CHAPTER XIV

PROGRAM CONTROL, JUMPING, AND BRANCHING

READ BRANCHING FREE-DOC ON COURSE WEBPAGE
• So far we have discussed how the instruction set architecture for a machine can be designed.

• Another important aspect is how to control the flow of a program execution.
  • What order should instructions be executed?
  • Are there times when we need to change the order of instruction execution?
  • How do we handle changes of the program flow and decide when to change the program flow?
How should a program or list of instructions be executed?

- The most obvious choice is to execute the 32-bit instruction words in sequential order.

PC ——- 1st  lw $2, 0x00001004($0)  
         2nd  addi $15, $2, 0x00201003  
         3rd  xor $13, $15, $2  
         4th  add $3 $13 $2  
         5th  sai $18, $3, 0x00000004  
         6th  sw $18, 0x00001003($0)  
         ...  ...

- Would be useful to have a pointer to the next instruction.
- We will call this the program counter (PC).
We can consider the program counter as pointing into memory at the next instruction to be executed.

- Instructions are 32-bits (4 bytes), so add 1 to get next instruction.
To make the memory map representation a little more compact, we will make each address location 32-bits with the PC incremented by 4.

Instruction $n$
(ex: add $3$, $2$, $5$)

Instruction $n + 1$

Instruction $n + 2$

R.M. Dansereau; v.1.0
The PC register can be added as follows to our single cycle DPU.
At the beginning of the clock cycle:

- Current contents of IR used and decoded as the current instruction.
- PC addresses the instruction memory to fetch the next instruction.
- The next instruction is output from the instruction memory and applied to the input of the IR, though, not loaded until the end of the clock cycle.
- PC + 4 is calculated and applied to the PC, though, not loaded until the end of the clock cycle. A +4 is used so that the next 32-bit (4-byte) word is addressed which is the next instruction to be addressed.

At the end of the clock cycle:

- The next instruction is clocked into the IR.
- The address for the following instruction is clocked into the PC.
While executing instructions in **sequential order** is a good **default mode**, it is desirable to be able to **change the program flow**.

- Two main classifications for deviation from sequential order are
  - **absolute** versus **relative** instruction addressing
  - and
  - **conditional** versus **unconditional** branching/jumping
- The MIPS R3000/4000 uses only
  - **unconditional absolute instruction addressing** and
  - **conditional relative instruction addressing**
Absolute instruction addressing, generally known as jumping.

- A specific address, or absolute address, is given where the next instruction is located.
  - PC = address
- This allows execution of any instruction in memory.
- Jumps are good if you have a piece of code that will not be relocated to another location in memory.
  - For instance, ROM BIOS code that never moves.
  - Main interrupt service routines that will always be located in a set instruction memory location.
- Different MIPS instructions will use byte or word addressing such that
  - PC = byte_address or PC = (word_address<<2)
• **Relative instruction addressing**, generally known as **branching**.
  - An **offset to the current address** is given and the next instruction address is calculated, in general, as \( \text{PC} = \text{PC} + \text{byte\_offset} \).
  - For MIPS, and many other processors, since \( \text{PC} \) has already been updated to \( \text{PC} + 4 \) when loading in the current instruction, it is actually calculated as
    - \( \text{PC} = \text{PC} + \text{inst\_size} + \text{inst\_offset} = \text{PC} + 4 + (\text{word\_offset} \ll 2) \)
  - Note that the offset can be **positive** or **negative**.
  - Useful since a program can therefore be loaded anywhere in the instruction memory and still function correctly.
    - Move a program around in memory, and it can still branch within itself since the branching is relative to the current \( \text{PC} \) value.
For unconditional program control instructions

- The absolute jump or relative branch is ALWAYS performed when that instruction is encountered.

