

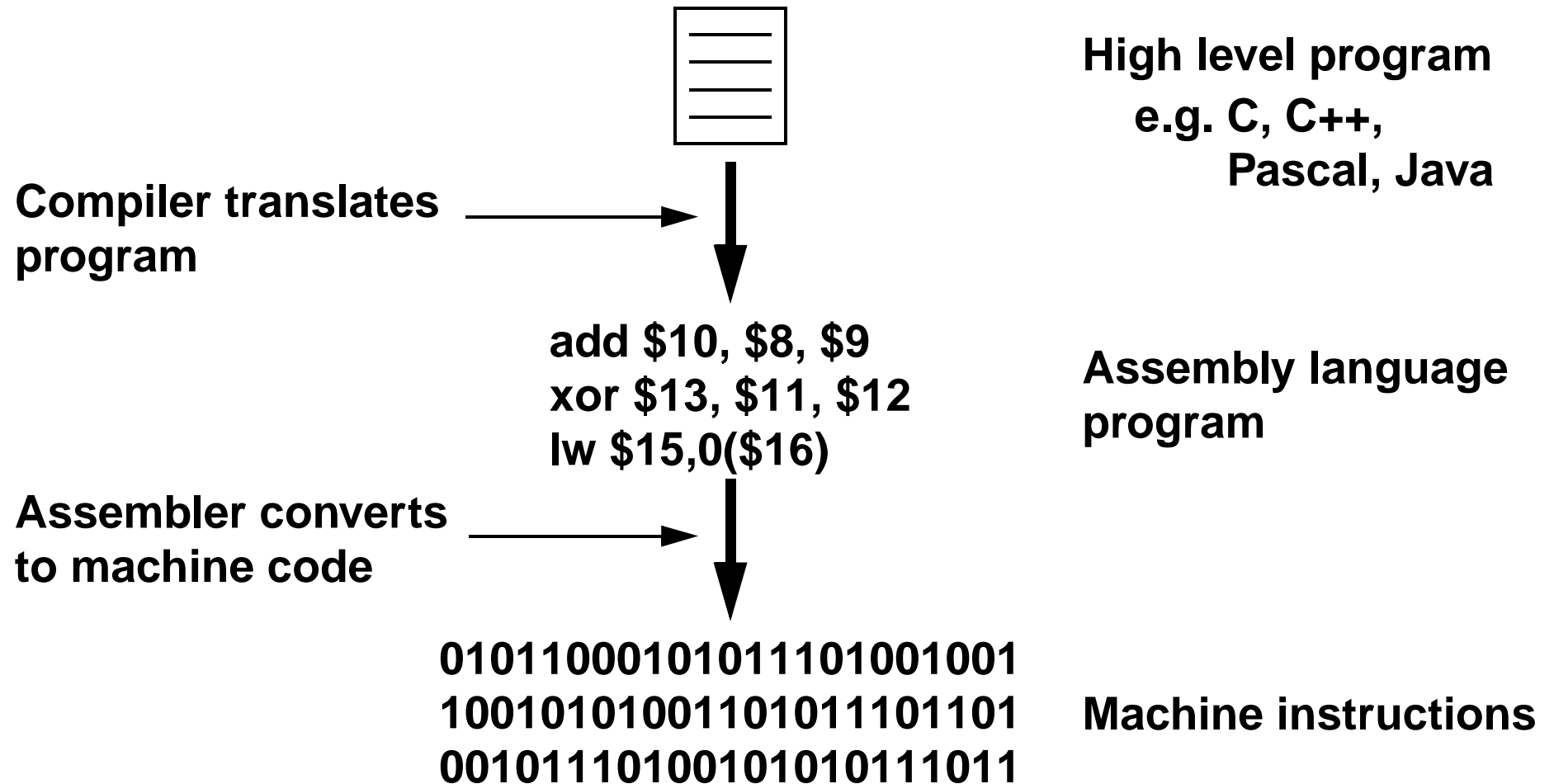
# **CHAPTER XIII**

## **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!

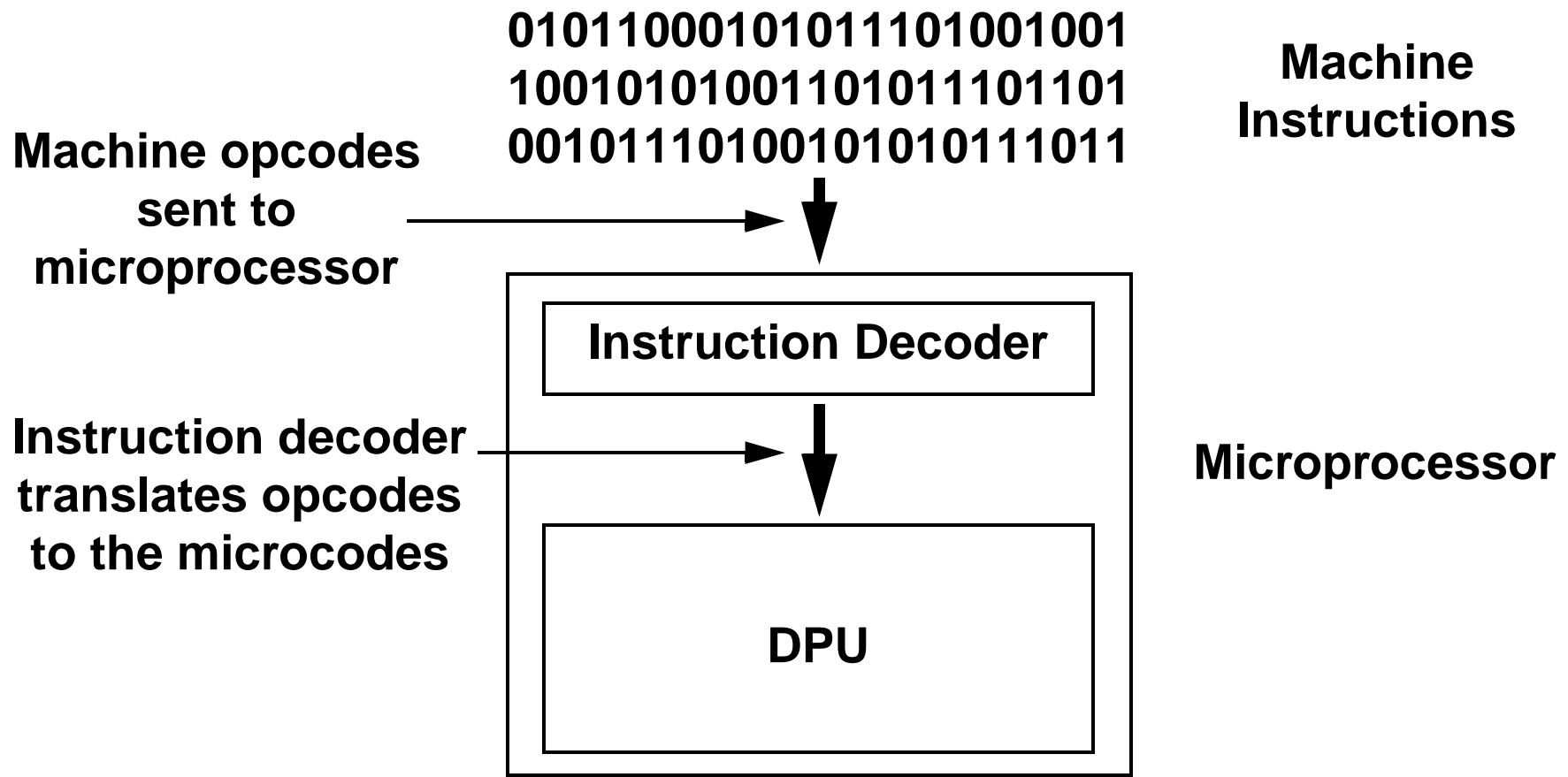
- Below is the process for translating a program to machine opcodes.



# PROGRAM PATH

## EXECUTING CODE

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



# **MIPS ASSEMBLY**

## **MIPS REGISTER NAMES**

- For MIPS assembly, many registers have alternate names or specific uses.

<b>Register</b>	<b>Name(s)</b>	<b>Use</b>
<b>0</b>	<b>\$zero</b>	<b>always zero (0x00000000)</b>
<b>1</b>		<b>reserved for assembler</b>
<b>2-3</b>	<b>\$v0-\$v1</b>	<b>results and expression evaluation</b>
<b>4-7</b>	<b>\$a0-\$a3</b>	<b>arguments</b>
<b>8-15</b>	<b>\$t0-\$t7</b>	<b>temporary values</b>
<b>16-23</b>	<b>\$s0-\$s7</b>	<b>saved values</b>
<b>24-25</b>	<b>\$t8-\$t9</b>	<b>temporary values</b>
<b>26-27</b>		<b>reserved for operating system</b>
<b>28</b>	<b>\$gp</b>	<b>global pointer</b>
<b>29</b>	<b>\$sp</b>	<b>stack pointer</b>
<b>30</b>	<b>\$fp</b>	<b>frame pointer</b>
<b>31</b>	<b>\$ra</b>	<b>return address</b>

- 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**

**destination**      **source 1**      **source 2**

- 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**

# **MIPS ASSEMBLY**

## **REGISTER INSTRUCTIONS**

- Below is the basic list of register format MIPS instructions.

