CHAPTER XIV

PROGRAM CONTROL, JUMPING, AND BRANCHING

READ BRANCHING FREE-DOC ON COURSE WEBPAGE
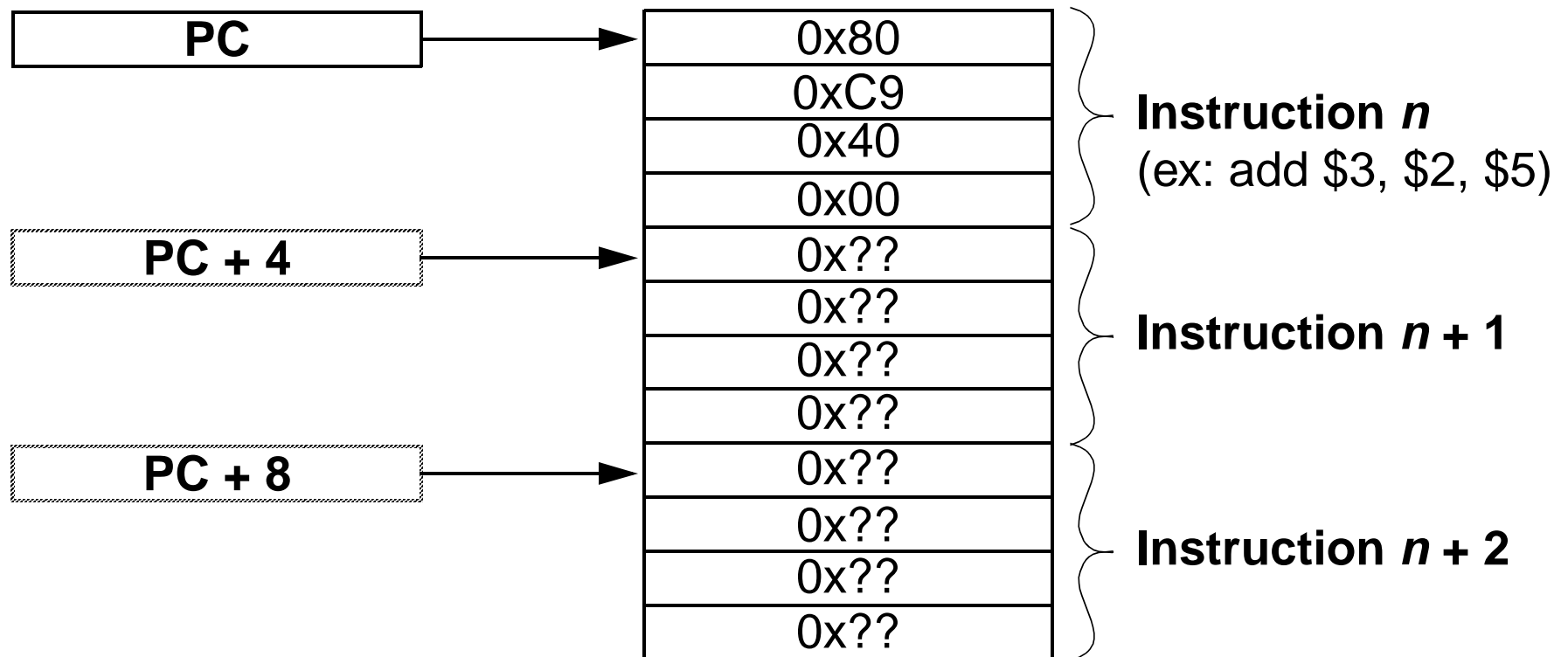
PROGRAM CONTROL
INTRODUCTION

- 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?

INTRO. TO COMP. ENG.
CHAPTER XIV-3

PROGRAM CONTROL

PROGRAM CONTROL

PROGRAM COUNTER (PC)

•PROGRAM CONTROL
-INTRODUCTION

- **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

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

PC	→	0x80	⎫
		0xC9	**Instruction _n_**
		0x40	(ex: add $3, $2, $5)
		0x00	⎭
PC + 4	→	0x??	⎫
		0x??	**Instruction _n_ + 1**
		0x??	
		0x??	⎭
PC + 8	→	0x??	⎫
		0x??	**Instruction _n_ + 2**
		0x??	
		0x??	⎭

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

INTRO. TO COMP. ENG.
CHAPTER XIV-5

PROGRAM CONTROL

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**..