For conditional program control instructions

- A condition is first tested.
  - If the result is true, then the branch/jump is taken.
    - $PC = \text{byte\_address}$ or $PC = (\text{word\_address} \ll 2)$ for a jump or
    - $PC = PC + 4 + (\text{word\_offset} \ll 2)$ for a branch.
  - If the result is false, then the branch/jump is NOT taken and program execution continues
    - ie. $PC = PC + 4$. 
JUMPING
JUMP W/ REGISTER (JR)

- The first form of program control is the **absolute jump** is as follows
  - **jr <register>**
  - The **jr** instruction changes **PC** to the value contained in the **register**.
  - For example, if **R10** contains **0x00004400** then after executing the following **jr** instruction, the next instruction executed is the **add**.

```
PC 0x00001000 jr $10
    0x00001004 sub $15, $2, $8
    ...
next PC 0x00004400 add $3 $13 $2
    0x00004404 ...
    ...
```
• We can also have an **immediate** form of the **jump instruction**
  
  • j <instruction address>
  
  • The j instruction changes **PC** to the given **instruction address**.
  
  • For example, with the following j instruction, the next instruction executed is the **add**.

```
PC 0x00001000  j 0x00004400  sub $15, $2, $8
   0x00001004  sub $15, $2, $8
      ...     ...        ...
next PC 0x00004400  add $3 $13 $2
           0x00004404  ...        ...
      ...     ...        ...
```

Note: assembler will convert to 0x0001100 so that 0x0001100<<2=0x00004400.
JUMPING
JR-FORMAT

- Both jump instructions have **one implied destination**, the **PC**, and one source, either a **register** or an **immediate value**.
- We therefore need some new instruction formats.
  - The **jr** instruction can essentially use the **R-format**, but need the **jr** opcode route $Z_{wa}$ to $Y_{ra}$ and route $Y_{bus}$ to the **PC** so that the address in the register is loaded in the **PC**.

  ![JR-format diagram]

  - The jump can go anywhere in memory using the 32-bit register value.
The j instruction only needs the opcode and the immediate address for the new value of the PC.

Unfortunately, the PC is 32 bits and using a 6-bit opcode, this leaves only 26 bits in our 32-bit instruction.

If we assume the immediate address is for 4 byte words, then our 26-bits can effectively address 28-bit bytes.

Update PC with \( PC[27:0] = (\text{word_address} \ll 2) \) leaving \( PC[31:28] \) unchanged. Therefore, cannot jump anywhere in memory, but almost.
As seen, the MIPS R3000/4000 has two basic forms of a jump or absolute instruction addressing.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Assigned Opcode</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>j 0x00004028</td>
<td>000010</td>
<td>The next instruction fetched is at address 0x00004028. Restriction: address is a 26-bit address to 4 byte words.</td>
</tr>
<tr>
<td>jr $10</td>
<td>000011</td>
<td>The next instruction fetched is at the 32-bit address stored in R10</td>
</tr>
</tbody>
</table>
As mentioned, branching uses an offset from the current instruction to determine the next instruction.

For the MIPS, the only branching is with conditional branches.

- Conditional branches typically compare two items, such as two registers, to test a condition.
- This comparison is usually done by simply subtracting one number from the other and setting the appropriate N, V, C, Z flags.
- i.e. for MIPS
  - `<branch mnemonic> <register 1> <register 2> <branch offset>`
  - Here, the calculation `<register 1> - <register 2>` is performed with the flags N, V, C, Z set accordingly (the subtraction result is not stored).
Below is a list of some possible branch types (many of these do not exist for the MIPS R3000/4000).

