EE 109 Unit 14 – Control Flow
CONTROL FLOW

Branch Instructions
Loops and Conditionals
Branch Instructions

- Branches allow us to jump backward or forward in our code
- How? By manipulating the Program Counter (PC)
- Operation: $PC = PC + \text{displacement}$
Branch Instructions

• Conditional Branches
  – Branches only if a particular condition is true
  – Fundamental Instrucs.: BEQ (if equal), BNE (not equal)
  – Syntax: BNE/BEQ Rs, Rt, label
    • Compares Rs, Rt and if EQ/NE, branch to label, else continue

• Unconditional Branches
  – Always branches to a new location in the code
  – Instruction: BEQ $0,$0, label
  – Pseudo-instruction: B label

Some Examples
Two-Operand Compare & Branches

• Two-operand comparison is accomplished using the SLT/SLTI/SLTU (Set If Less-than) instruction followed by a BNE/BEQ
  – Syntax: SLT Rd,Rs,Rt or SLT Rd,Rs,imm
    • If Rs < Rt then Rd = 1, else Rd = 0
  – Use appropriate BNE/BEQ instruction to infer relationship

<table>
<thead>
<tr>
<th>Branch if...</th>
<th>SLT</th>
<th>BNE/BEQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2 &lt; $3</td>
<td>SLT $1,$2,$3</td>
<td>BNE $1,$0,label</td>
</tr>
<tr>
<td>$2 ≤ $3</td>
<td>SLT $1,$3,$2</td>
<td>BEQ $1,$0,label</td>
</tr>
<tr>
<td>$2 &gt; $3</td>
<td>SLT $1,$3,$2</td>
<td>BNE $1,$0,label</td>
</tr>
<tr>
<td>$2 ≥ $3</td>
<td>SLT $1,$2,$3</td>
<td>BEQ $1,$0,label</td>
</tr>
</tbody>
</table>
Branch Pseudo-Instructions

• Rather than writing two instructions (SLT and BNE/BEQ) we can use provided pseudoinstructions

<table>
<thead>
<tr>
<th>Pseudo-instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLT Rt, Rs, label</td>
<td>Branch if less-than</td>
</tr>
<tr>
<td>BLE Rt, Rs, label</td>
<td>Branch if less-than or equal</td>
</tr>
<tr>
<td>BGT Rt, Rs, label</td>
<td>Branch if greater-than</td>
</tr>
<tr>
<td>BGE Rt, Rs, label</td>
<td>Branch if greater-than of equal</td>
</tr>
<tr>
<td>BLTU Rt, Rs, label</td>
<td>Branch if less-than (unsigned)</td>
</tr>
<tr>
<td>BLT Rt, imm, label</td>
<td>Branch if less-than immediate</td>
</tr>
</tbody>
</table>

Note: Pseudoinstructions are assembler-dependent. See MARS Help for more details.
Comparison with SLT

• Performing comparison with the SLT instruction is really accomplished by subtracting A-B and examining the sign of the result
  – if A-B = 0, then A=B
  – if A-B = negative #, then A<B
  – If A-B = positive # and not 0, then A>B

• Determining if the result is positive or negative requires
  – knowing what system is being used
    • signed or unsigned?
  – if overflow occurred
    • when overflow occurs the sign of the result is incorrect (i.e. p+p = n or n+n = p)
SLT/SLTU Operation

• Use appropriate version based on system being used
  – SLT for signed operand
  – SLTU for unsigned operands

• An SLT instruction subtracts A-B and examine sign of the result and the overflow test to determine if it should set the result
Single-Operand Compare & Branches

• BGT, BLT, BGE, BLE are pseudoinstructions
  – Shorthand for an SLT and BEQ
• MIPS does have some single instructions to compare and branch all in one, but only for a single operand compared with 0
• Syntax: BccZ Rt, label
  – cc = {LT, LE, GT, GE}