PC	→	0x80C94000	Instruction *n* (ex: add $3, $2, $5)
PC + 4	→	0x????????	Instruction *n* + 1
PC + 8	→	0x????????	Instruction *n* + 2
		0x????????	

INTRO. TO COMP. ENG.
CHAPTER XIV-6

PROGRAM CONTROL

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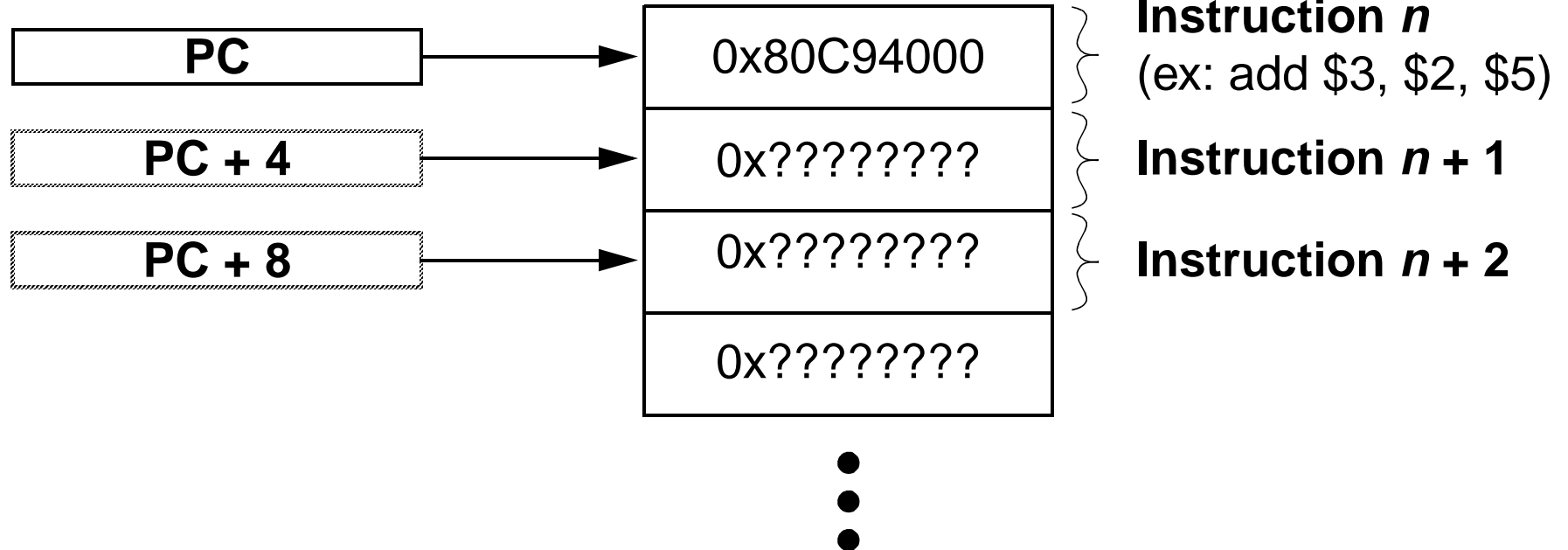
PROGRAM CONTROL

