CHAPTER XIV

PROGRAM CONTROL, JUMPING, AND BRANCHING

READ BRANCHING FREE-DOC ON COURSE WEBPAGE

PROGRAM CONTROL

INTRODUCTION

- 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?

PROGRAM CONTROL

PROGRAM COUNTER (PC)

- **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)
	...	...



Standard Order of Execution

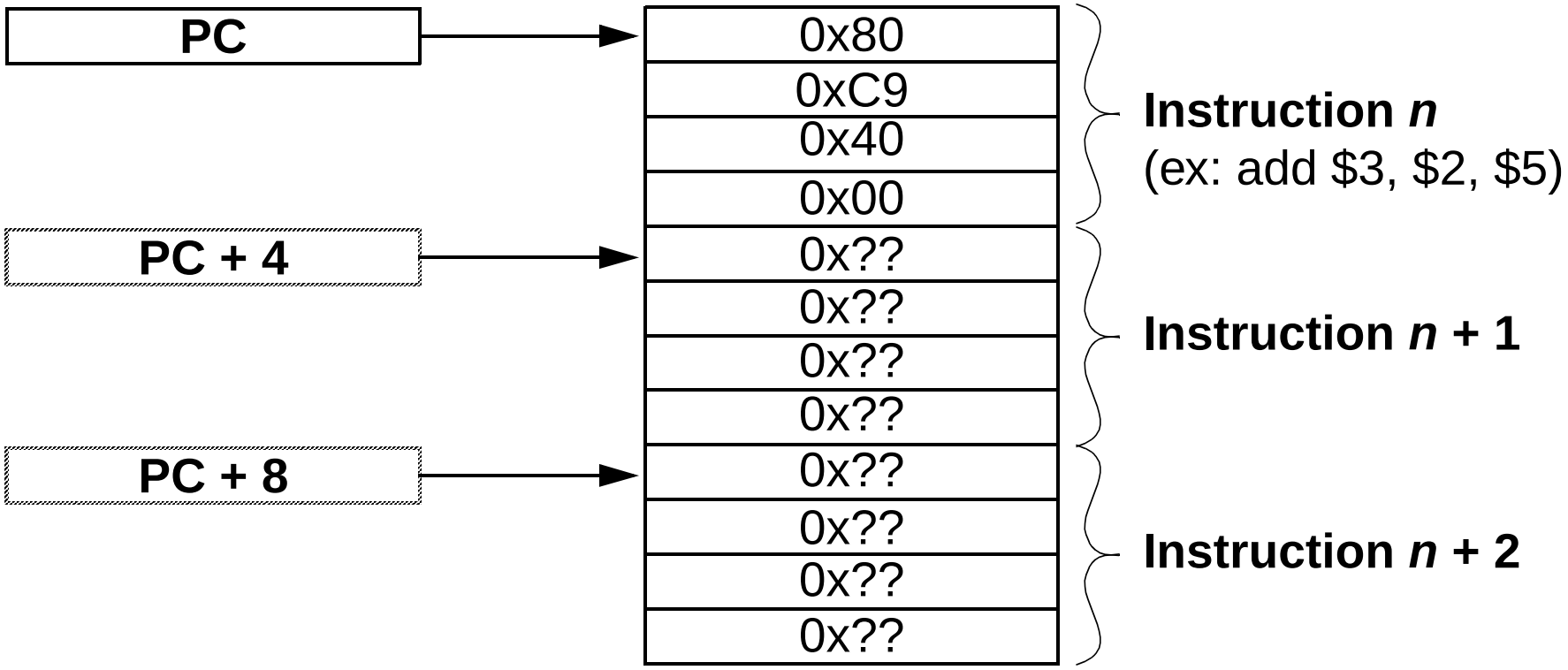
  - Would be useful to have a pointer to the next instruction.
  - We will call this the program counter (**PC**).

PROGRAM CONTROL

PC AND MEMORY MAP

•PROGRAM CONTROL
-INTRODUCTION
-PROGRAM COUNTER (PC)

- We can consider the program counter as pointing into memory at the next instruction to be executed.