<table>
<thead>
<tr>
<th>Common Mnemonics</th>
<th>Branch Type</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq</td>
<td>Branch if equal</td>
<td>Z = 1</td>
</tr>
<tr>
<td>bne or bnq</td>
<td>Branch if not equal</td>
<td>Z = 0</td>
</tr>
<tr>
<td>bpl</td>
<td>Branch if positive</td>
<td>N = 0</td>
</tr>
<tr>
<td>bmi</td>
<td>Branch if negative</td>
<td>N = 1</td>
</tr>
<tr>
<td>bcc</td>
<td>Branch on carry clear</td>
<td>C = 0</td>
</tr>
<tr>
<td>bcs</td>
<td>Branch on carry set</td>
<td>C = 1</td>
</tr>
<tr>
<td>bvc</td>
<td>Branch on overflow clear</td>
<td>V = 0</td>
</tr>
<tr>
<td>bvs</td>
<td>Branch on overflow set</td>
<td>V = 1</td>
</tr>
</tbody>
</table>
### BRANCH TYPES

<table>
<thead>
<tr>
<th>Common Mnemonics</th>
<th>Branch Type</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>blt</td>
<td>Branch on less than</td>
<td>N ⊕ V</td>
</tr>
<tr>
<td>ble</td>
<td>Branch on less than or equal</td>
<td>Z + (N ⊕ V)</td>
</tr>
<tr>
<td>bge</td>
<td>Branch on greater than or equal</td>
<td>N ⊕ V</td>
</tr>
<tr>
<td>bgt</td>
<td>Branch on greater</td>
<td>Z + (N ⊕ V)</td>
</tr>
<tr>
<td>bra</td>
<td>Branch always</td>
<td>No flags needed</td>
</tr>
<tr>
<td>bsr</td>
<td>Branch to subroutine</td>
<td>No flags needed</td>
</tr>
</tbody>
</table>
One MIPS instruction is the **branch if equal** (beq) instruction that checks if the contents of two registers are equal and branches if they are equal.

For example, consider the following code:

```
PC → start:       beq $1, $2, skip
   (false) next PC → sub $15, $2, $8
                   ... 
   R1 != R2

   (true) next PC → skip:  add $3 $13 $2
                   ... 
                   ... 
   R1 = R2
```

Notice that the branch is taken if $1 = $2.
Another MIPS instruction is the **branch if not equal** (bne) instruction that checks if two registers are **NOT** equal.

For example, consider the following code:

```
PC  →  start:  bne $1, $2, skip  
    (false) next PC  →  sub $15, $2, $8  
          R1 = R2  
    (true) next PC  →  skip:  add $3 $13 $2  
          R1 != R2  
```

Notice that the branch is taken if $1 != $2.
As seen, the MIPS R3000/4000 has two basic forms of a jump or absolute instruction addressing.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq $10, $8, label</td>
<td>000100</td>
<td>If contents of R10 is equal to contents of R8, next instruction fetched is the instruction labeled “label”. Otherwise, the next instruction fetched is after the beq.</td>
</tr>
<tr>
<td>bne $10, $8, label</td>
<td>000101</td>
<td>If contents of R10 is not equal to contents of R8, next instruction fetched is the instruction labeled “label”. Otherwise, the next instruction fetched is after the bne.</td>
</tr>
</tbody>
</table>
### BRANCHING

#### INSTRUCTION FORMAT

- **Branching requires two sources** for comparison and the **relative offset**.

<table>
<thead>
<tr>
<th>B-format</th>
<th>opcode</th>
<th>Y</th>
<th>X</th>
<th>relative word_offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 25 20 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **This B-format** is effectively the same as the **I-format**.

<table>
<thead>
<tr>
<th>I-format</th>
<th>opcode</th>
<th>Z</th>
<th>X</th>
<th>immediate value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 25 20 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- We can likely make the instruction decoder simpler if we take the **B-format** to be the same as the **I-format**.

- This might take a bit of extra decoding elsewhere in our DPU.
Consider the following pseudo-code for a loop.

<table>
<thead>
<tr>
<th>Pseudo-Code</th>
<th>MIPS Assembly</th>
<th>Register Transfer Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 0</td>
<td>add $26, $0, $0</td>
<td>R26 = 0</td>
</tr>
<tr>
<td>do</td>
<td>add $14, $0, 0x05</td>
<td>R14 = 5</td>
</tr>
<tr>
<td>...</td>
<td>loop: ...</td>
<td></td>
</tr>
<tr>
<td>a = a + 1</td>
<td>add $26, $26, 0x01</td>
<td>R26 = R26 + 1</td>
</tr>
<tr>
<td>while ( a != 5 )</td>
<td>bne $26, $14, loop</td>
<td>PC = PC + 4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>+word_offset&lt;&lt;2</td>
</tr>
</tbody>
</table>

Notice how a conditional branch is used for the while loop.
Consider the following pseudo-code for an **if-then-else** statement.