PROGRAM CONTROL
PC IN SINGLE CYCLE DPU

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

INTRO. TO COMP. ENG.
CHAPTER XIV-7

PROGRAM CONTROL

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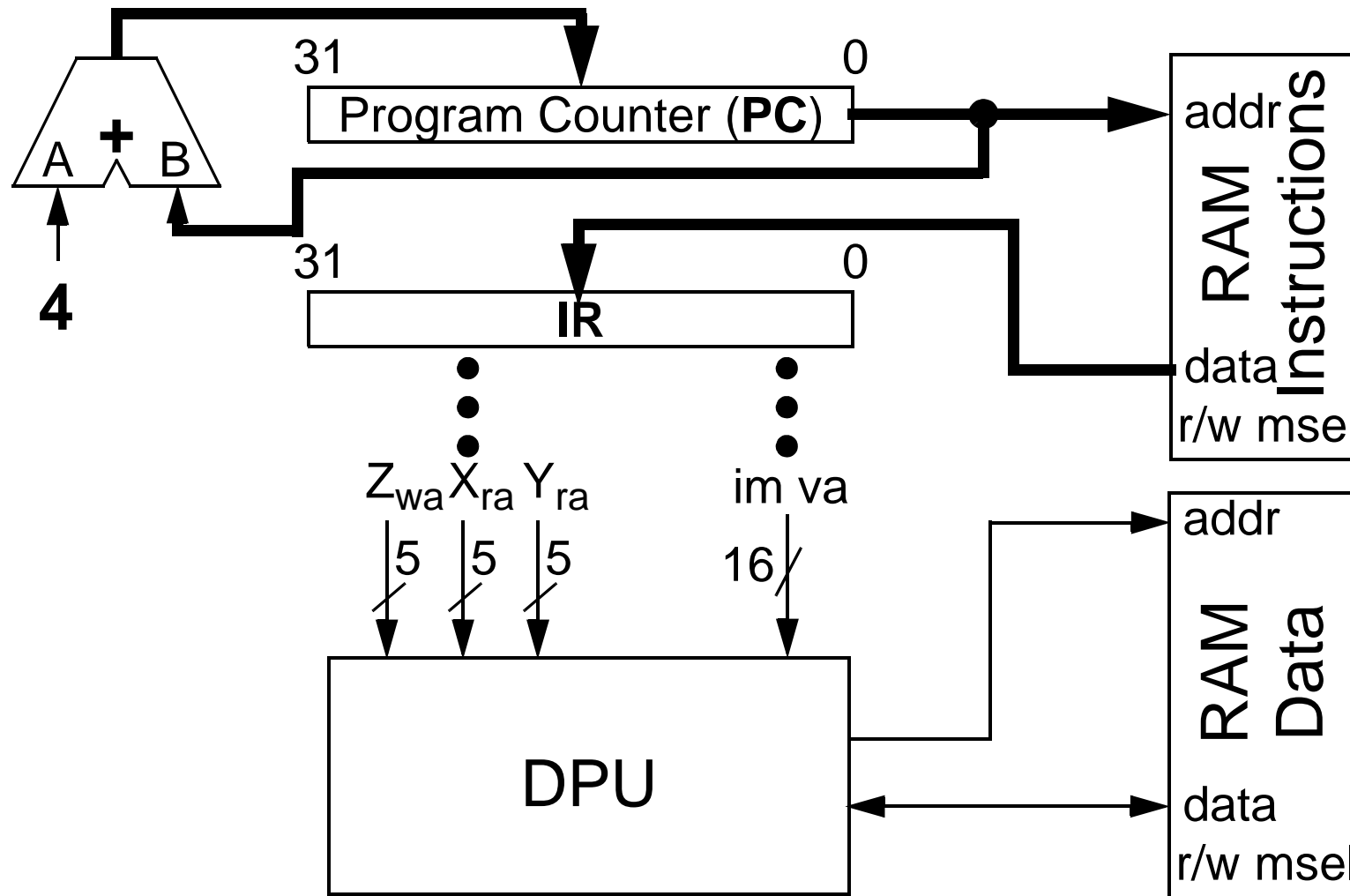
•PROGRAM CONTROL
  -PROGRAM COUNTER (PC)
  -PC AND MEMORY MAP
  -PC IN SINGLE CYCLE DPU

- **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

- 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**

INTRO. TO COMP. ENG.
CHAPTER XIV-9

PROGRAM CONTROL

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.