<table>
<thead>
<tr>
<th>Branch Instruc.</th>
<th>Branch if...</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLTZ $2,label</td>
<td>$2 &lt; 0</td>
</tr>
<tr>
<td>BLEZ $2,label</td>
<td>$2 ≤ 0</td>
</tr>
<tr>
<td>BGTZ $2,label</td>
<td>$2 &gt; 0</td>
</tr>
<tr>
<td>BGEZ $2,label</td>
<td>$2 ≥ 0</td>
</tr>
</tbody>
</table>
TRANSLATING LOOPS & CONDITIONALS
Branch Example 1

C Code

if A > B  (&A in $t0)
    A = A + B  (&B in $t1)
else
    A = 1

MIPS
Assembly

.text
LW  $t2,0($t0)
LW  $t3,0($t1)
SLT $1,$t3,$t2
BEQ $1,$0,ELSE
ADD $t2,$t2,$t3
B  NEXT
ELSE: ADDI $t2,$0,1
NEXT:  SW  $t2,0($t0)
        ----

Could use pseudo-inst. “BLE $4,$5,ELSE”

This branch skips over the “else” portion. This is a pseudo-instruction and is translated to BEQ $0,$0,next
Branch Example 2

C Code

```c
for(i=0;i < 10;i++) ($t0=i)
j = j + i;        ($t1=j)
```

MIPS Assembly

```assembly
.text
ADD     $t0,$0,$0
LOOP:   SLTI $1,$t0,10
        BEQ  $1,$0,NEXT
        ADD  $t1,$t1,$t0
        ADDI $t0,$t0,1
        B     LOOP
NEXT:   ----
```

Branches if i is not less than 10
Loops back to the comparison check
Another Branch Example

C Code

```c
int dat[10];
for(i=0;i < 10;i++)
data[i] = 5;
```

MIPS Assembly

```assembly
.data
dat: .space 40

.text
la $t0,dat
addi $t1,$zero,10
addi $t2,$zero,5

LOOP: sw $t2,0($t0)
addi $t0,$t0,4
addi $t1,$t1,-1
bne $t1,$zero,LOOP

NEXT: ----
```
A Final Example

C Code

```c
char A[] = "hello world";
char B[50];
// strcpy(B,A);
i=0;
while(A[i] != 0){
    B[i] = A[i]; i++;
}
B[i] = 0;
```

MIPS Assembly

```
data
A:  .asciiz "hello world"
B:  .space  50

.text
la $t0,A
la $t1,B
LOOP:  lb $t2,0($t0)
    beq $t2,$zero,NEXT
    sb $t2,0($t1)
    addi $t0,$t0,1
    addi $t1,$t1,1
    b LOOP
NEXT:  sb $t2,0($t1)
```
A Final Example

char A[] = "hello world";
char B[50];

// strcpy(B,A);
while(A[i] != 0){
    B[i] = A[i]; i++;
}
B[i] = 0;

.data
A:     .asciiz "hello world"
B:     .space  50

.text
la      $t0,A
la      $t1,B
LOOP:  lb      $t2,0($t0)
       beq     $t2,$zero,NEXT
       sb      $t2,0($t1)
       addi    $t0,$t0,1
       addi    $t1,$t1,1
       b       LOOP
NEXT:  sb      $t2,0($t1)
DEEPER LOOK AT HOW BRANCHES WORK
Branch Machine Code Format

• Branch instructions use the I-Type Format

<table>
<thead>
<tr>
<th>6-bits</th>
<th>5-bits</th>
<th>5-bits</th>
<th>16-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs (src1)</td>
<td>rt (src2)</td>
<td>Signed displacement</td>
</tr>
</tbody>
</table>

• Operation: PC = PC + \{disp.,00\}

• Displacement notes
  – Displacement is the value that should be added to the PC so that it now points to the desired branch location
  – Processor appends two 0’s to end of disp. since all instructions are 4-byte words
    • Essentially, displacement is in units of words
    • Effective range of displacement is an 18-bit signed value = ±128KB address space (i.e. can’t branch anywhere in memory…but long branches are rare and there is a mechanism to handle them)
Branch Displacement

- To calculate displacement you must know where instructions are stored in memory (relative to each other)
  - Don’t worry, assembler finds displacement for you...you just use the label

