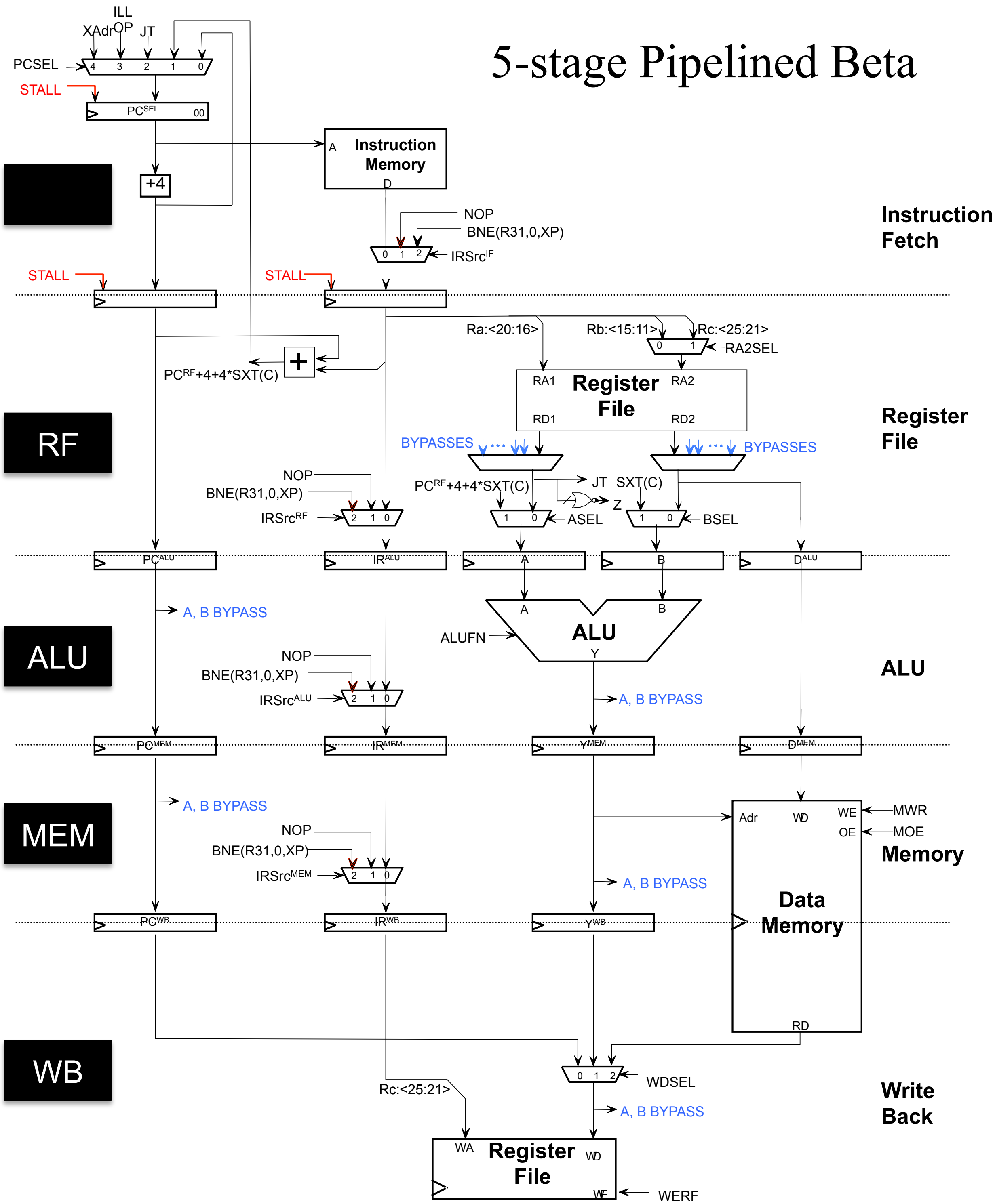
Computation Structures

Pipelining the Beta Worksheet

Options for dealing with *data* and *control* hazards: **stall, bypass, speculate**



Problem 1.

The program shown on the right is executed on a 5-stage pipelined Beta with full bypassing and annulment of instructions following taken branches.

The program has been running for a while and execution is halted at the end of cycle 108.