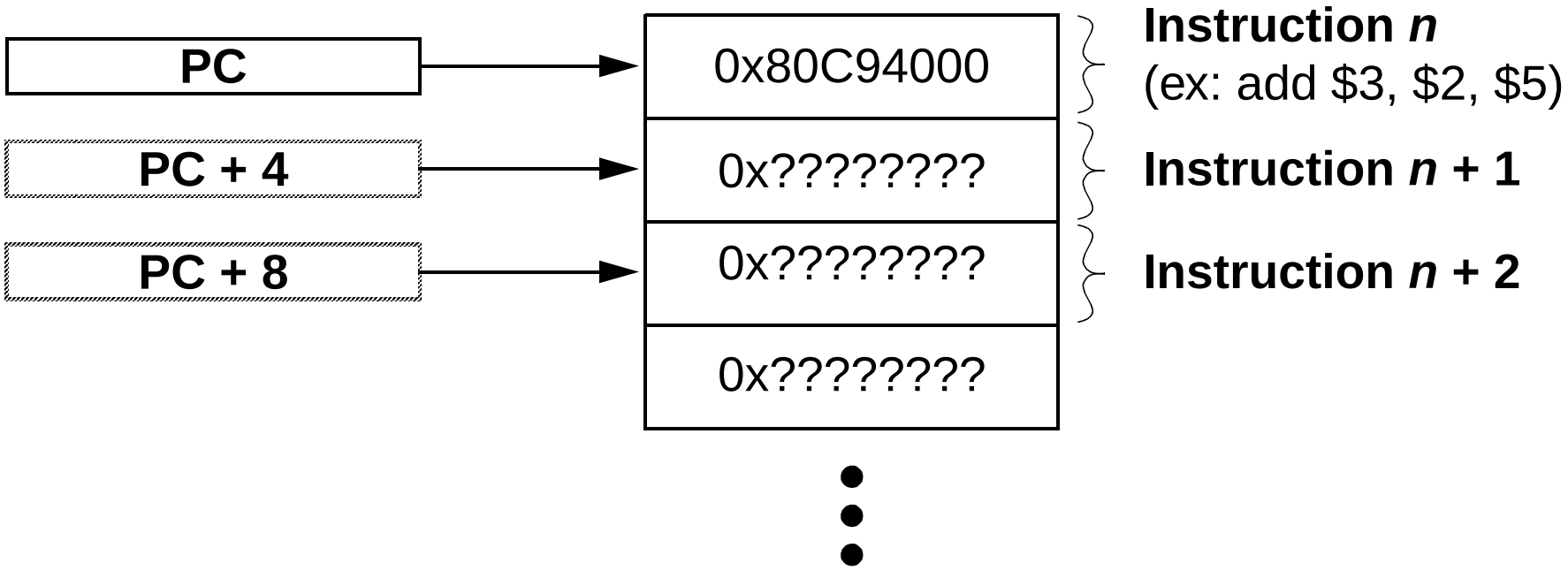
- Instructions are 32-bits (4 bytes), so add 1 to get next instruction.

PROGRAM CONTROL

PC AND MEMORY MAP

•PROGRAM CONTROL
-INTRODUCTION
-PROGRAM COUNTER (PC)
-PC AND MEMORY MAP

- To make the memory map representation a little more compact, we will make each address location 32-bits with the **PC** incremented by **4**..