MIPS Assembly

```
.text
ADD  $8,$0,$0
ADDI $7,$0,10
LOOP:  SLTI $1,$8,10
BEQ  $1,$0,NEXT
ADD  $9,$9,$8
ADDI $8,$8,1
BEQ  $0,$0,LOOP
NEXT:  ----
```

A
A + 0x4
A + 0x8
A + 0xC
A + 0x10
A + 0x14
A + 0x18
A + 0x1C

ADD
ADDI
SLTI
BEQ
ADD
ADDI
BEQ
----

1 word for each instruction
Calculating Displacements

• Disp. = [(Addr. of Target) – (Addr. of Branch + 4)] / 4
  – Constant 4 is due to the fact that by the time the branch executes the PC will be pointing at the instruction after it (i.e. plus 4 bytes)
• Following slides will show displacement calculation for BEQ $1,$0,NEXT

---

MIPS Assembly

```
.text
ADD     $8,$0,$0
ADDI    $7,$0,10
SLTI    $1,$8,10
BEQ     $1,$0,NEXT
ADD     $9,$9,$8
ADDI    $8,$8,1
BEQ     $8,$8,1
BEQ     $0,$0,LOOP
```

A

A + 0x4
A + 0x8
A + 0xC
A + 0x10
A + 0x14
A + 0x18
A + 0x1C

ADD
ADDI
SLTI
BEQ
ADD
ADDI
BEQ

1 word for each instruction
Calculating Displacements

- Disp. = [(Addr. of Target) - (Addr. of Branch + 4)] / 4
- Disp. = (A+0x1C) - (A+0x0C+ 4) = 0x1C - 0x10 = 0x0C / 4
  = 0x03

### MIPS Assembly

```
.text
ADD $8, $0, $0
ADDI $7, $0, 10
SLTI $1, $8, 10
BEQ $1, $0, NEXT
ADD $9, $9, $8
ADDI $8, $8, 1
BEQ $0, $0, LOOP

LOOP:
ADD $8, $0, $0
ADDI $7, $0, 10
SLTI $1, $8, 10
BEQ $0, $0, LOOP

NEXT: ----
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A + 0x4</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>ADD</td>
<td></td>
</tr>
<tr>
<td>A + 0x8</td>
<td>ADDI</td>
<td></td>
</tr>
<tr>
<td>SLTI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A + 0xC</td>
<td>BEQ</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>ADD</td>
<td></td>
</tr>
<tr>
<td>A + 0x10</td>
<td>ADDI</td>
<td></td>
</tr>
<tr>
<td>BEQ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A + 0x14</td>
<td>BEQ</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A + 0x18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDI</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BEQ</td>
<td></td>
</tr>
<tr>
<td>A + 0x1C</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 word for each instruction
Calculating Displacements

- If the BEQ does in fact branch, it will add the displacement \(\{0x03, 00\} = 0x000C\) to the PC \((A+0x10)\) and thus point to the instruction at the NEXT label \((A+0x1C)\)

MIPS Assembly

```
.text
ADD $8,$0,$0
ADDI $7,$0,10
SLTI $1,$8,10
BEQ $1,$0,NEXT
ADD $9,$9,$8
ADDI $8,$8,1
BEQ $0,$0,LOOP

LOOP: 
BEQ $1,$0,NEXT
ADD $9,$9,$8
ADDI $8,$8,1
BEQ $0,$0,LOOP

NEXT: ----
```

<table>
<thead>
<tr>
<th>Instruction</th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ $1,$0,0x3</td>
<td>000100</td>
<td>00001</td>
<td>00000</td>
<td>0000 0000 0000 0011</td>
</tr>
</tbody>
</table>

Diagram:

- PC (after fetching BEQ): \(A + 0x10\) + \(000C\) = \(A + 0x1C\)
- PC (after adding displacement): \(A + 0x1C\)

- ADD
- ADDI
- SLTI
- BEQ
- ADD
- ADDI
- BEQ
- BEQ

- A
- A + 0x4
- A + 0x8
- A + 0xC
- A + 0x10
- A + 0x14
- A + 0x18
- A + 0x1C
- ----
Another Example

• Disp. = [(Addr. of Label) – (Addr. of Branch + 4)] / 4
• Disp. = $(A+0x04) - (A+0x14 + 4) = 0x04 - 0x18 = 0xFFEC / 4 = 0xFFFFB$