<b>Instruction</b>	<b>Interpretation</b>
<b>add \$10, \$8, \$9</b>	<b><math>R10 = R8 + R9</math></b>
<b>sub \$10, \$8, \$9</b>	<b><math>R10 = R8 - R9</math></b>
<b>and \$10, \$8, \$9</b>	<b><math>R10 = R8 \text{ and } R9</math></b>
<b>or \$10, \$8, \$9</b>	<b><math>R10 = R8 \text{ or } R9</math></b>
<b>xor \$10, \$8, \$9</b>	<b><math>R10 = R8 \text{ xor } R9</math></b>
<b>sa \$10, \$8, \$9 (shift arithmetic)</b>	<b>Shift R8 by R9 and store in R10</b>
<b>sl \$10, \$8, \$9 (shift logical)</b>	<b>Shift R8 by R9 and store in R10</b>
<b>rot \$10, \$8, \$9 (rotate)</b>	<b>Rotate R8 by R9 and store in R10</b>
<b>lw \$10, 0(\$8)</b>	<b><math>R10 = M[0+R8]</math></b>
<b>sw \$10, 0(\$8)</b>	<b><math>M[0+R8] = R10</math></b>

- 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>**

**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**

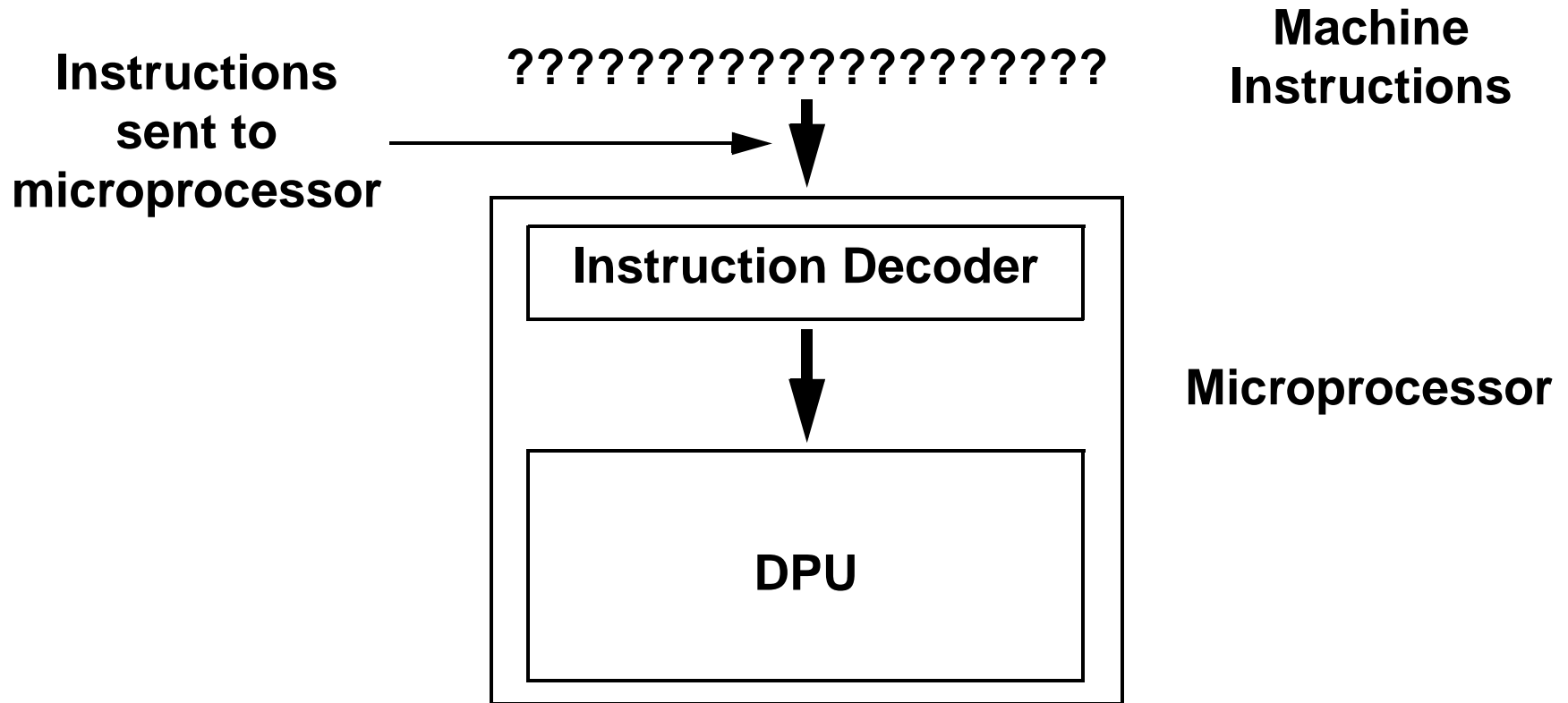
# MIPS ASSEMBLY

## IMMEDIATE INSTRUCTIONS

- Below is the basic list of immediate format MIPS instructions.