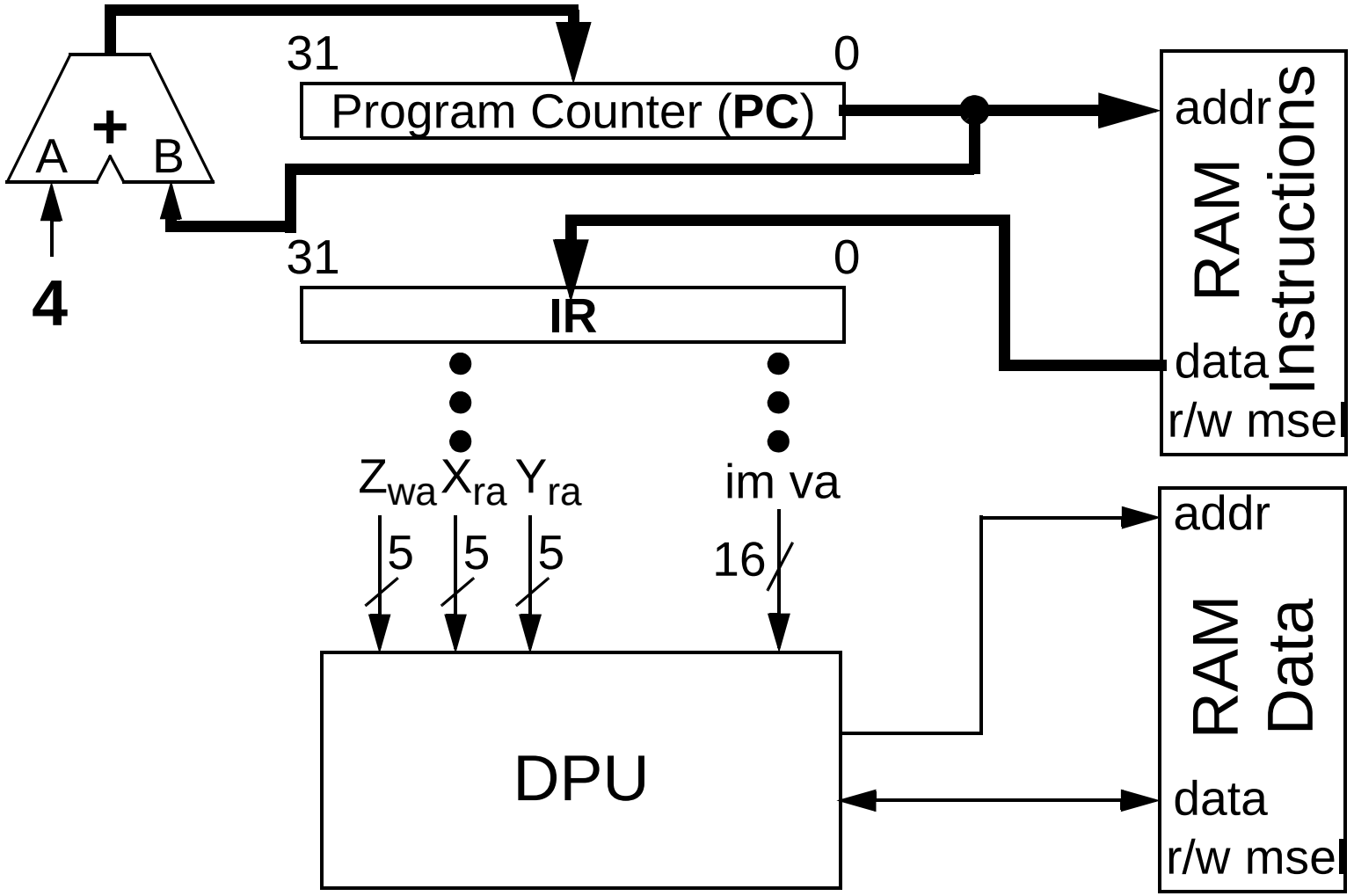


PROGRAM CONTROL

PC IN SINGLE CYCLE DPU

•PROGRAM CONTROL
-INTRODUCTION
-PROGRAM COUNTER (PC)
-PC AND MEMORY MAP

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



PROGRAM CONTROL

PC IN SINGLE CYCLE DPU

•PROGRAM CONTROL
-PROGRAM COUNTER (PC)
-PC AND MEMORY MAP
-PC IN 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 intruction 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**.

PROGRAM CONTROL

CHANGING PROGRAM FLOW

•PROGRAM CONTROL
-PROGRAM COUNTER (PC)
-PC AND MEMORY MAP
-PC IN SINGLE CYCLE DPU

- 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**

PROGRAM CONTROL

ABSOLUTE ADDRESSING

•PROGRAM CONTROL
-PC AND MEMORY MAP
-PC IN SINGLE CYCLE DPU
-CHANGING FLOW

- **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)**

PROGRAM CONTROL

RELATIVE ADDRESSING

- **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 + 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 + inst_size + inst_offset = PC + 4 + (word_offset << 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.

PROGRAM CONTROL

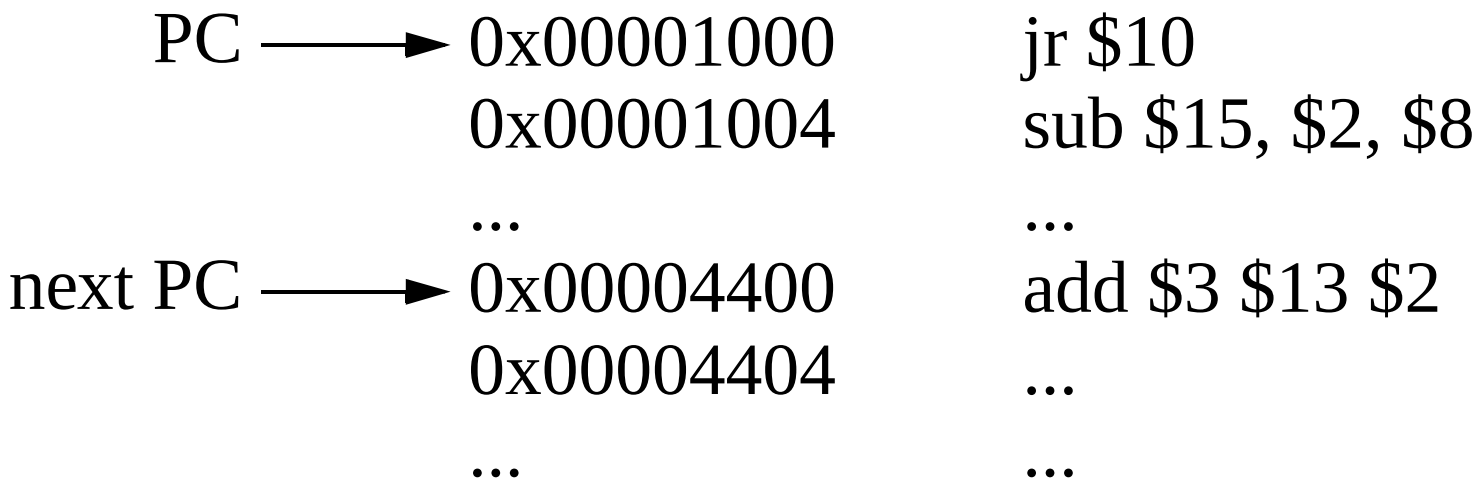
(UN)CONDITIONAL

- 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**.

JUMPING

JUMP W/ REGISTER (JR)