INTRO. TO COMP. ENG.
CHAPTER XIV-11

PROGRAM CONTROL

•PROGRAM CONTROL
-CHANGING FLOW
-ABSOLUTE ADDRESSING
-RELATIVE ADDRESSING

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**.

INTRO. TO COMP. ENG.
CHAPTER XIV-12

PROGRAM CONTROL

JUMPING

JUMP W/ REGISTER (JR)

•PROGRAM CONTROL
  -ABSOLUTE ADDRESSING
  -RELATIVE ADDRESSING
  -(UN)CONDITIONAL

- 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**.
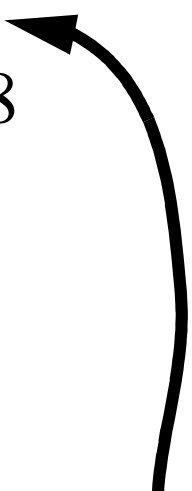
$$
\begin{array}{lll}
\text{PC} \longrightarrow & \text{0x00001000} & \text{jr \$10} \\
& \text{0x00001004} & \text{sub \$15, \$2, \$8} \\
& \dots & \dots \\
\text{next PC} \longrightarrow & \text{0x00004400} & \text{add \$3 \$13 \$2} \\
& \text{0x00004404} & \dots \\
& \dots & \dots
\end{array}
$$

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**.

$$
\begin{array}{llll}
\text{PC} \longrightarrow & \text{0x00001000} & \text{j 0x00004400} \\
& \text{0x00001004} & \text{sub \$15, \$2, \$8} \\
& \ldots & \ldots \\
\text{next PC} \longrightarrow & \text{0x00004400} & \text{add \$3 \$13 \$2} \\
& \text{0x00004404} & \ldots \\
& \ldots & \ldots
\end{array}
$$

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

INTRO. TO COMP. ENG.
CHAPTER XIV-14

PROGRAM CONTROL

JUMPING
JR-FORMAT

•PROGRAM CONTROL
•JUMPING
-JUMP W/ REGISTER (JR)
-JUMP W/ IMMEDIATE (J)

- 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**.

31   25   20   0
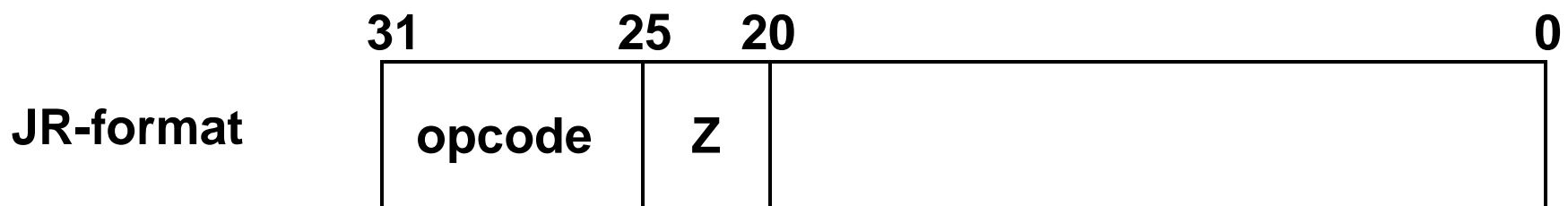
JR-format
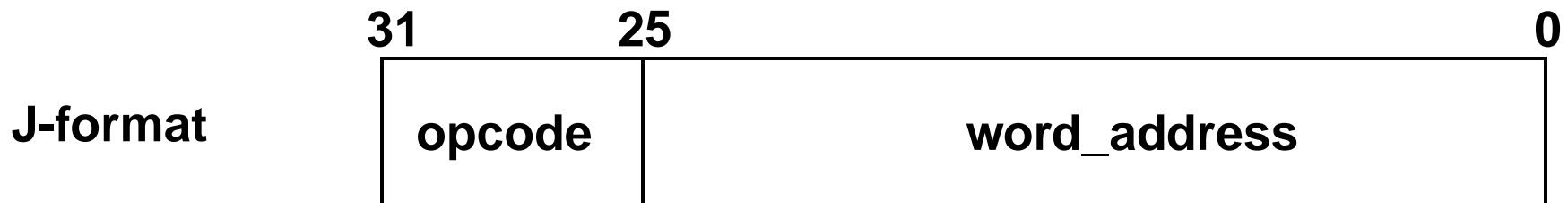
opcode	Z	

