CHAPTER XIII

INSTRUCTION SET ARCHITECTURE (ISA)

READ INSTRUCTIONS FREE-DOC ON COURSE WEBPAGE
• We have now considered the beginnings of the internal architecture of a computer.

• With this, we considered microcode operations for performing simple data routing and calculations in one clock cycle.

• As a programmer, we don’t want to interface with the microprocessor and manually send each and every control signal as is done with microcode.

• We would prefer to abstract the instruction sent to the microprocessor.

• Let the microprocessor designer handle the decoding of the abstracted instruction into the microcode control operations.

• Start to define an assembly language! MIPS R3000/4000!
Below is the process for translating a program to machine opcodes.

Compiler translates program

High level program
  e.g. C, C++, Pascal, Java

Assembler converts to machine code

Machine instructions
Once the opcodes are given to the microprocessor, it translates the opcode instructions to the microcodes operations we discussed.

Machine opcodes sent to microprocessor

Instruction decoder translates opcodes to the microcodes

Microprocessor

Instruction Decoder

DPU

Machine Instructions

01011000101011101001001
1001010100110101110110100101110100101010111011
001011110100101010111011
For MIPS assembly, many registers have alternate names or specific uses.

<table>
<thead>
<tr>
<th>Register</th>
<th>Name(s)</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$zero</td>
<td>always zero (0x00000000)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>reserved for assembler</td>
</tr>
<tr>
<td>2-3</td>
<td>$v0-$v1</td>
<td>results and expression evaluation</td>
</tr>
<tr>
<td>4-7</td>
<td>$a0-$a3</td>
<td>arguments</td>
</tr>
<tr>
<td>8-15</td>
<td>$t0-$t7</td>
<td>temporary values</td>
</tr>
<tr>
<td>16-23</td>
<td>$s0-$s7</td>
<td>saved values</td>
</tr>
<tr>
<td>24-25</td>
<td>$t8-$t9</td>
<td>temporary values</td>
</tr>
<tr>
<td>26-27</td>
<td></td>
<td>reserved for operating system</td>
</tr>
<tr>
<td>28</td>
<td>$gp</td>
<td>global pointer</td>
</tr>
<tr>
<td>29</td>
<td>$sp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>$fp</td>
<td>frame pointer</td>
</tr>
<tr>
<td>31</td>
<td>$ra</td>
<td>return address</td>
</tr>
</tbody>
</table>
• Need to consider an assembly language example. We will use the MIPS R3000/4000 assembly so that you can refer to the Instruction free-doc.

• MIPS R3000/4000 assembly instruction format:
  • The *majority* of MIPS instructions have the following assembly language instruction format.
    • `<inst mnemonic> <destination>, <source 1>, <source 2>`
  • You can see that this instruction format fits the register transfer level notation discussed with the single cycle DPU

\[
R18 = R12 + R15
\]
- Register format (R-format) instructions
  - Many MIPS instructions have the following format for register to register type binary operations.
    - `<instr> $<write register>, $<read register 1>, $<read register 2>`
  - An example of this is
    - `add $10, $8, $9`
  - This is the same as with our register transfer level operation
    - `R10 = R8 + R9`
Below is the basic list of register format MIPS instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $10, $8, $9</td>
<td>R10 = R8 + R9</td>
</tr>
<tr>
<td>sub $10, $8, $9</td>
<td>R10 = R8 - R9</td>
</tr>
<tr>
<td>and $10, $8, $9</td>
<td>R10 = R8 and R9</td>
</tr>
<tr>
<td>or $10, $8, $9</td>
<td>R10 = R8 or R9</td>
</tr>
<tr>
<td>xor $10, $8, $9</td>
<td>R10 = R8 xor R9</td>
</tr>
<tr>
<td>sa $10, $8, $9 (shift arithmetic)</td>
<td>Shift R8 by R9 and store in R10</td>
</tr>
<tr>
<td>sl $10, $8, $9 (shift logical)</td>
<td>Shift R8 by R9 and store in R10</td>
</tr>
<tr>
<td>rot $10, $8, $9 (rotate)</td>
<td>Rotate R8 by R9 and store in R10</td>
</tr>
<tr>
<td>lw $10, 0($8)</td>
<td>R10 = M[0+R8]</td>
</tr>
<tr>
<td>sw $10, 0($8)</td>
<td>M[0+R8] = R10</td>
</tr>
</tbody>
</table>
• Immediate format (I-format) instructions

  • Many MIPS instructions have the following format for register to register type binary operations.
    • `<instr> $<write register>, $<read register>, <immediate value>`

    An example of this is
    • `addi $10, $8, 4`

    Note: No $ for last argument

    • An example of this is
      • `addi $10, $8, 4`

      Again, no $ for immediate value

      Note: Include “i” to indicate an immediate value is used.

