LE15.2

Bookmark this page

Calculator

For all Beta related questions, you should make use of the Beta documentation, the Beta Instruction Summary, the Unpipelined Beta Diagram and the Pipelined Beta Diagram.

LE15.2.1: Pipelined Execution

1/1 point (ungraded)

You are exploring several different Beta implementations, using a benchmark program that includes the simple loop shown below. Note that the code contains a number of **NOP()** instructions, defined as **ADD(R31,R31,R31)**.

```
        ...
        NOP() NOP() NOP() NOP()
Loop:
        LD(R0, 0, R0)
AA:
        MUL(R0, R1, R4)
BB:
        BNE(R0, LOOP)
CC:
        ADD(R0, R3, R3)
        NOP() NOP() NOP() NOP()
        ...
```

Your first experiment is to run the code on an unpipelined Beta, like the one shown in Lecture (and studied in the lab assignment).

You observe that the program runs through several iterations of the loop, before dropping out and executing the **ADD** and subsequent instructions.

1. On an **unpipelined** Beta, how many clock cycles of execution time are required **for each iteration** through the loop?

   Unpipelined Beta, clocks per loop iteration:

   3 ✓ Answer: 3

   Explanation
   On an unpipelined beta the number of clock cycles required for each iteration of the loop is equal to the number of instructions in the loop which is 3.

   Next, you run the code on a **fully functional 5-stage Beta pipeline** (with working bypass, annul, and stall logic) and count the cycles per loop iteration. Note that the code scrap begins and ends with sequences of **NOPs**; thus you don't need to worry about pipeline hazards involving interactions with instructions outside of the region shown.

2. How many clock cycles of execution time are required by the fully functional 5-stage pipelined Beta **for each iteration** through the loop?

   Functional 5-stage pipelined Beta, clocks per loop iteration:

   6 ✓ Answer: 6

   Explanation
   On an fully functional 5-stage pipelined Beta the number of clock cycles required for each iteration of the loop is equal to 6 as shown in the pipeline diagram below.

Cycle #	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
IF	LD	MUL	BNE	BNE	BNE	ADD	LD	MUL	BNE	BNE
RF	NOP	LD	MUL	MUL	MUL	BNE	NOP	LD	MUL	MUL
ALU	BNE	NOP	LD	NOP	NOP	MUL	BNE	NOP	LD	NOP
MEM	MUL	BNE	NOP	LD	NOP	NOP	MUL	BNE	NOP	LD
WB	NOP	MUL	BNE	NOP	LD	NOP	NOP	MUL	BNE	NOP

The first iteration of the loop begins at cycle 1 and the second iteration begins at cycle 7 and procee

Calculator

without further stalls. So the number of clocks per iteration is 6.

Finally, you try to run the code on a **defective** version of the 5-stage pipelined Beta: **it has no bypass paths, annulment of instructions in branch delay slots, or pipeline stalls**. You undertake to convert the above code to run on the defective processor, by adding the **minimum** number of **NOP** instructions at the various tagged points in this code to make it give the same results on your defective pipelined Beta as it gives on a normal, unpipelined Beta.

3. Specify the minimal number of **NOP** instructions (defined as **ADD(R31,R31,R31)**) to be added at each of the labeled points in the above program to make it work properly on the defective 5-stage Beta pipeline.

NOPs at Loop: 0 ✓ Answer: 0

NOPs at AA: 3 ✓ Answer: 3

NOPs at BB: 0 ✓ Answer: 0

NOPs at CC: 1 ✓ Answer: 1

Explanation
In order to get the loop to work as desired in the defective beta, then NOPs must be inserted to deal with all the hazards, resulting in the following pipeline diagram:

Cycle #	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
IF	LD	NOP	NOP	NOP	MUL	BNE	NOP	LD
RF	NOP	LD	NOP	NOP	NOP	MUL	BNE	NOP
ALU	BNE	NOP	LD	NOP	NOP	NOP	MUL	BNE
MEM	MUL	BNE	NOP	LD	NOP	NOP	NOP	MUL
WB	NOP	MUL	BNE	NOP	LD	NOP	NOP	NOP

The diagram shows that in order for the MUL to read the correct value of R0, 3 NOPS must be inserted between the LD and MUL to allow the LD enough time to write its result back to the register file. The BNE then reads the same R0 from the register file with no additional stalls. In order to avoid fetching the ADD instruction since there is no branch annulment, an additional NOP must be inserted after the BNE instruction to fill the branch delay slot, and then the LD can once again be fetched in order to repeat the loop.