  - 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

31	25	0
opcode	word_address	

- 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

•JUMPING
•BRANCHING
   -BRANCHES AND COND.
   -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

•JUMPING
•BRANCHING
  -BRANCHES AND COND.
  -BRANCH TYPES

- 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

$$PC \longrightarrow \quad \text{start:} \quad \text{beq \$1, \$2, skip}$$

$$\text{(false) next PC} \longrightarrow \quad \text{sub \$15, \$2, \$8}$$

$$\text{R1} \neq \text{R2} \qquad \qquad \text{...}$$

$$\text{(true) next PC} \longrightarrow \quad \text{skip:} \quad \text{add \$3 \$13 \$2}$$

$$\text{R1} = \text{R2} \qquad \qquad \text{...}$$

$$\qquad \qquad \qquad \qquad \text{...}$$

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

BRANCHING

BRANCH IF NOT 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

$$
\begin{array}{lll}
\text{PC} \longrightarrow & \text{start:} & \text{bne \$1, \$2, skip} \\
\text{(false) next PC} \longrightarrow & & \text{sub \$15, \$2, \$8} \\
\text{R1 = R2} & & \text{...} \\
\\
\text{(true) next PC} \longrightarrow & \text{skip:} & \text{add \$3 \$13 \$2} \\
\text{R1 != R2} & & \text{...} \\
& & \text{...}
\end{array}
$$

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

BRANCHING

ASSIGNED OPCODES

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

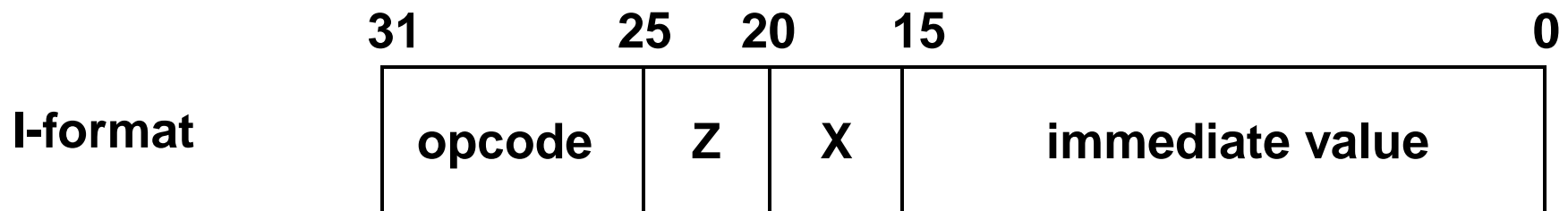| Instruction	Opcode	Interpretation
beq $10, $8, label	**000100**	**If contents of R10 is equal to contents of R8, next instruction that is fetched is the instruction labeled "label". Otherwise, the next instruction fetched is after the beq.**
bne $10, $8, label	**000101**	**If contents of R10 is not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

| 31 | 25 | 20 | 15 | 0 |

B-format

| opcode | Y | X | relative word_offset |
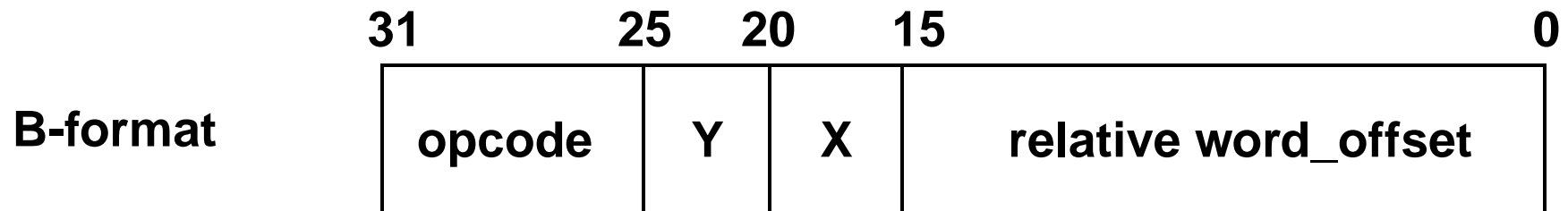
- This **B-format** is effectively the same as the **I-format**.

| 31 | 25 | 20 | 15 | 0 |

I-format

| opcode | Z | X | immediate value |

- 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.