    • This is the same as with our register transfer level operation
      • `R10 = R8 + 4`
### MIPS ASSEMBLY

#### IMMEDIATE INSTRUCTIONS

Below is the basic list of immediate format MIPS instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $10, $8, 4</td>
<td>R10 = R8 + 4</td>
</tr>
<tr>
<td>subi $10, $8, 4</td>
<td>R10 = R8 - 4</td>
</tr>
<tr>
<td>andi $10, $8, 4</td>
<td>R10 = R8 and 4</td>
</tr>
<tr>
<td>ori $10, $8, 4</td>
<td>R10 = R8 or 4</td>
</tr>
<tr>
<td>xori $10, $8, 4</td>
<td>R10 = R8 xor 4</td>
</tr>
<tr>
<td>sai $10, $8, 4 (shift arithmetic)</td>
<td>Shift R8 by 4 and store in R10</td>
</tr>
<tr>
<td>sli $10, $8, 4 (shift logical)</td>
<td>Shift R8 by 4 and store in R10</td>
</tr>
<tr>
<td>roti $10, $8, 4 (rotate)</td>
<td>Rotate R8 by 4 and store in R10</td>
</tr>
<tr>
<td>lw $10, 4($0)</td>
<td>R10 = M[4+R0]</td>
</tr>
<tr>
<td>sw $10, 4($0)</td>
<td>M[4+R0] = R10</td>
</tr>
</tbody>
</table>
• How should the assembly be translated to machine code?

• Have to consider what control signals the DPU requires!

• How do we abstract from the DPU’s requirements?
First important part of a machine instruction is known as the operational codes (opcodes).

- An **opcode** indicates what **major operation** to perform.
  - Example major operations:
    - add, subtract, AND, OR, NOT, XOR, shift
- Once all major operations are identified for a processor design, **assign binary codes** to each of the operation.
  - For example, say that we want to design a machine that can perform 40 different types of major operations.
  - Then we would require at least 6 bits to represent all of the opcodes.
Some example opcodes used in the MIPS processors are as follows.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Assigned Opcode Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $10, $8, $9</td>
<td>100000</td>
</tr>
<tr>
<td>sub $10, $8, $9</td>
<td>100010</td>
</tr>
<tr>
<td>and $10, $8, $9</td>
<td>100100</td>
</tr>
<tr>
<td>or $10, $8, $9</td>
<td>100101</td>
</tr>
<tr>
<td>lw $10, 0($8)</td>
<td>100011</td>
</tr>
<tr>
<td>sw $10, 0($8)</td>
<td>101011</td>
</tr>
<tr>
<td>addi $10, $8, 4</td>
<td>001000</td>
</tr>
<tr>
<td>nop (no operation)</td>
<td>000000</td>
</tr>
</tbody>
</table>

Note: Different opcodes for `add` and `addi`. Why?
• Once you have assigned opcodes to all of your major functions, now need to decode the opcodes to the appropriate controller signals.
  • i.e. we no longer want to control the DPU manually.
• Recall that we used the following DPU signals when performing

\[
R10 = R8 + R9
\]

• \(a/s = 0\) and \(en = 1\) for AU.
• \(en = 0\) for LU, \(en = 0\) for SU.
• \(st_{-}en = 0\), \(ld_{-}en = 0\), \(r/w = X\), \(mssel = 0\).
• \(X_{ra} = 01000\), \(Y_{ra} = 01001\), \(Z_{wa} = 01010\), and \(rwe = 1\) for RF.
• Note: We will pass \(X_{ra}\), \(Y_{ra}\), and \(Z_{wa}\) from the outside.
• Refer to Table 3 in Instruction free-doc for other examples.
In general, these control signals can be burned into a ROM.

- Each opcode has its own set of general control signals for the DPU.
For our DPU, the control signals are as follows.

**opcode**

![Diagram of opcode with signals: rwe, imm en, au en, a/s, lu en, If (4 bits), su en, st (2 bits), st en, ld en, r/w, msel, rom (opcodes)]
• Now, an input opcode will send appropriate control signals to the DPU for that major operation.

Notice that we still need register addresses and the immediate value.
While instructions can come in many different shapes and forms, we will consider the following 32-bit instruction formats to loosely follow the MIPS R3000/4000 format.

**R-format**

<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>20</th>
<th>15</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>Z</td>
<td>X</td>
<td>Y</td>
<td>other potential bits</td>
<td></td>
</tr>
</tbody>
</table>

**I-format**

<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>20</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>Z</td>
<td>X</td>
<td>immediate value</td>
<td></td>
</tr>
</tbody>
</table>
• If we have an R-format instruction, we link the bits as follows.

```
<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>20</th>
<th>15</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>Z</td>
<td>X</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

If we have an R-format instruction, we link the bits as follows:

- **Opcode**: Z
- **X**: do
- **Y**: raZ
- **32x32**: ra
- **32**: X
- **Z**: wa
- **im en**: im
- **im va**: va
- **32**: immediate register
- **addr**: data
- **DPU**:
- **Z bus**:
- **X bus**:
- **Y bus**:
- **RF**:
- **Zdi**:
- **rwe**:
- **32**:
- **Clk**:

**ROM Opcodes**
If we have an I-format instruction, we link the bits as follows.

If we have an I-format instruction, we link the bits as follows.

- **Opcode**
- **Z**
- **X**
- **Immediate**

Notice sign extension of 16-bit value.

DPU

- **RWE**
- **R**
- **32x32**
- **X_do**
- **RF**
- **Y_do**
- **Zi**
- **Zwa**
- **Xra**
- **Yra**

DPU

- **Addr**
- **Data**

**Z bus**

**X bus**

**Y bus**
• Use a general instruction register that can act as R- or I-Format.