, $0
L1:     (instruction after if)
```

2. Unusual example:

```
if ($4 == 0)
    ... very long series of statements
```

Might have to be translated as follows:

```
        beq     $4, $0, L1
        j       L2
L1:     ... translation of long series of statements
        ...
L2
```

IV. Handling Comparisons for Other than Exact Equality

A. The conditional branches on MIPS allow us to compare two registers for exact equality - i.e. they correspond to the C/C++/Java operators == and !=. What if we want to compare for inequality - i.e. we want the assembly language equivalents of >, >=, <, or <= ?

B. The MIPS ISA uses auxillary instructions that set a register to 1 or 0 based on comparison of two other values.

slt - compare two registers
slti - compare a register to an immediate value

1. The slt instruction has the following format in machine language:

# of bits	6	5	5	5	5	6
field name	op	rs	rt	rd	shamt	funct
contents	op = 0 for most R type instructons	1st source reg	2nd source reg	dest reg	(not used - 0)	arith/logical function = 42 for slt

(Note: this is an R Format instruction)

Example: slt $2, $4, $0

bits	31..26 (6)	25..21 (5)	20..16 (5)	15..11 (5)	10..6 (5)	5..0 (6)
field values (decimal)	0	4	0	2	0	42
binary	000000	00100	00000	00010	00000	101010

= 0000 0000 1000 0000 0001 0000 0010 1010

hexadecimal = 0x0080102a

2. The slti instruction has the following format in machine language:

# of bits	6	5	5	16
field name	op	rs	rt	immediate value
contents	op = 10 for slti	source reg	destination reg	value to compare to (two's complement signed number)

(Note: this is an I Format instruction)

Example: slti $2, $4, 42

bits	31..26 (6)	25..21 (5)	20..16 (5)	15..0 (16)
field values (decimal)	10	4	2	42
binary	001010	00100	00010	0000000000101010

= 0010 1000 1000 0010 0000 0000 0010 1010

hexadecimal = 0x2882002a

3. Example: suppose we want to calculate the absolute value of the number in register 4 - i.e. if the value in register 4 is less than 0, we want to negate it (by subtracting it from zero.)

Assume we can use $2 as a temporary.

a. One approach, using slt to compare $4 to $0 (always zero).

```
        slt     $2, $4, $0      # $2 = 1 iff $4 < 0
        beq     $2, $0, L1      # if $2 = 0. skip next instruction
        sub     $4, $0, $4      # negate $4
L1:
```

b. A similar approach, but using slti to compare $4 to literal 0.

```
        slti    $2, $4, 0       # $2 = 1 iff $4 < 0
        beq     $2, $0, L1      # if $2 = 0. skip next instruction
        sub     $4, $0, $4      # negate $4
L1:
```

C. With only a set if less than, how do we handle other comparison orders?

ASK

```
1. x < y        slt     temp, x, y
                bne     temp, target

2. x > y        slt     temp, y, x
                bne     temp, target

3. x <= y       slt     temp, y, x
                beq     temp, target    (x <= y <=> not y < x)

4. x >= y       slt     temp, x, y
                beq     temp, target    (x >= y <=> not x < y)
```