INTRO. TO COMP. ENG.
CHAPTER XIV-25

PROGRAM CONTROL

HIGHER LANGUAGES
IF-THEN-ELSE ABSTRACTION

•BRANCHING
•HIGHER LANGUAGES
  -LOOP 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**.

INTRO. TO COMP. ENG.
CHAPTER XIV-26

PROGRAM CONTROL

HIGHER LANGUAGES
IF-THEN-ELSE ABSTRACTION

•BRANCHING
•HIGHER LANGUAGES
  -LOOP ABSTRACTION
  -IF-THEN-ELSE ABSTRACT.

- 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

...

...

...

...

else

beq $0, $0, endif

...

else: ...

...

...

endif

endif: ...

HIGHER LANGUAGES

SIMPLE EXAMPLE

- Another example is given below. Note: $0 contains 0x00000000.

Pseudo-Code	MIPS Assembly (almost)
	lwi $15, num
if (num<0) then	bge $15, $0, endif0
num = -num	sub $15, $0, $15
	swi $15, num
end	endif0: lwi $15, temperature
if (temperature>=25) then	blt $15, 0x0019, else25
activity = "swim"	swim: ...
	j endif25
else	else25: ...
activity = "cycle"	cycle: ...
endif	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

- 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**.

INTRO. TO COMP. ENG.
CHAPTER XIV-30

PROGRAM CONTROL

BRANCHES ON MIPS
EXAMPLE

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

- 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
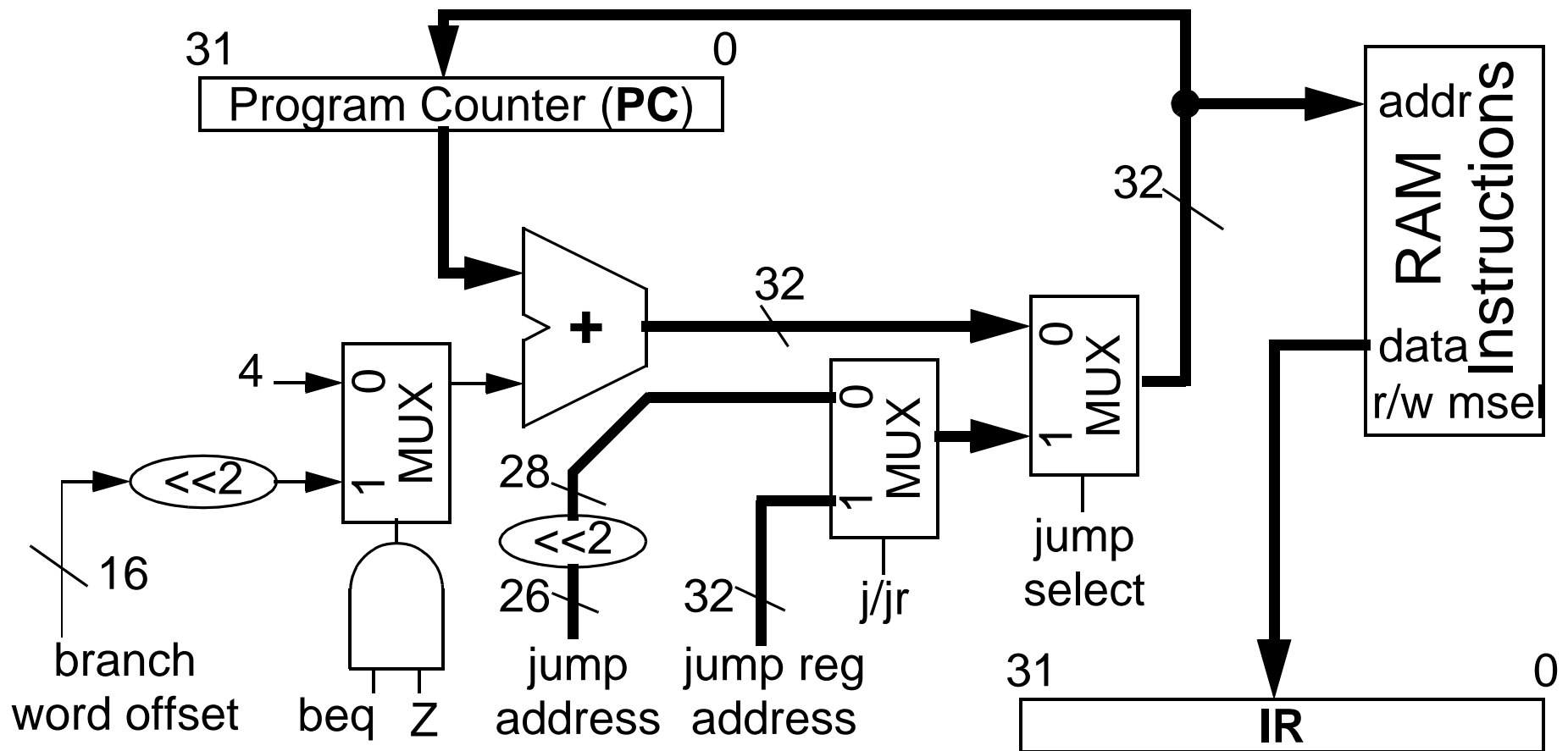
swim: ...