Instruction	Interpretation
<b>addi \$10, \$8, 4</b>	<b><math>R10 = R8 + 4</math></b>
<b>subi \$10, \$8, 4</b>	<b><math>R10 = R8 - 4</math></b>
<b>andi \$10, \$8, 4</b>	<b><math>R10 = R8 \text{ and } 4</math></b>
<b>ori \$10, \$8, 4</b>	<b><math>R10 = R8 \text{ or } 4</math></b>
<b>xori \$10, \$8, 4</b>	<b><math>R10 = R8 \text{ xor } 4</math></b>
<b>sai \$10, \$8, 4 (shift arithmetic)</b>	<b>Shift R8 by 4 and store in R10</b>
<b>sli \$10, \$8, 4 (shift logical)</b>	<b>Shift R8 by 4 and store in R10</b>
<b>roti \$10, \$8, 4 (rotate)</b>	<b>Rotate R8 by 4 and store in R10</b>
<b>lw \$10, 4(\$0)</b>	<b><math>R10 = M[4+R0]</math></b>
<b>sw \$10, 4(\$0)</b>	<b><math>M[4+R0] = R10</math></b>

- 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.

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

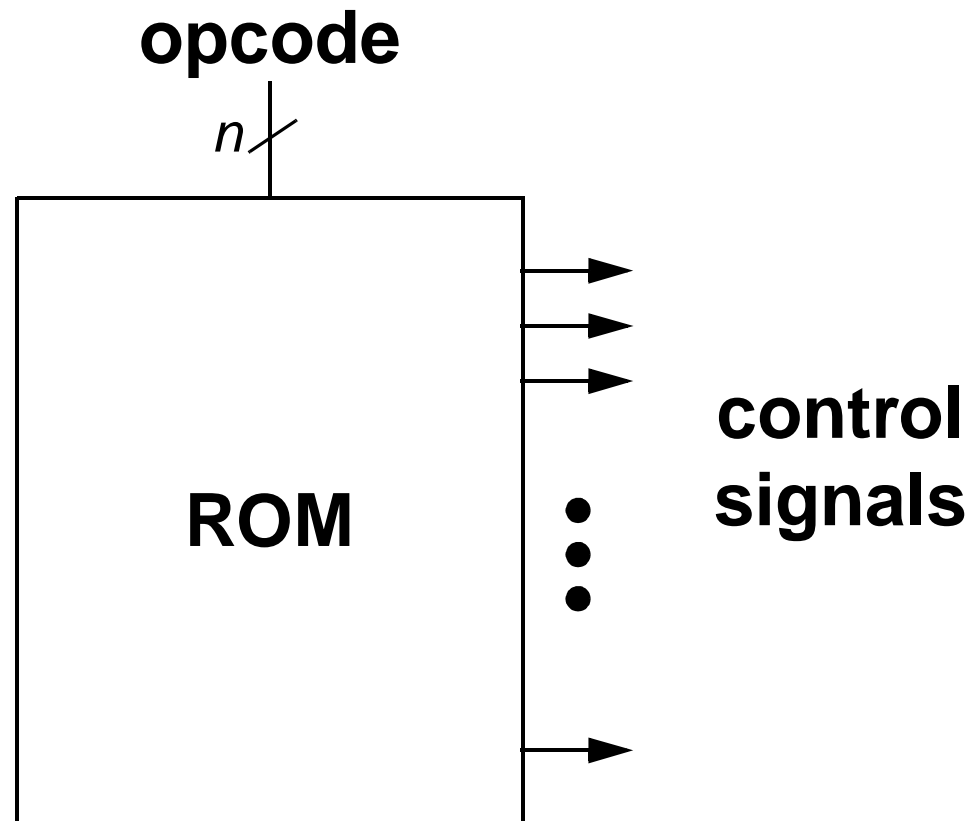
- Note: Different opcodes for **add** and **addi**. Why?