Submit

Answers are displayed within the problem

LE15.2.2: Branch delay slots

0.5/1 point (ungraded)
This is a very tricky, classic pipelining problem.

The following instruction sequence is executed on a 5-stage pipelined Beta with bypass paths **but no branch delay slot annulment**, i.e., the instruction following a branch is always executed whether the branch is taken or not.

```
      CMOVE(5, R0)          | Load 5 into R0
      CMOVE(7, R1)          | Load 7 into R1

      BR(X)
      BR(Y)

Y:    ADDC(R0, 1, R0)       | Increment R0 contents by 1

X:    SUBC(R1, 1, R1)       | Decrement R1 contents by 1
```

Calculator

```
        NOP()                 | wait, to stabilize registers
        NOP()
        NOP()
        HALT()
```

Your challenge is to figure out the final values in R0 and R1.

Final R0 Value: 6 Answer: 6

Final R1 Value: 5 ✓ Answer: 5

Explanation
The pipeline diagram for this problem looks as follows:

Cycle #	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
IF	CMOVE	CMOVE	BR(X)	BR(Y)	SUBC	ADDC	SUBC				
RF		CMOVE	CMOVE	BR(X)	BR(Y)	SUBC	ADDC	SUBC			
ALU			CMOVE	CMOVE	BR(X)	BR(Y)	SUBC	ADDC	SUBC		
MEM				CMOVE	CMOVE	BR(X)	BR(Y)	SUBC	ADDC	SUBC	
WB					CMOVE	CMOVE	BR(X)	BR(Y)	SUBC	ADDC	SUBC

Since this beta does not have branch annulment, the BR(Y) instruction that is fetched immediately after BR(X), does not get annuled. The result of this is that once the BR(X) gets into the RF stage, it determines that its branch is taken so the IF of cycle 5 should fetch the SUBC. However, when the BR(Y) gets into the RF stage, it determines that its branch is taken so the IF of cycle 6 should fetch the ADDC. After that, the SUBC is reexecuted since it follows the ADDC instruction. The result of executing two SUBC operations and one ADDC operation is that R0 ends up incremented once to 6, and R1 ends up decremented twice to end up with a 5. Note that if branch annulment had been in place, then only one SUBC instruction would get executed and no ADDC instructions would get executed so we would get R0 = 5 and R1 = 6 if things were working correctly.

Submit

Discussion

Hide Discussion

Topic: 15. Pipelining the Beta / LE15.2

Add a Post

All Posts

Is BNE an instruction?

discussion posted 6 years ago by **Ucheinyama**

Hi, is BNE an instruction, because the answer to unpipelined beta, seems to rule out BNE?

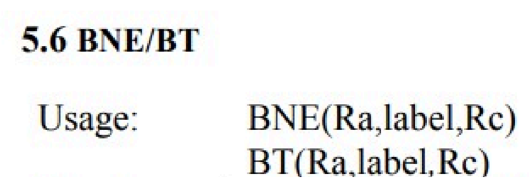
This post is visible to everyone.