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.
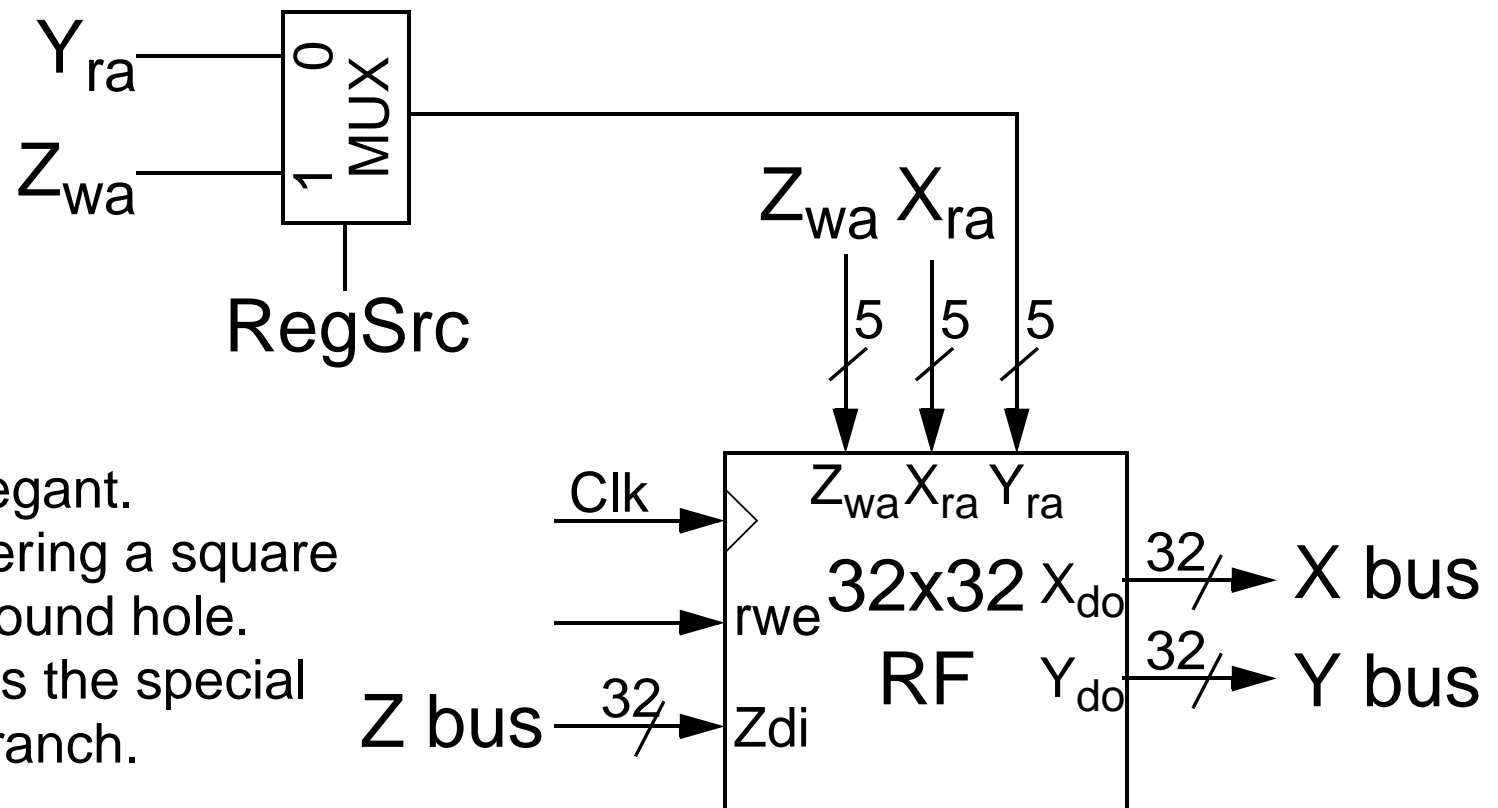
* Note: Not quite accurate for the MIPS architecture.