- 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

$$\mathbf{R10 = R8 + R9}$$

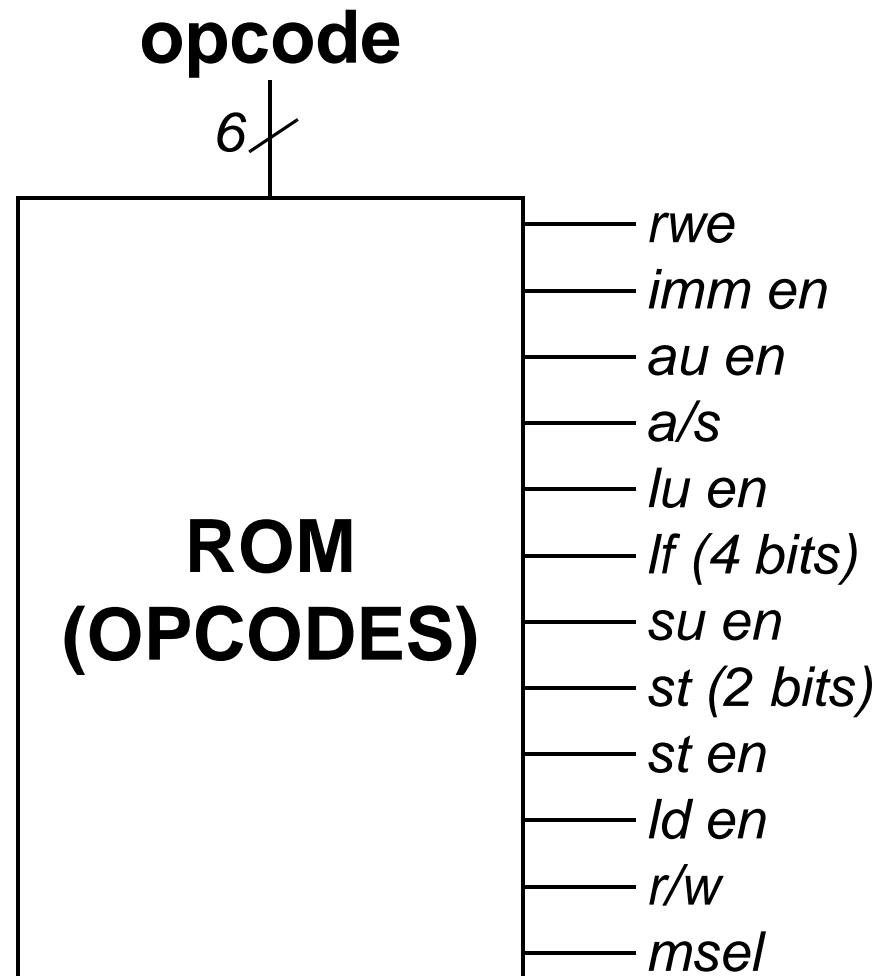
- $\bar{a}/s = 0$  and  $en = 1$  for **AU**.
- $en = 0$  for **LU**,  $en = 0$  for **SU**.
- $st\_en = 0$ ,  $ld\_en = 0$ ,  $r/w = X$ ,  $mselect = 0$ .
- $X_{ra} = 01000$ ,  $Y_{ra} = 01001$ ,  $Z_{wa} = 01010$ , and  $rwe = 1$  for **RF**.
- Note: We will pass  $X_{ra}$ ,  $Y_{ra}$ , and  $Z_{wa}$  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.