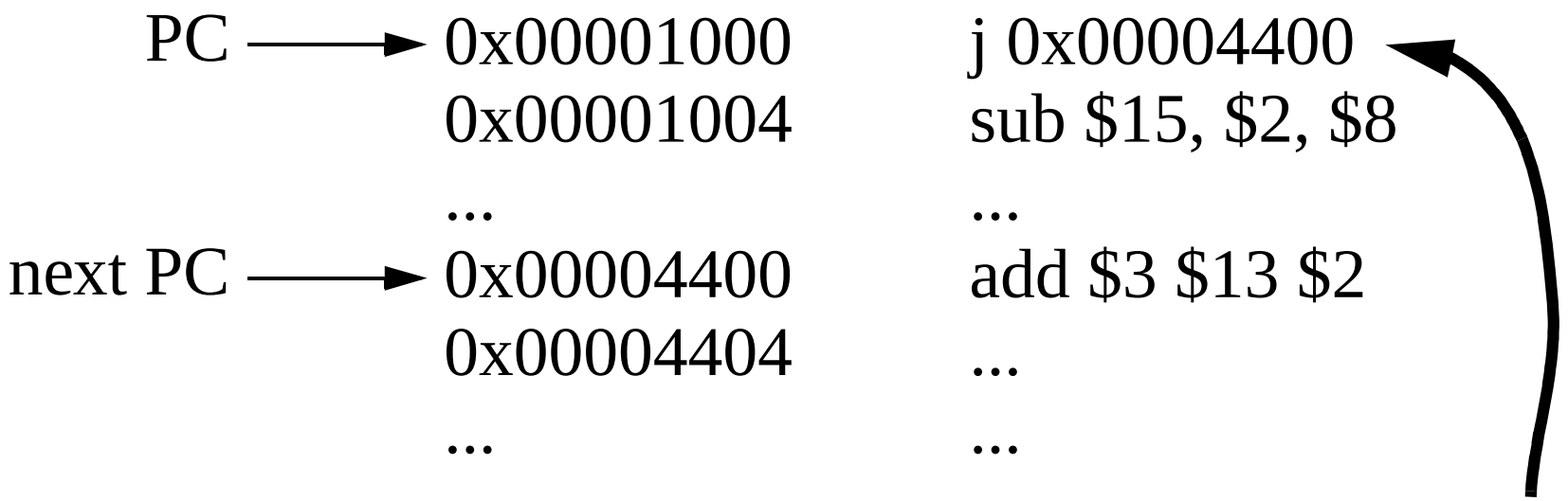
- The first form of program control is the **absolute jump** is as follows
  - **jr <register>**
  - The **jr** instruction changes **PC** to the value contained in the **register**.
  - For example, if **R10** contains **0x00004400** then after executing the following **jr** instruction, the next instruction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



JUMPING

JUMP W/ IMMEDIATE (J)

- 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**.



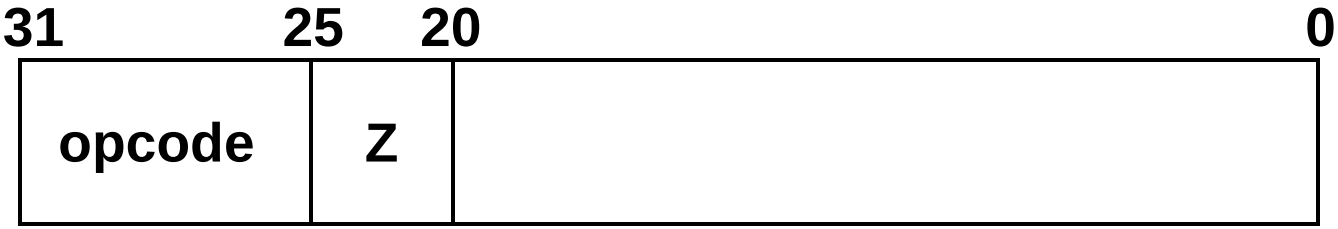
Note: assembler will convert to 0x0001100 so that 0x0001100<<2=0x00004400.

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 $\mathbf{Z_{wa}}$ to $\mathbf{Y_{ra}}$ and route **Y bus** to the **PC** so that the address in the register is loaded in the **PC**.

JR-format



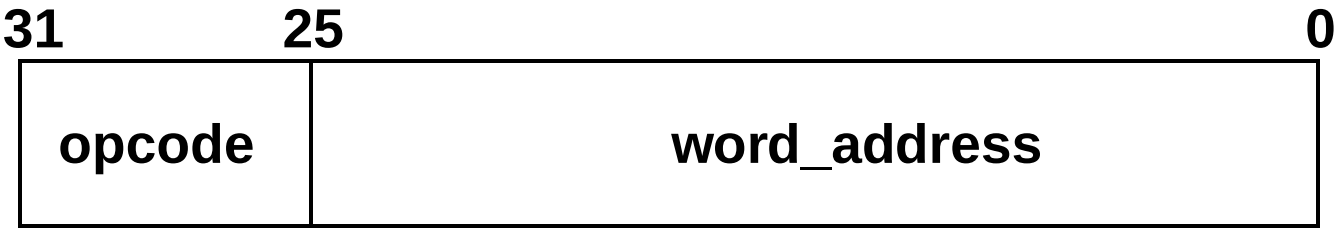
  - The jump can go anywhere in memory using the 32-bit register value.

JUMPING

J-FORMAT

- 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.

J-format



  - 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<<2)** leaving **PC[31:28]** unchanged. Therefore, cannot jump anywhere in memory, but almost.

JUMPING

ASSIGNED OPCODES

- As seen, the MIPS R3000/4000 has two basic forms of a jump or absolute instruction addressing.

Instruction	Assigned Opcode	Interpretation
j 0x00004028	**000010**	**The next instruction fetched is at address 0x00004028. Restriction: address is a 26-bit address to 4 byte words.**
jr $10	**000011**	**The next instruction fetched is at the 32-bit address stored in R10**

BRANCHING

BRANCHES AND CONDITIONS

- 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).

BRANCHING

BRANCH TYPES

- Below is a list of some possible branch types (many of these do not exist for the MIPS R3000/4000).

