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, 0x0001003($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 the next instruction.

<table>
<thead>
<tr>
<th>PC</th>
<th>0x80</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0xC9</td>
</tr>
<tr>
<td></td>
<td>0x40</td>
</tr>
<tr>
<td></td>
<td>0x00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC + 4</th>
<th>0x??</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x??</td>
</tr>
<tr>
<td></td>
<td>0x??</td>
</tr>
<tr>
<td></td>
<td>0x??</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC + 8</th>
<th>0x??</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x??</td>
</tr>
<tr>
<td></td>
<td>0x??</td>
</tr>
<tr>
<td></td>
<td>0x??</td>
</tr>
</tbody>
</table>

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

Instruction \( n + 1 \)

Instruction \( n + 2 \)
To make the memory map representation a little more compact, we will make each address location 32-bits with the PC incremented by 4.

```
PC  0x80C94000
PC + 4  0x????????
PC + 8  0x????????
```

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

Instruction $n + 1$

Instruction $n + 2$

•
The PC register can be added as follows to our single cycle DPU.

**Diagram**: The diagram shows the interaction between the Program Counter (PC), IR, RAM instructions, RAM data, and DPU. The PC register is shown being added to other registers and used to access memory and execute instructions. The diagram also includes labels for data and address transfer, indicating the flow of information between components.
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
  - $PC = \text{byte\_address}$ or $PC = (\text{word\_address} \ll 2)$
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 $PC = PC + \text{byte\_offset}$.
- For MIPS, and many other processors, since $PC$ has already been updated to $PC + 4$ when loading in the current instruction, it is actually calculated as
  - $PC = PC + \text{inst\_size} + \text{inst\_offset} = PC + 4 + (\text{word\_offset} \ll 2)$
- 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 $PC$ value.
(UN)CONDITIONAL PROGRAM CONTROL

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 = `byte_address` or PC = `(word_address<<2)` for a jump or
    - PC = PC + 4 + `(word_offset<<2)` for a branch.
  - If the result is **false**, then the branch/jump is NOT taken and program execution continues
    - ie. PC = PC + 4.
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
  
  • \textbf{j} \textless \textit{instruction address}\textgreater
  
  • The j instruction changes \textbf{PC} to the given instruction address.
  
  • For example, with the following j instruction, the next instruction executed is the \textbf{add}.

  \begin{align*}
  \text{PC} & \quad \rightarrow \quad 0x00001000 \quad \text{j} \quad 0x00004400 \\
  0x00001004 & \quad \text{sub} \quad \$15, \quad \$2, \quad \$8 \\
  \cdots & \quad \text{...} \\
  \text{next PC} & \quad \rightarrow \quad 0x00004400 \quad \text{add} \quad \$3 \quad \$13 \quad \$2 \\
  0x00004404 & \quad \text{...} \\
  \cdots & \quad \text{...}
  \end{align*}

  Note: assembler will convert to 0x0001100 so that 0x0001100<<2=0x00004400.
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.

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 $PC[27:0] = (word\_address \ll 2)$ 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 000010</td>
<td>The next instruction fetched is at address 0x00004028. Restriction: address is a 26-bit address to 4 byte words.</td>
<td></td>
</tr>
<tr>
<td>jr $10 000011</td>
<td>The next instruction fetched is at the 32-bit address stored in R10</td>
<td></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.

ie. 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>
## BRANCHING
### 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>$\text{N} \oplus \text{V}$</td>
</tr>
<tr>
<td>ble</td>
<td>Branch on less than or equal</td>
<td>$\text{Z} + (\text{N} \oplus \text{V})$</td>
</tr>
<tr>
<td>bge</td>
<td>Branch on greater than or equal</td>
<td>$\overline{\text{N}} \oplus \text{V}$</td>
</tr>
<tr>
<td>bgt</td>
<td>Branch on greater</td>
<td>$\overline{\text{Z}} + (\text{N} \oplus \text{V})$</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>

- continued...
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
    sub $15, $2, $8
    ...
(false) next PC  skip: add $3 $13 $2
    R1 = R2
    ...
    ...
(true) next PC
    R1 != R2
    ...
```

Notice that the branch is taken if $1 \neq $2.
As seen, the MIPS R3000/4000 has two basic forms of a branch instruction.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>beq $10, $8, label</code></td>
<td>000100</td>
<td>If contents of R10 is equal to contents of R8, next instruction that is fetched is the instruction labeled “label”. Otherwise, the next instruction fetched is after the beq.</td>
</tr>
<tr>
<td><code>bne $10, $8, label</code></td>
<td>000101</td>
<td>If contents of R10 is not equal to contents of R8, next instruction that is fetched is the instruction labeled “label”. Otherwise, the next instruction fetched is after the bne.</td>
</tr>
</tbody>
</table>
Branching requires **two sources** for comparison and the **relative offset**.

**B-format**

- 31 25 20 15 0
- Opcode | Y | X | Relative word_offset

**I-format**

- 31 25 20 15 0
- Opcode | Z | X | Immediate value

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

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.

Pseudo-Code