```
.text
ADD $8,$0,$0
LOOP:  SLTI $1,$8,10
        BEQ $1,$0,NEXT
ADD $9,$9,$8
ADDI $8,$8,1
BEQ $0,$0,LOOP
NEXT:  ----
```

BEQ $0,$0,0xFFFFB

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>000100</td>
<td>0000</td>
<td>0000</td>
<td>1111 1111 1111 1011</td>
</tr>
</tbody>
</table>
Jump Instructions

- Jumps provide method of branching beyond range of 16-bit displacement
- Syntax: \( J \) \( label/address \)
  - Operation: \( PC = \) address
  - Address is appended with two 0’s just like branch displacement yielding a 28-bit address with upper 4-bits of PC unaffected
- New instruction format: J-Type
Jump Example

- Take 28 LSB’s of target address, remove LSB’s (which are 0’s) and store 26-bits in the jump instruction

```
.text
ADD     $8,$0,$0
SLTI    $1,$8,10
BEQ     $1,$0,SKIP
J       L1
SKIP:   ADDI   $8,$8,1
        ...
L1:     SUB
```

- **PC before exec. of jump:**
  - 0xE0400000
  - 0xE0400004
  - 0xE0400008
  - 0xE040000C
  - 0xE0400010
  - 0xE0400014
  - 0xEC800004

- **PC after exec. of jump:**
  - 0xE0400000
  - 0xE0400004
  - 0xE0400008
  - 0xE040000C
  - 0xE0400010
  - 0xEC800004

<table>
<thead>
<tr>
<th>opcode</th>
<th>Target Address / 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 10</td>
<td>11 0010 0000 0000 0000 0000 0000 001</td>
</tr>
<tr>
<td></td>
<td>0xC8000004 / 4</td>
</tr>
</tbody>
</table>

- **Instruction Breakdown:**
  - **ADD $8,$0,$0**
  - **SLTI $1,$8,10**
  - **BEQ $1,$0,SKIP**
  - **J L1**
  - **ADDI $8,$8,1**
  - **SUB**

- **opcode:**
  - 0xe0000000
  - 0xe0000004
  - 0xe0000008
  - 0xe000000c
  - 0xe0000010
  - 0xe0000014
  - **0 ecx800000**
Jump Register

• ‘jr’ instruction can be used if a full 32-bit jump is needed or variable jump address is needed

• Syntax: JR rs
  – Operation: PC = R[s]
  – R-Type machine code format

• Usage:
  – Can load rs with an immediate address
  – Can calculate rs for a variable jump (class member functions, switch statements, etc.)
jr Example

- Take whatever value is in the source register and places it in the PC (jumping you to that address)

```assembly
.text
ADD     $8, $0, $0
LUI     $6, 0xffff
ORI     $6, $6, 0x0010
JR      $6
...
```

PC before exec. of jr: 00400010

Value of $6: FFFF0010

PC after exec. of jr: FFFF0010
BACKUP / OLD
Linker Directives

• `.globl label`
  – Allows the following label and data to be referenced from other compilation units (files)

• `.extern label size`
  – Defines an externally allocated (in another file) static data storage with `size` at address `label`

```
.globl main
main:
    la $1, dat1
...

.globl dat1
.data
    .word 0x12345678

.extern dat1 4
.text
    main:
        la $1, dat1
...

.globl dat1
.data
    .word 0x12345678

.extern dat1 4
.data
    .word 0x12345678
```

file1.s  file2.s
SLT/SLTU Operation

• Use appropriate version based on sign
  – SLT for signed operand
  – SLTU for unsigned operands

• An SLT instruction subtracts A-B and examine sign of the result and the overflow test to determine if it should set the result

• Tests to determine less-than (negative) condition
  – < Signed: (Neg. & No OV) OR (Pos. & OV)
  – < Unsigned: (Unsigned OV)

  • For unsigned subtraction, overflow is defined when the result is negative (i.e. Cout = 0)...thus we use that test