Add a Response

1 response

rhodesd (Community TA)
6 years ago

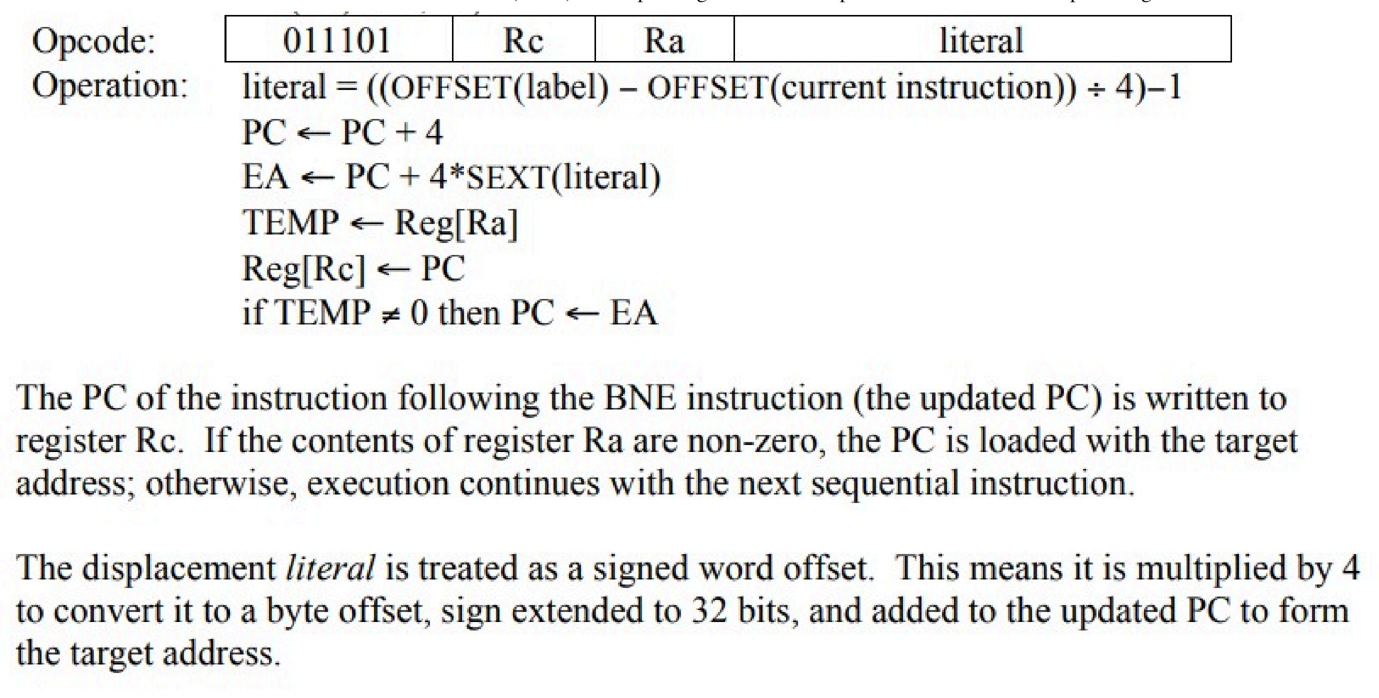
Hi Ucheinyama, the BNE instruction can be found on page 4 of the Beta Docs, but I'll just include it here.



5.6 BNE/BT

Usage:	BNE(Ra,label,Rc)
	BT(Ra,label,Rc)

Calculator



Opcode:

011101	Rc	Ra	literal

Operation:

```
literal = ((OFFSET(label) – OFFSET(current instruction)) ÷ 4)–1
PC ← PC + 4
EA ← PC + 4*SEXT(literal)
TEMP ← Reg[Ra]
Reg[Rc] ← PC
if TEMP ≠ 0 then PC ← EA
```

The PC of the instruction following the BNE instruction (the updated PC) is written to register Rc. If the contents of register Ra are non-zero, the PC is loaded with the target address; otherwise, execution continues with the next sequential instruction.

The displacement *literal* is treated as a signed word offset. This means it is multiplied by 4 to convert it to a byte offset, sign extended to 32 bits, and added to the updated PC to form the target address.

Maybe you are considering the number of instructions executed if the branch is *not* taken, that is LD, then MUL, then BNE, then ADD? The question asks for the number of cycles for each **loop iteration**. Is that it? If not, then why does the answer seem to rule out BNE?

Add a comment

Showing all responses

Add a response:

Preview

Submit

Previous

Next

Calculator

Accessibility Policy
Trademark Policy
Sitemap
Cookie Policy
Your Privacy Choices

Connect

Idea Hub
Contact Us
Help Center
Security
Media Kit

GET IT ON Google play
Download on the App Store

Calculator