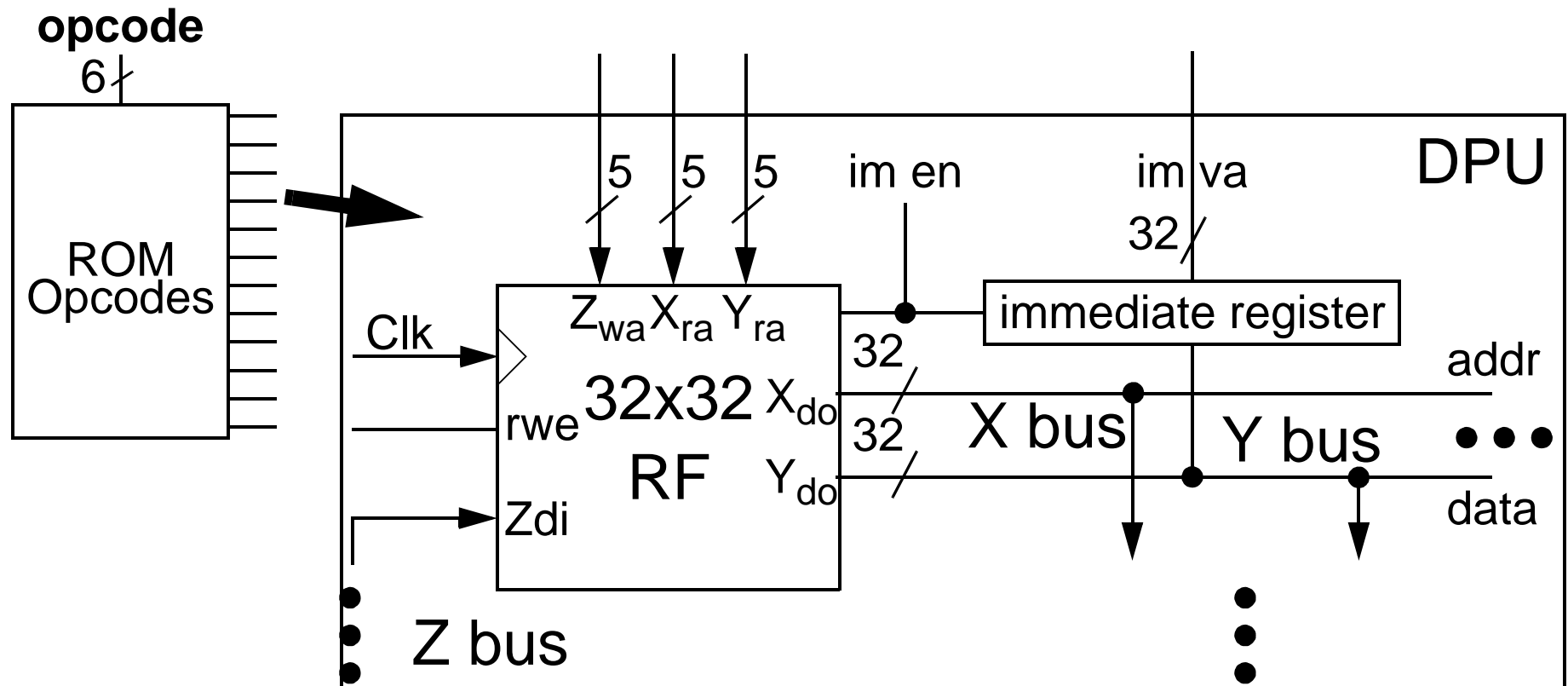


- Each opcode has its own set of general control signals for the DPU.

- For our DPU, the control signals are as follows.



- Now, an input opcode will send appropriate control signals to the DPU for that major operation.

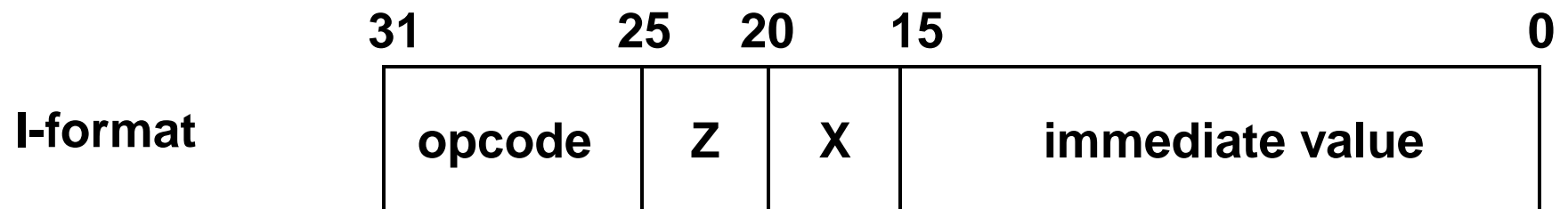
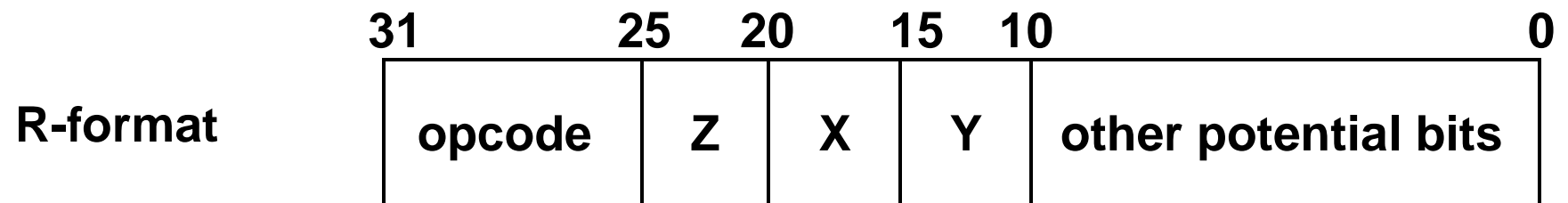