The pipeline diagram shown below shows the history of execution at the time the program was halted.

```
  . = 0
outer_loop:
  CMOVE(16,R0)   // initialize loop index J
  CMOVE(0,R1)

loop:              // add up  elements in array
  SUBC(R0,1,R0)  // decrement index
  MULC(R0,4,R2)  // convert to byte offset
  LD(R2,0x310,R3)// load value from A[J]
  ADD(R3,R1,R1)  //  add to sum
  BNE(R0,loop)   // loop until all words are summed

  BR(outer_loop) // perform test again!
```

cycle	100	101	102	103	104	105	106	107	108
IF	MULC	LD	ADD	BNE	BNE	BNE	BR	SUBC	MULC
RF	SUBC R0	MULC	LD	ADD	ADD	ADD	BNE	NOP	SUBC
ALU	NOP	SUBC	MULC	LD	NOP	NOP	ADD	BNE	NOP
MEM	BNE	NOP	SUBC	MULC	LD	NOP	NOP	ADD	BNE
WB	ADDC	BNE	NOP	SUBC	MULC	LD	NOP	NOP	ADD

Please indicate on which cycle(s), 100 through 108, each of the following actions occurred. If the action did not occur in any cycle, write "NONE". You may wish to refer to the signal names in the 5-stage Pipelined Beta Diagram included in the reference material.

100, 105, 106, 108

Register value used from Register File: ______ ?

Register value bypassed from ALU stage to RF stage: 101, 102

Register value bypassed from MEM stage to RF stage: ~~104~~ None

Register value bypassed from WB stage to RF stage: ~~None~~ 105

NoNo **IRSrc^{IF} was 1:** ~~None~~ 106.

IRSrc^{IF} was 2: None

STALL was 1: 104, ~~105~~ 105

PCSEL was 1: 106.

WDSEL was 2: 105.

Problem 2.

The following program fragments are being executed on the 5-stage pipelined Beta described in lecture with full bypassing, stall logic to deal with LD data hazards, and speculation for JMPs and taken branches (i.e., IF-stage instruction is replaced with a NOP if necessary). The execution pipeline diagram is shown for cycle 1000 of execution. Please fill in the diagram for cycle 1001; use "?" if you cannot tell what opcode to write into a stage. Then for **both** cycles use arrows to indicate any bypassing from the ALU/MEM/WB stages back to the RF stage (see example for cycle 1000 in part A).

(A) (2 points) **Assume BNE is taken.**

```
    …
    ADDC(R1,5,R1)
L:  SUBC(R1,1,R1)
    SHRC(R0,1,R0)
    BNE(R1,L)
    ST(R1,data)
    …
```

Cycle	*1000*	*1001*
IF	ST	SUBC
RF	BNE	NOP
ALU	SHRC	BNE
MEM	SUBC	SHRC
WB	NOP	SUBC

(B) (2 points)

```
…
ST(R31,0,BP)
LD(BP,-12,R17)
ADDC(SP,4,SP)
SHLC(R17,2,R1)
ST(R1,-4,SP)
BEQ(R31,fact,LP)
…
```

Cycle	*1000*	*1001*
IF	ST	ST
RF	SHLC	SHLC
ALU	ADDC	NOP
MEM	LD	ADDC
WB	ST	LD.

(C) (2 points)

```
…
XOR(R1,R2,R1)
MULC(R2,3,R2)
SUB(R2,R1,R3)
AND(R3,R1,R2)
ADD(R3,R2,R3)
ST(R3,x)
…
```

Cycle	*1000*	*1001*
IF	ADD	ST
RF	AND	ADD
ALU	SUB	AND
MEM	MULC	SUB
WB	XOR	MULC

(D) (2 points) Assume during cycle 1000 the DIV instruction in the RF stage **triggers an ILLEGAL OPCODE (ILLOP) exception**.

```
…
LD(x,R1)
LD(y,R2)
SHLC(R1,3,R1)
DIV(R2,R1,R3)
ADDC(R3,17,R3)
ST(R3,z)
…
```

Cycle	*1000*	*1001*
IF	ADDC	?
RF	DIV	NOP.
ALU	SHLC	BNE(R31, 0, XP)
MEM	NOP	SHLC
WB	LD	NOP.

remember the next PC.

Problem 3.

```
     …
L1:  SUBC(R0,4,R0)
     CMPLTC(R0,10,R1)
     BF(R1,L2)
     LD(R0,A,R2)
     BR(L3)
L2:  LD(R0,B,R2)
L3:  ST(R2,C,R31)
     BNE(R0,L1)
     ADDC(R2,1,R2)
     …
```

In answering this question, you may wish to refer to the diagram of the 5-stage pipelined beta provided with the reference material.