INTRO. TO COMP. ENG.
CHAPTER XIV-32

PROGRAM CONTROL

SINGLE CYCLE DPU
MODIFICATIONS TO DPU

•BRANCHES ON MIPS
•SINGLE CYCLE DPU
  -PC UPDATE

- Note that branch instructions have **two sources** and an **immediate value**.

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

Y_{ra}

Z_{wa}

MUX 1 0

RegSrc

Z_{wa} X_{ra}

5   5   5

Not very elegant.
Like hammering a square
peg into a round hole.
But, it solves the special
case of a branch.

Clk

Z_{wa} X_{ra} Y_{ra}

32x32 X_{do} $\xrightarrow{32}$ X bus

rwe

RF Y_{do} $\xrightarrow{32}$ Y bus

Z bus $\xrightarrow{32}$ Zdi

TARGET CALC.

JUMP TARGET CALCULATION

- To calculate jump target consider the following instruction

  - **j 0x00400040**

- The encoding of the jump would be

	31 25	0
J-format	**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)**

INTRO. TO COMP. ENG.
CHAPTER XIV-34

PROGRAM CONTROL

TARGET CALC.

JUMP TARGET CALCULATION

•BRANCHES ON MIPS
•SINGLE CYCLE DPU
•TARGET CALCULATION
  -JUMP TARGET CALC.

- 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

	31	25	20	0
JR-format	**opcode**	**Z**		
	0000 11	**01 000**	**X XXXX XXXX XXXX XXXX XXXX**	

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

TARGET CALC.

BRANCH TARGET CALC.

- For branch target calculation, consider the following code fragment.

$$0x00001000 \qquad \text{beq } \$1, \$2, \text{skip}$$
$$0x00001004 \qquad \text{sub } \$15, \$2, \$8$$
$$0x00001008 \qquad ...$$
$$...$$
$$0x00004400 \quad \text{skip:} \quad \text{add } \$3 \ \$13 \ \$2$$

- 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

$$PC = PC + 4 + (\text{word offset} << 2)$$
$$= 0x00001000 + 0x04 + (0x0CFF << 2)$$
$$= 0x00001000 + 0x04 + 0x000033FC = 0x00004400$$

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

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

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