- Notice that we still need register addresses and the immediate value.

# **INSTRUCTIONS**

## **INSTRUCTION FORMATS**

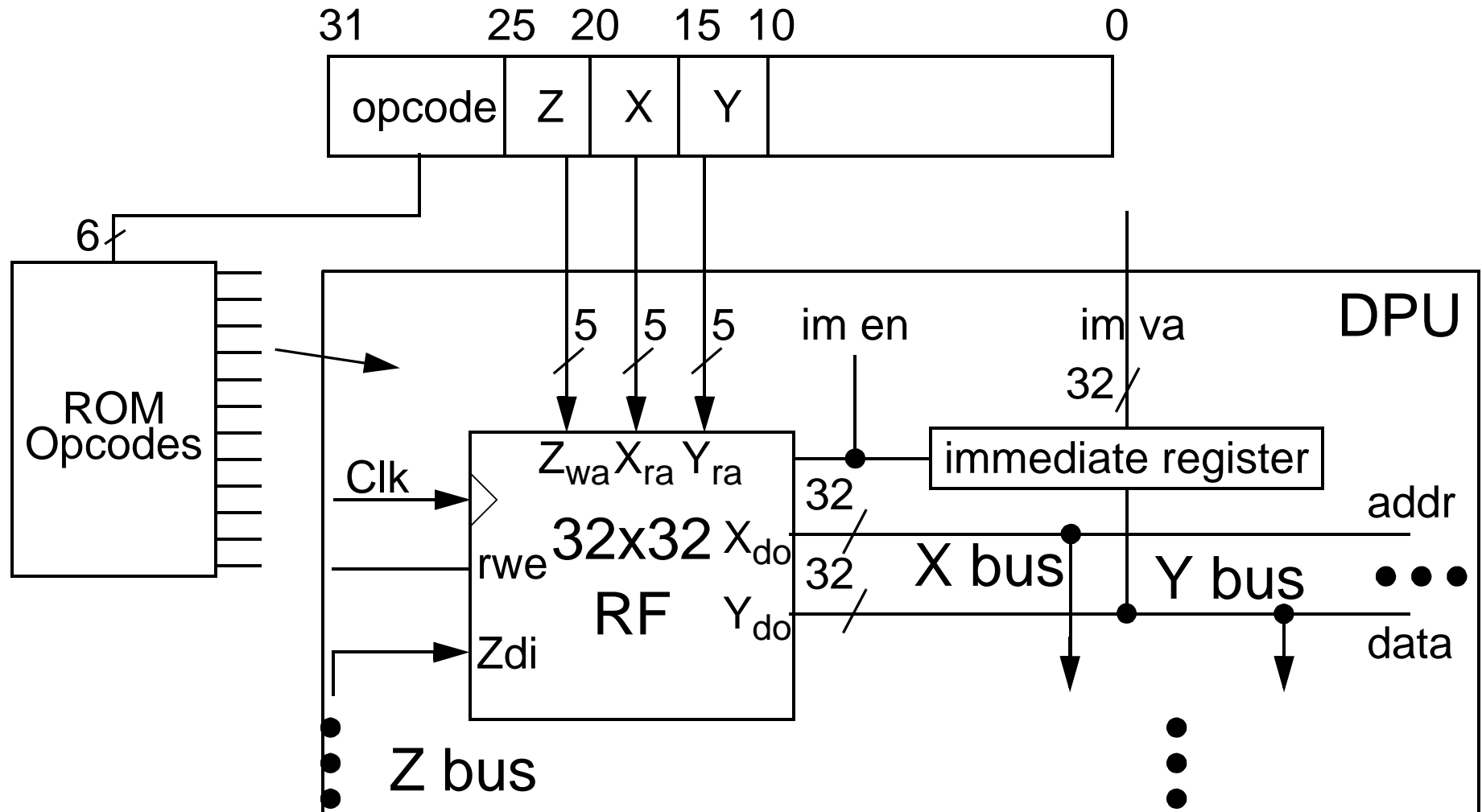
- While instructions can come in many different shapes and forms, we will consider the following 32-bit instruction formats to loosely follow the MIPS R3000/4000 format.