**Pseudo-Code**

if ( x != y ) then
  ...
  ...
else
  ...
endif

**MIPS Assembly**

(assume x in $5, y in $6)

```
beq $5, $6, else
  ...
  ...
j endif
derif:
else: ...
  ...
endif: ...
```

Notice use of **beq** for **if-then-else** and **j** at end of **if-then**.
Problem with previous slide is that we cannot relocate assembly code because of \textit{j} instruction. Therefore, change assembly as follows.

Pseudo-Code

\begin{verbatim}
if ( x != y ) then
    ...
    ...
else
    ...
    ...
endif
\end{verbatim}

MIPS Assembly (assume x in $5, y in $6)

\begin{verbatim}
beq $5, $6, else
    ...
    ...

else:
    beq $0, $0, endif
    ...
endif:
\end{verbatim}
Another example is given below. Note: $0$ contains 0x00000000.

Pseudo-Code

if (num<0) then
    num = -num
end

if (temperature>=25) then
    activity = “swim”
else
    activity = “cycle”
endif

MIPS Assembly (almost)

lwi $15, num
bge $15, $0, endif0
sub $15, $0, $15
swi $15, num
endif0: lwi $15, temperature
blt $15, 0x0019, else25
swim: ...
j endif25
else25: ...
cycle: ...
endif25: ...
The MIPS processor does not include all of the branches listed in the branch types table. The assembly makes *synthetic instructions* available.

- It actually only has `beq` and `bne` as built-in instructions.
- To perform branches such as `blt`, `ble`, `bgt`, and `bge`, MIPS uses another instruction, `slt` or `slti`, in combination with `beq` or `bne`.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>slt $10, $8, $9</code></td>
<td>101010</td>
<td>If contents of $8 &lt; contents of $9, then $10 = 0x01, else $10 = 0x00.</td>
</tr>
<tr>
<td><code>slti $10,$8, 4</code></td>
<td>001010</td>
<td>If contents of $8 &lt; 4, then $10= 0x01, else $10 = 0x00.</td>
</tr>
</tbody>
</table>
How can **blt**, **ble**, **bgt**, and **bge** effectively be performed using **slt** and **slti**?

<table>
<thead>
<tr>
<th>Desired Instruction</th>
<th>Meaning</th>
<th>Equivalent slt Condition</th>
<th>MIPS Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>blt $10, $8, loop</td>
<td>Branch to loop if $10 &lt; $8</td>
<td>Branch to loop if $10 &lt; $8</td>
<td>slt $5, $10, $8 bne $5, $0, loop</td>
</tr>
<tr>
<td>bge $10, $8, loop</td>
<td>Branch to loop if $10 &gt;= $8</td>
<td>Branch to loop if NOT ($10 &lt; $8)</td>
<td>slt $5, $10, $8 beq $5, $0, loop</td>
</tr>
<tr>
<td>bgt $10, $8, loop</td>
<td>Branch to loop if $10 &gt; $8</td>
<td>Branch to loop if $8 &lt; $10</td>
<td>slt $5, $8, $10 bne $5, $0, loop</td>
</tr>
<tr>
<td>ble $10, $8, loop</td>
<td>Branch to loop if $10 &lt;= $8</td>
<td>Branch to loop if NOT ($8 &lt; $10)</td>
<td>slt $5, $8, $10 beq $5, $0, loop</td>
</tr>
</tbody>
</table>

Note: $0 contains **0x00000000**.
Therefore, our previous example would actually be assembled as:

MIPS Assembly (almost)

```
  lwi $15, num
  bge $15, $0, endif0
  sub $15, $0, $15
  swi $15, num
  endif0: lwi $15, temperature
    blt $15, 0x0019, else25
    swim: ...
    j endif25
  else25: ...
  cycle: ...
  endif25: ...
```

- Therefore, our previous example would actually be assembled as
- MIPS Assembly (almost)
  lwi $15, num
  bge $15, $0, endif0
  sub $15, $0, $15
  swi $15, num
  endif0: lwi $15, temperature
    blt $15, 0x0019, else25
    swim: ...
    j endif25
  else25: ...
  cycle: ...
  endif25: ...

  - slt $5, $15, $0
  - beq $5, $0, endif0
  - slti $5, $15, 0x0019
  - bne $5, $0, else25
• Of course, modifications are needed to the DPU* to allow updating the program counter appropriately with these branches and jumps.

* Note: Not quite accurate for the MIPS architecture.
• Note that branch instructions have **two sources** and an **immediate value**.

• Differs from **I-format** with **one destination**, **one source**, and an **immediate value**.

Not very elegant. Like hammering a square peg into a round hole. But, it solves the special case of a branch.
To calculate jump target consider the following instruction

- \textbf{j 0x00400040}
- The encoding of the jump would be

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
31 & 25 & \multicolumn{2}{c|}{0} \\
\hline
\text{opcode} & \text{instruction word address} & \text{0000 10} & \text{00 0001 0000 0000 0000 0001 0000} \\
\hline
\end{tabular}
\end{center}

which gives an instruction encoding of \textbf{0x08100010} and not \textbf{0x08400040}.

- Why? Because we need to encode with word addresses such that
  - \textbf{0x00100010} \ll 2 = \textbf{0x00400040}
  - This gives the preferred 28-bits over 26-bits.
- Hence, \textbf{PC[27:0] = (word_address \ll 2)}
Now consider the following instruction when $R8=0x00400040$,

- **jr $8**
- For this instruction, since the register $R8$ is already 32-bits, we do not need to perform any shifting of the contents of $R8$.
- Hence, $PC = R8$, which is effectively $PC = 0x00400040$ in the case.
- The encoding of this instruction will look like

<table>
<thead>
<tr>
<th>JR-format</th>
<th>31</th>
<th>25</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>0000 11</td>
<td>01 000</td>
<td>X XXXX XXXX XXXX XXXX XXXX</td>
</tr>
</tbody>
</table>

- This gives an instruction encoding of $0x0D000000$ (for $X=0$).
• For branch target calculation, consider the following code fragment.

\[
\begin{align*}
0x00001000 & \quad \text{beq } $1, $2, \text{ skip} \\
0x00001004 & \quad \text{sub } $15, $2, $8 \\
0x00001008 & \quad \text{...} \\
\text{...} \\
0x00004400 & \quad \text{skip: add } $3 \ $13 \ $2
\end{align*}
\]

• What is the value of the label \textit{skip}? \textbf{skip} = 0x00004400

• We do not want to encode \textit{skip} directly. We need \textbf{word offset}!!

  • \textbf{word offset} = (0x00004400 - (0x00001000 + 0x04)) \gg 2 = 0x0CFF
Using the word offset calculated on the previous slide of we can verify that this is the correct word offset since

\[
PC = PC + 4 + (\text{word offset } \ll 2)
\]

\[
= 0x00001000 + 0x04 + (0x0CFF \ll 2)
\]

\[
= 0x00001000 + 0x04 + 0x000033FC = 0x00004400
\]

Therefore, the instruction encoding for the branch `beq $1, $2, skip` is

<table>
<thead>
<tr>
<th>I-format</th>
<th>opcode</th>
<th>Z</th>
<th>X</th>
<th>word offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0001 00</td>
<td>00 001</td>
<td>00010</td>
<td>0000 1100 1111 1111</td>
</tr>
</tbody>
</table>

This gives an instruction encoding of `0x10220CFF`. 