CPS311 Lecture: Control Structures

Last revised July 28, 2021

Objectives:

1. To show how HLL control structures can be realized by conditional branches

Materials:

1. MIPS ISA Handout (they already have)
2. Control Structures Handout and projectable form
3. Projectables
4. Activity

I. Introduction

A. In our first introduction 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 useful.

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, in MIPs and most ISA's, 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. (MIPS j, jr)

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. (MIPs beq, bne)

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., such as $0 == $0.

   However, most ISA's, 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 using MIPS instructions for altering the order of program execution. However, similar facilities are found in all ISAs.

II. MIPS Conditional Branches

A. Recall from our previous introduction to MIPS that the conditional branch instructions (beq, bne) compare two registers and branch if the two registers are equal or not equal as the case may be.

B. Both conditional branches are I format instructions, and look like this

# of bits	6	5	5	16
field name	op	rs	rt	immediate value
contents	op = 4 for beq 5 for bne	first reg to compare	second reg to compare	offset (two's complement signed number)

(where rs and rt now specify the registers to compare)

PROJECT

C. Both conditional branches specify the destination of the branch as an offset relative to the value currently in the PC.

1. The offset is sign-extended to yield a signed number, and then is multiplied by 4 (because all instruction addresses are a multiple of 4) and then added to the value currently in he pc, which is by this time the address of the NEXT instruction to be executed. (So the offset if effect specifies the distance in instructions - not bytes.)

2. 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

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

D. What about other comparisons?

1. The conditional branch instructions allow you to test if two values are equal or not equal, but what about comparisons such as <, >, <= or >=?

2. The `slt, sltu, slti, and sltiu` instructions are used for this.

   a) These are R-Format instructions, so they specify three register operands - two sources and a destination.

   b) `slt` sets rd to 1 if rs < rt, and sets it to 0 if this is not the case. Thus, to branch to a line labeled "less" if $4 < $5, using $2 as a temporary,, one can write:

```
slt $2, $4, $5
bne less, $2, $0
```

   PROJECT

   (1) If $4 is less than $5, $2 is set to 1 by the `slt` - else it is set to 0.

(2)The `bne` branches if $2 != 0 - which is the case if it was set to 1 - or does not branch if $2 == 0.

c) `sltu` is like `slt`, except that it compares compares two values on the basis of unsigned comparison, To see why this is important, consider values such as the bytes `0x0` and `0xff`.

(1)Regarded as signed numbers, `0xff` is -1, which is < 0. But regarded as unsigned numbers, `0xff` is 255, which is > 0.

(2)Indeed, the distinction between the signed and unsigned forms is always relevant whenever it is possible that one or both of the values represent negative numbers, which look like large unsigned values.

d) `slti` and `sltiu` are like `slt` and `sltu`, except that they compare a register to a constant rather than another register.

So, for example, to branch to less if the value of $4 is less than 2, one could write

```
slti $2, $4, 2
bne less, $2
```

PROJECT

3. What about comparisons other than less?

a) At first glance, it appears that mips should also include instructions like `sgt` (set if greater than)

b) But, in fact, in keeping with the "reduced instruction set" (RISC) philosophy the designers of mips only included less than comparisons, because all other comparisons can be done using this and the right value of `beq` or `bne`.

(1) a > b is equivalent to as b < a.

(2) a >= b is equivalent to !(a < b), which requires using using `beq` instead of `bne` after the `slt`.

(3) a <= b is equivalent to ! (b > a), which is equivalent to ! (a < b) - which means testing for a < b and then using `beq` instead of `bne`.

E. An example:

C/C++:
```
if (x == y)
    x ++;
```

MIPS Assembly language - assume that x is in $4 and y in $5:

```
bne     $4, $5, notequal
nop
addiu   $4, $4, 1
```

PROJECT

Encoding of the branch instruction - what must the offset value be?

ASK

2 - address of nop + 2 = instruction following addiu

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

PROJECT

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 n terms of HLL variables which have to be mapped/loaded 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
```

   PROJECT

   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:

3. 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       opcode  rs      rt      offset
values
(decimal)   5       4       5       2

binary      000101  00100   00101   0000000000000010

= 0001 0100 1000 0101 0000 0000 0000 0010
hexadecimal     =  0x14850002
```

   PROJECT

The instruction contains 2 in the offset field because the instruction following the branch is at 0x1004, and0x1004 + (4 x 2) = 0x100c = desired target.

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

B. DISTRIBUTE Control Structures Handout

PROJECT HANDOUT

C. Go over patterns in handout.

D. Do activity as a class, projecting results