```
a = 0
do
  ...
  a = a + 1
while ( a != 5 )
  ...
```

MIPS Assembly

```
add $26, $0, $0
add $14, $0, 0x05
```

Register Transfer Notation

```
R26 = 0
R14 = 5
```

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

**Pseudo-Code**

\begin{align*}
\text{if ( x \neq y ) then} & \\
\text{...} & \\
\text{...} & \\
\text{else} & \\
\text{...} & \\
\text{...} & \\
\text{endif}
\end{align*}

**MIPS Assembly**

\begin{align*}
\text{beq}\ \$5,\ \$6,\ \text{else} & \\
\text{...} & \\
\text{...} & \\
\text{...} & \\
\text{j endif} & \\
\text{else: \ ...} & \\
\text{...} & \\
\text{endif: \ ...}
\end{align*}

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

Pseudo-Code

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

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

beq $5, $6, else
  ...
else: ...
beq $0, $0, endif
  ...
else:
endif: ...
A higher language example of program control is given below. Note: $0$ contains 0x00000000.

**Pseudo-Code**

```plaintext
if (num<0) then
    num = -num
end
if (temperature>=25) then
    activity = “swim”
else
    activity = “cycle”
endif
```

**MIPS Assembly (almost)**

```assembly
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: ...
```

R.M. Dansereau; v.1.0
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 \texttt{blt}, \texttt{ble}, \texttt{bgt}, and \texttt{bge} effectively be performed using \texttt{slt} and \texttt{slti}?

<table>
<thead>
<tr>
<th>Desired Instruction</th>
<th>Meaning</th>
<th>Equivalent s</th>
<th>t Condition</th>
<th>MIPS Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{blt} $10, $8, loop</td>
<td>Branch to loop if $10 &lt; $8</td>
<td>Branch to loop if $10 &lt; $8</td>
<td>\texttt{slt} $5, $10, $8 \texttt{bne} $5, $0, loop</td>
<td></td>
</tr>
<tr>
<td>\texttt{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>\texttt{slt} $5, $10, $8 \texttt{beq} $5, $0, loop</td>
<td></td>
</tr>
<tr>
<td>\texttt{bgt} $10, $8, loop</td>
<td>Branch to loop if $10 &gt; $8</td>
<td>Branch to loop if $8 &lt; $10</td>
<td>\texttt{slt} $5, $8, $10 \texttt{bne} $5, $0, loop</td>
<td></td>
</tr>
<tr>
<td>\texttt{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>\texttt{slt} $5, $8, $10 \texttt{beq} $5, $0, loop</td>
<td></td>
</tr>
</tbody>
</table>

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

MIPS Assembly (almost)

\[
\begin{align*}
\text{lwi} & \; $15, \; \text{num} \\
\text{bge} & \; $15, \; $0, \; \text{endif0} \\
\text{sub} & \; $15, \; $0, \; $15 \\
\text{swi} & \; $15, \; \text{num} \\
\text{endif0:} & \; \text{lwi} \; $15, \; \text{temperature} \\
\text{blt} & \; $15, \; 0x0019, \; \text{else25} \\
\text{swim:} & \; ... \\
\text{j} & \; \text{endif25} \\
\text{else25:} & \; ... \\
\text{cycle:} & \; ... \\
\text{endif25:} & \; ...
\end{align*}
\]

\[
\begin{align*}
\{ \text{slt} \; $5, \; $15, \; $0, \text{endif0} \\
\text{beq} \; $5, \; $0, \; \text{endif0} \\
\{ \text{slti} \; $5, \; $15, \; 0x0019 \\
\text{bne} \; $5, \; $0, \; \text{else25} \\
\text{endif25:} & \; ...
\end{align*}
\]
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

- \texttt{j 0x00400040}

- The encoding of the jump would be

\[
\begin{array}{c|c|c}
31 & 25 & 0 \\
\hline
\text{opcode} & \text{instruction word address} & \\
0000 & 00 & 0001 0000 0000 0000 0000 0001 0000 \\
\end{array}
\]

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

- Why? Because we need to encode with word addresses such that
  - \texttt{0x00100010} = 2 = \texttt{0x00400040}
  - This gives the preferred 28-bits over 26-bits.
  - Hence, \( \text{PC}[27:0] = (\text{word\_address} \ll 2) \)
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

   JR-format

   31  25  20  0
   opcode  Z
   0000 11 01 000
   X XXXX XXXX XXXX XXXX XXXX XXXX

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

```
0x00001000  beq $1, $2, skip
0x00001004  sub $15, $2, $8
0x00001008  ...
...
0x00004400  skip:  add $3 $13 $2
```

- What is the value of the label `skip`?  `skip = 0x00004400`
- We do not want to encode `skip` directly. We need `word offset`!!
  - `word offset = (0x00004400 - (0x00001000 + 0x04)) >> 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

| 31 | 25 | 20 | 15 | \multicolumn{3}{c}{word offset} |
|----|----|----|----|-----------------|-----------------|-----------------|
|    |    |    |    | opcode  | \text{Z} | \text{X} | \text{word offset} |
|    |    |    |    | 0001 00 | 00 001 | 0 0010 | 0000 1100 1111 1111 |

• This gives an instruction encoding of \text{0x10220CFF}.  