The loop on the right has been executing for a while on our standard 5-stage pipelined Beta with branch annulment and full bypassing. The pipeline diagram below shows the opcode of the instruction in each pipeline stage during 10 consecutive cycles of execution.

Cycle #	*300*	*301*	*302*	*303*	*304*	*305*	*306*	*307*	*308*	*309*	
IF	SUBC	CMPLTC	BF	LD	LD	ST	BNE	BNE	BNE	ADDC	Subc
RF	NOP	SUBC	CMPLTC	BF	NOP	LD	ST	ST	ST	BNE	NOP.
ALU	BNE		SUBC	CMPLTC	BF	NOP	LD	NOP	NOP	ST	BNE
MEM	ST			SUBC	CMPLTC	BF	NOP	LD	NOP	NOP	
WB	NOP.				SUBC	CMPLTC	BF	NOP	LD	NOP	

(A) (4 Points) Indicate which bypass/forwarding paths are active in each cycle by drawing a vertical arrow in the pipeline diagram from pipeline stage X in a column to the RF stage in the same column if an operand would be bypassed from stage X back to the RF stage that cycle. Note that there may be more than one vertical arrow in a column.

Draw bypass arrows in pipeline diagram above

(B) (2 Points) Assume that the previous iteration of the loop executed the same instructions as the iteration show here. Please complete the pipeline diagram for cycle 300 by filling in the OPCODEs for the instructions in the RF, ALU, MEM, and WB stages.

Fill in OPCODEs for Cycle 300

For the following questions *think carefully* about when a signal would be asserted in order to produce the effect you see in the pipeline diagram.

(C) (2 Points) During which cycle(s), if any, would the $IRSrc^{IF}$ signal be 1?

$IRscr^{IF}$ ==1 , when taken branch in RF.

Cycle number(s) or NONE: 309 303

(D) (2 Points) During which cycle(s), if any, would the $IRSrc^{RF}$ signal be 1?

when register operand. is not available in dat path.

Cycle number(s) or NONE: ~~303~~ 306, 307

(E) (2 Points) During which cycle(s), if any, would the STALL signal be 1, *i.e.*, cycle(s) when the IF and RF stages would be stalled?

#E =40.

Cycle number(s) or NONE: 306, 307.

Problem 4.

You've discovered a secret room in the basement of the Stata center full of discarded 5-stage pipelined Betas. Unfortunately, many have certain defects. You discover that they fall into four categories:

- **C1:** Completely functional 5-stage Betas with working bypass paths, annulment, and other components.
- **C2:** Betas with a bad register file: all data read from the register file is zero.
- **C3:** Betas without bypass paths: all source operands come from the register file.
- **C4:** Betas without annulment of instructions following branches.

To help sort the Beta chips into the above classes, you write the following small test program:

```
. = 0x0
// Start at 0x0, with ZERO in all registers…
      ADDC(R31, 4, R0)
      BEQ(R31, X, R2)
      MULC(R2, 2, R2)
X:    SUBC(R2, 4, R2)
      ADD(R0, R2, R3)
      JMP(R3)
```

0 4 8

$R_0 \leftarrow 4$

$R_2 \leftarrow 8$

$R_2 \leftarrow 8$ 4.

$R_3 \leftarrow$ C.

JMP C

$R_2 \leftarrow 10.$

$R_2 = 24.$

20.

Your plan is to single-step through the program using each Beta chip, carefully noting the address the final JMP loads into the PC. Your goal is to determine which of the above four classes a chip falls into by this JMP address.

For each class of Beta processor described above, specify the value that will be loaded into the PC by the final JMP instruction.

Pipeline diagram showing first 7 cycles of test program executing on C1:

$R_2 = 12.$

cycle	*0*	*1*	*2*	*3*	*4*	*5*	*6*
IF	ADDC	BEQ	MULC	SUBC	ADD	JMP	
RF		ADDC	BEQ	NOP	SUBC	ADD	JMP
ALU			ADDC	BEQ	NOP	SUBC	ADD
MEM				ADDC	BEQ	NOP	SUBC
WB					ADDC	BEQ	NOP

C1: JMP goes to address: 8.

C2: JMP goes to address: 4.