Common Mnemonics	Branch Type	Flags
beq	**Branch if equal**	**Z = 1**
bne or bnq	**Branch if not equal**	**Z = 0**
bpl	**Branch if positive**	**N = 0**
bmi	**Branch if negative**	**N = 1**
bcc	**Branch on carry clear**	**C = 0**
bcs	**Branch on carry set**	**C = 1**
bvc	**Branch on overflow clear**	**V = 0**
bvs	**Branch on overflow set**	**V = 1**

BRANCHING

BRANCH TYPES

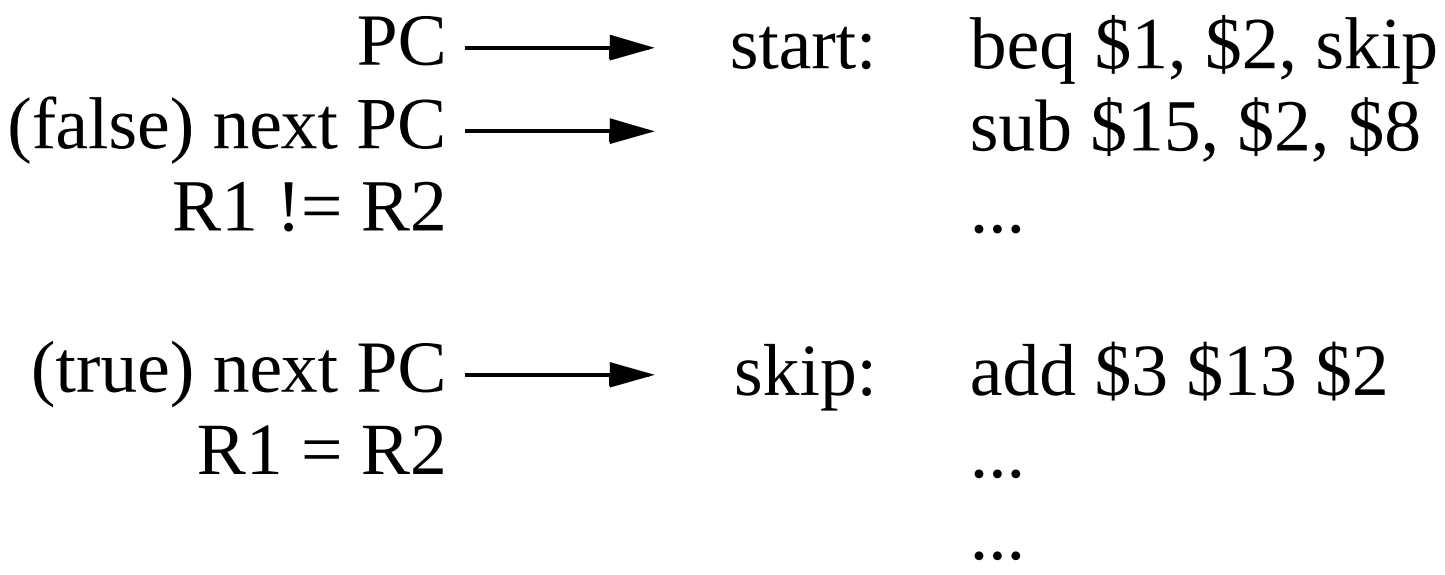
- continued...

Common Mnemonics	Branch Type	Flags
blt	**Branch on less than**	$N \oplus V$
ble	**Branch on less than or equal**	$Z + (N \oplus V)$
bge	**Branch on greater than or equal**	$\overline{N \oplus V}$
bgt	**Branch on greater**	$\overline{Z + (N \oplus V)}$
bra	**Branch always**	**No flags needed**
bsr	**Branch to subroutine**	**No flags needed**

BRANCHING

BRANCH IF EQUAL

- 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



- Notice that the branch is taken if **$1 = $2**.

BRANCHING

BRANCH IF NOT EQUAL

•BRANCHING
-BRANCHES AND COND.
-BRANCH TYPES
-BRANCH IF EQUAL

- 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**.

BRANCHING

ASSIGNED OPCODES

•BRANCHING
-BRANCH TYPES
-BRANCH IF EQUAL
-BRANCH IF NOT EQUAL

- As seen, the MIPS R3000/4000 has two basic forms of a jump or absolute instruction addressing.

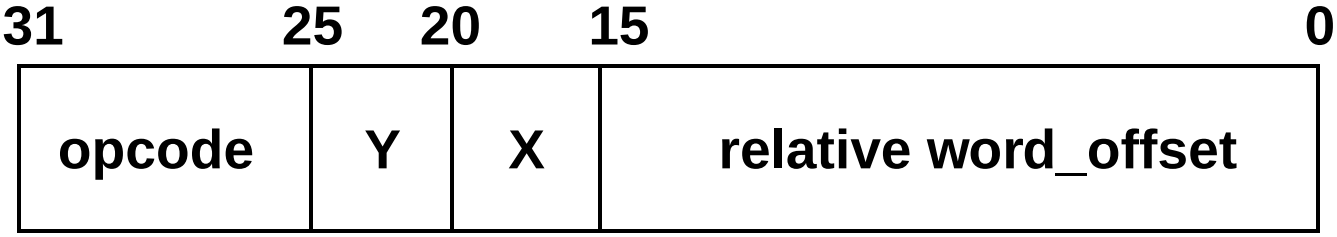
Instruction	Opcode	Interpretation
beq $10, $8, label	**000100**	**If contents of R10 is equal to contents of R8, next instruction that is fetched is the instruction labeled "label". Otherwise, the next instruction fetched is after the beq.**
bne $10, $8, label	**000101**	**If contents of R10 is not equal to contents of R8, next instruction that is fetched is the instruction labeled "label". Otherwise, the next instruction fetched is after the bne.**