# INSTRUCTIONS

## R-FORMAT W/ DPU

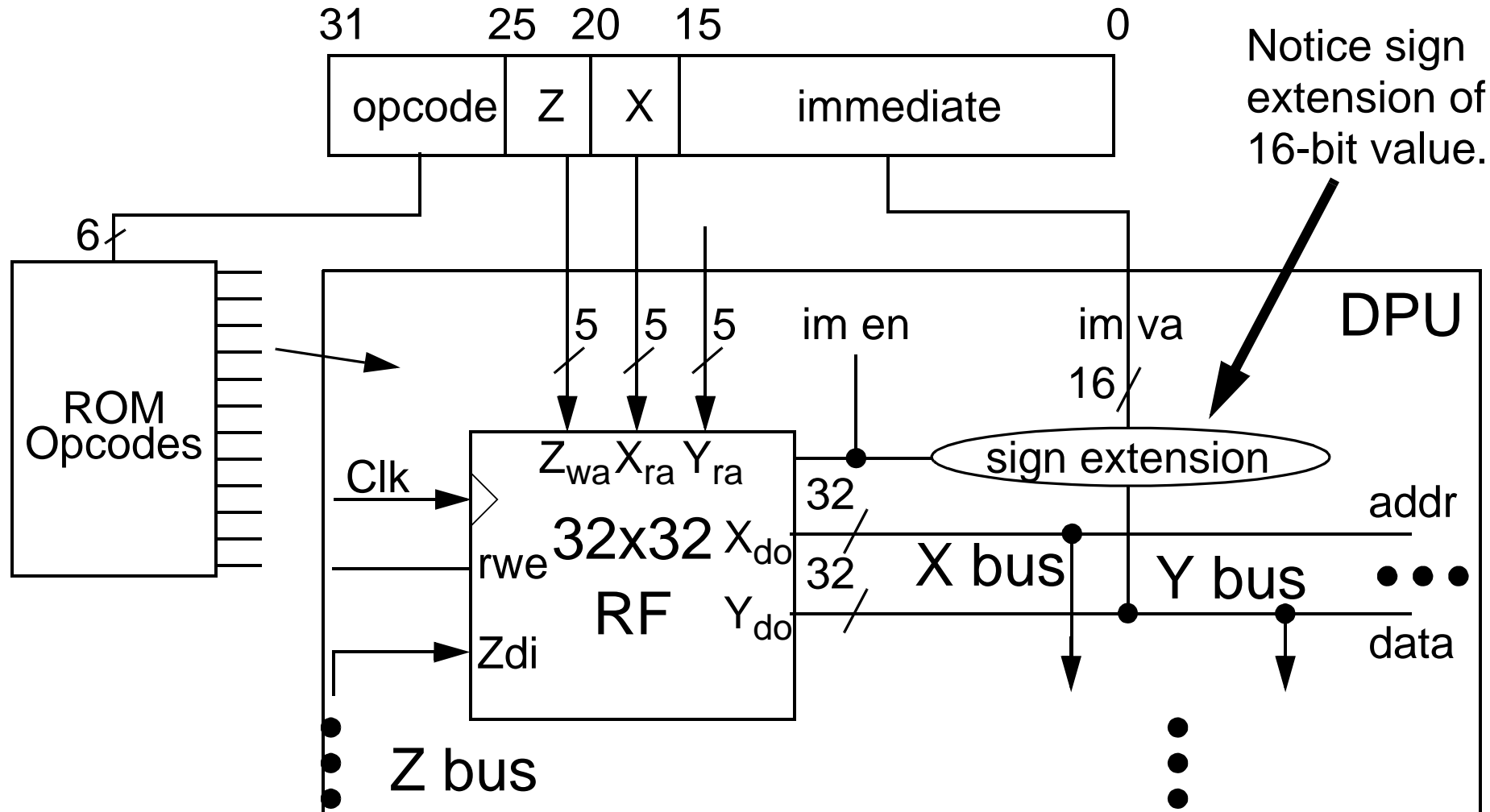
- If we have an R-format instruction, we link the bits as follows.



# INSTRUCTIONS

## I-FORMAT W/ DPU

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



# INSTRUCTIONS

## INSTRUCTION REGISTER

- Use a general instruction register that can act as R- or I-Format.