BRANCHING

INSTRUCTION FORMAT

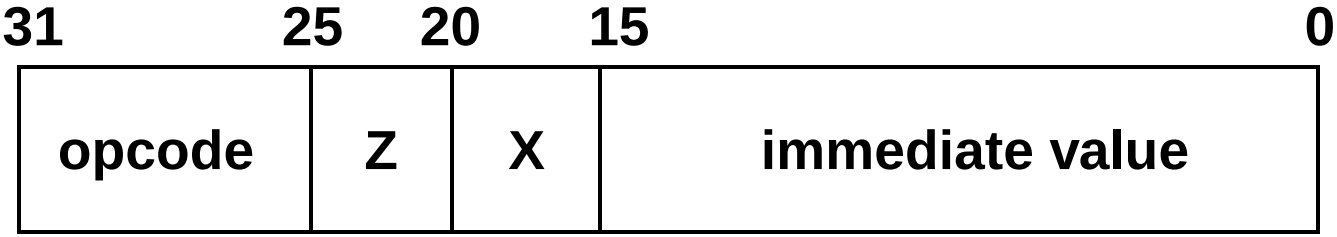
- Branching requires **two sources** for comparison and the **relative offset**.

B-format



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

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.

HIGHER LANGUAGES

LOOP ABSTRACTION

- Consider the following pseudo-code for a **loop**.

Pseudo-Code	MIPS Assembly	Register Transfer Notation
	add $26, $0, $0	R26 = 0
a = 0	add $14, $0, 0x05	R14 = 5
do	loop: ...	
...	...	
a = a + 1	add $26, $26, 0x01	R26 = R26 + 1
while (a != 5)	bne $26, $14, loop	PC = PC + 4 +word_offset<<2
...	...	

- Notice how a conditional branch is used for the while loop.

HIGHER LANGUAGES

IF-THEN-ELSE ABSTRACTION

- Consider the following pseudo-code for an **if-then-else** statement.

Pseudo-Code	MIPS Assembly (assume x in $5, y in $6)
if (x != y) then	beq $5, $6, else
...	...
...	...
else	j endif
...	else: ...
...	...
endif	endif: ...

- Notice use of **beq** for **if-then-else** and **j** at end of **if-then**.

HIGHER LANGUAGES

IF-THEN-ELSE ABSTRACTION

- Problem with previous slide is that we cannot relocate assembly code because of **j** instruction. Therefore, change assembly as follows.

Pseudo-Code	MIPS Assembly (assume x in $5, y in $6)
if (x != y) then	beq $5, $6, else
...	...
...	...
	beq $0, $0, endif
else	else: ...
...	...
...	...
endif	endif: ...

HIGHER LANGUAGES

SIMPLE EXAMPLE

- 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: ...
```

BRANCHES ON MIPS

GENERAL COMPARISONS

- 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**.

Instruction	Opcode	Interpretation
slt $10, $8, $9	**101010**	**If contents of $8 < contents of $9, then $10 = 0x01, else $10 = 0x00.**
slti $10,$8, 4	**001010**	**If contents of $8 < 4, then $10= 0x01, else $10 = 0x00.**

BRANCHES ON MIPS

SLT AND SLTI

•HIGHER LANGUAGES
•BRANCHES ON MIPS
-GENERAL COMPARISONS
-SLT AND SLTI

- How can **blt**, **ble**, **bgt**, and **bge** effectively be performed using **slt** and **slti**?

Desired Instruction	Meaning	Equivalent slt Condition	MIPS Instructions
blt $10, $8, loop	**Branch to loop if $10 < $8**	**Branch to loop if $10 < $8**	**slt $5, $10, $8 bne $5, $0, loop**
bge $10, $8, loop	**Branch to loop if $10 >= $8**	**Branch to loop if NOT ($10 < $8)**	**slt $5, $10, $8 beq $5, $0, loop**
bgt $10, $8, loop	**Branch to loop if $10 > $8**	**Branch to loop if $8 < $10**	**slt $5, $8, $10 bne $5, $0, loop**
ble $10, $8, loop	**Branch to loop if $10 <= $8**	**Branch to loop if NOT ($8 < $10)**	**slt $5, $8, $10 beq $5, $0, loop**

- Note: **$0** contains **0x00000000**.

BRANCHES ON MIPS

EXAMPLE

- Therefore, our previous example would actually be assembled as

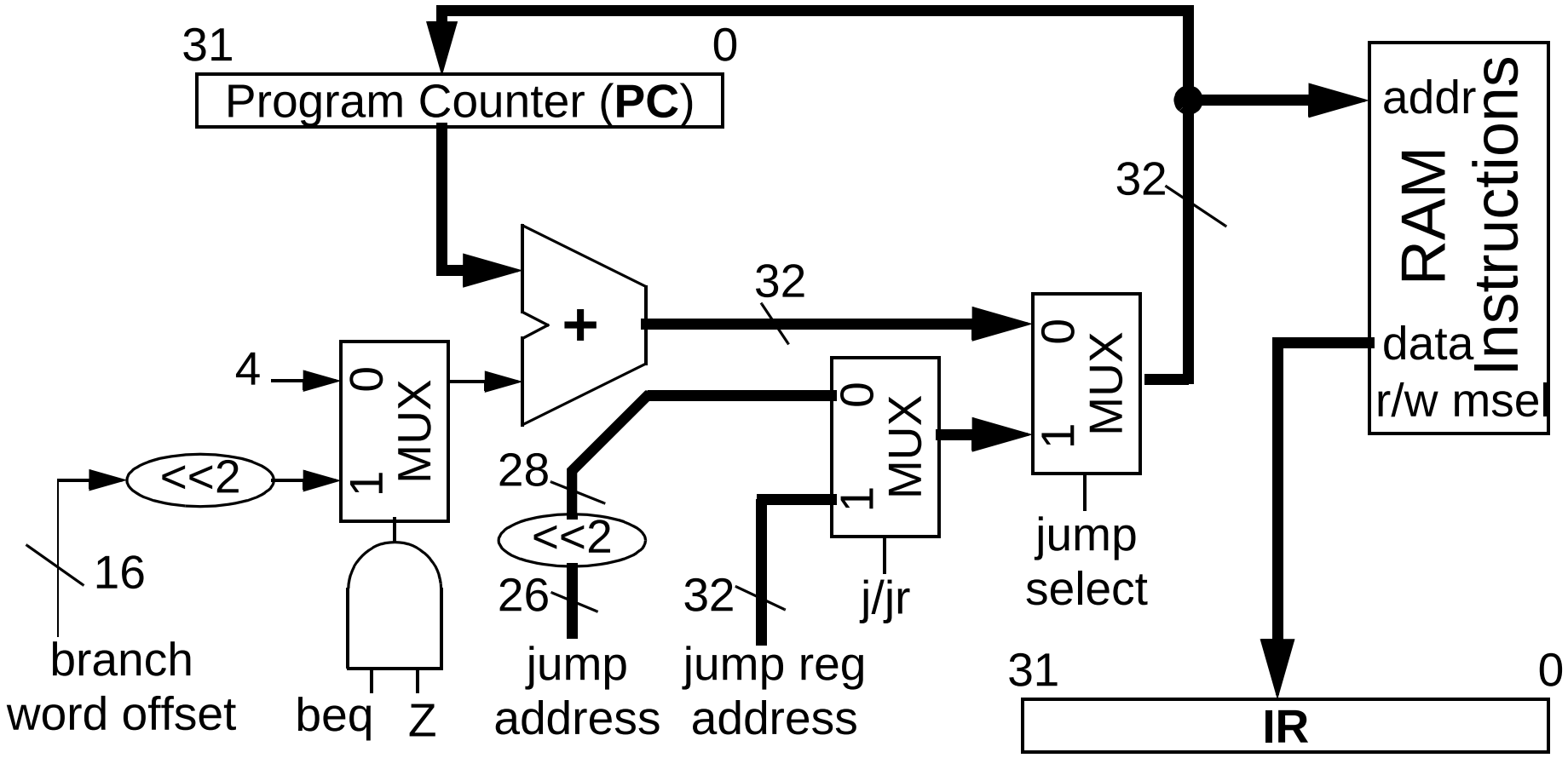
MIPS Assembly (almost)

```
        lwi $15, num
        bge $15, $0, endif0   <-- slt $5, $15, $0 ; beq $5, $0, endif0
        sub $15, $0, $15
        swi $15, num
endif0: lwi $15, temperature
        blt $15, 0x0019, else25   <-- slti $5, $15, 0x0019 ; bne $5, $0, else25
swim:   ...
        j endif25
else25: ...
cycle:  ...
endif25: ...
```

SINGLE CYCLE DPU

PC UPDATE

- Of course, modifications are needed to the DPU* to allow updating the program counter appropriately with these branches and jumps.

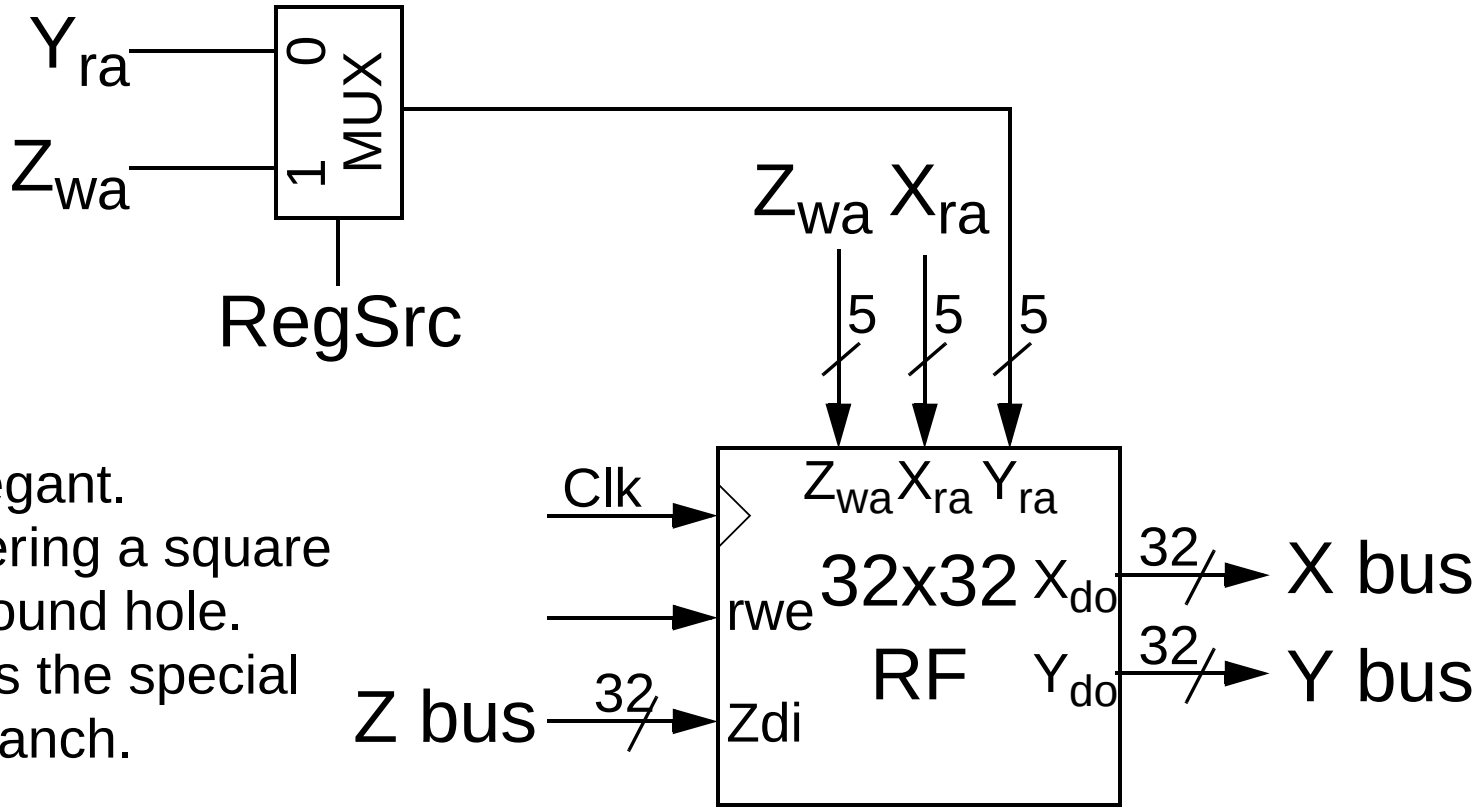


* Note: Not quite accurate for the MIPS architecture.

SINGLE CYCLE DPU

MODIFICATIONS TO DPU

- 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.

TARGET CALC.

JUMP TARGET CALCULATION

- To calculate jump target consider the following instruction
  - **j 0x00400040**
- The encoding of the jump would be

J-format

31 25	0
opcode	**instruction word address**
0000 10	**00 0001 0000 0000 0000 0001 0000**

which gives an instruction encoding of **0x08100010** and not 0x08400040.

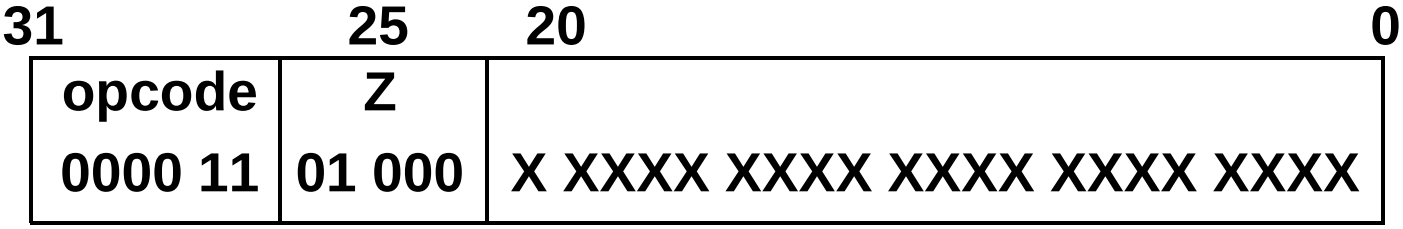
- Why? Because we need to encode with word addresses such that
  - **0x00100010 << 2 = 0x00400040**
  - This gives the preferred 28-bits over 26-bits.
- Hence, **PC[27:0] = (word_address << 2)**

TARGET CALC.

JUMP TARGET CALCULATION

- 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



- This gives an instruction encoding of **0x0D000000** (for **X=0**).

TARGET CALC.

BRANCH TARGET CALC.

- 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**

TARGET CALC.

BRANCH TARGET CALC.

- Using the word offset calculated on the previous slide of we can verify that that this is the correct word offset since

PC = PC + 4 + (word offset << 2)
= 0x00001000 + 0x04 + (0x0CFF << 2)
= 0x00001000 + 0x04 + 0x000033FC = 0x00004400

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

I-format

31 25	20 15		0
opcode	**Z**	**X**	**word offset**
0001 00	**00 001**	**0 0010**	**0000 1100 1111 1111**

- This gives an instruction encoding of **0x10220CFF**.