C3: JMP goes to address: 0

C4: JMP goes to address: 16

Problem 5.

Recall the code for gcd that we saw in lecture, and the assembly code for the while loop:

C code

```c
int gcd(int x, int y) {
  while (x != y) {
    if (x > y) {
      x = x - y;
    } else {
      y = y - x;
    }
  }
  return x;
}
```

Corresponding Beta assembly for while loop

```
       // x in R0, y in R1
       CMPEQ(R0, R1, R2)   // R2 ← (x == y)
       BT(R2, end)
loop:  CMPLT(R1, R0, R2)   // R2 ← (x > y)
       BF(R2, else)
       SUB(R0, R1, R0)     // x ← x - y
       BR(cond)
else:  SUB(R1, R0, R1)     // y ← y - x
cond:  CMPEQ(R1, R0, R2)   // R2 ← (x == y)
       BF(R2, loop)
end:   …
```

Assume a **5-stage pipelined Beta** as presented in lecture, with **full bypass paths**, and which **predicts branches by assuming they are not taken** to resolve control (i.e., the instruction following the branch is fetched in the IF stage on the cycle after the branch is in the IF stage).

First, find the number of cycles per iteration in steady state (do not worry about the first or last iterations). Note that the BF(R2, else) branch is not taken if `x > y` and taken if `x < y`, so you should consider these two cases separately.

(A) Fill in the following table:

	Iterations where x > y	**Iterations where x < y**
Instructions per iteration	6	5
+ Cycles lost to data hazards	0	0
+ Cycles lost to annulments	2	2
= Total cycles per iteration	8	7

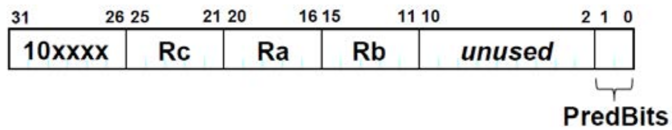
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
IF	CMPLT	BF	sub	cond	sub	CMPEQ	BF	?	CMPLT					
RF		CMPLT	BF	sub	cond	NOP.	CMPEQ	BF	NOP					
ALU			CMPLT	BF	sub	cond	NOP	CMPEQ	BF					
MEM				CMPLT	BF	sub	cond	NOP						
WB					CMPLT	BF	sub.	cond						

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
IF	CMPLT	BF	sub	sub	CMPEQ	BF	?	CMPLT						
RF		CMPLT	BF	NOP	sub	CMPEQ	BF	NOP-						
ALU			CMPLT	BF	NOP	sub	COMPEQ	BF						
MEM				CMPLT	BF	NOP	sub	COMPEQ						
WB					COMPLT	BF	NOP	sub						

To make this code faster, we modify the Beta ISA and pipeline to implement a technique called **predication** to reduce the number of branches.

First, all the compare instructions (CMPEQ, CMPLT, CMPLE, and their C variants) write their result into a special 1-bit register, called the **predicate register**, in addition to their normal destination register.

Second, we change the format of ALU instructions with two register source operands to use their lower two bits, which were previously unused:



- If PredBits == 10, the instruction only executes if the predicate register is false (0)
- If PredBits == 11, the instruction only executes if the predicate register is true (1)
- If PredBits == 0X, the instruction always executes and writes its result, as before

We say that instructions that depend on the predicate register are predicated. We denote predicated instructions in assembly as follows:

- If PredBits == 10, OP(Ra, Rb, Rc) [predFalse]
- If PredBits == 11, OP(Ra, Rb, Rc) [predTrue]
- If PredBits == 0X, OP(Ra, Rb, Rc), as before

For example, consider the following instruction sequence:

```
CMPLT(R1, R2, R3)
MUL(R3, R4, R5)
ADD(R4, R5, R6) [predTrue]
SUB(R5, R6, R7)
```

If the CMPLT instruction evaluates to true (i.e., writes 1 to R3), this sequence is equivalent to:

```
CMPLT(R1, R2, R3)
MUL(R3, R4, R5)
ADD(R4, R5, R6)
SUB(R5, R6, R7)
```

If the CMPLT instruction evaluates to false (i.e., writes 0 to R3), this sequence is equivalent to:

```
CMPLT(R1, R2, R3)
MUL(R3, R4, R5)
SUB(R5, R6, R7)
```

(B) Modify the code to use predication, minimizing the number of instructions per loop iteration.

Original code

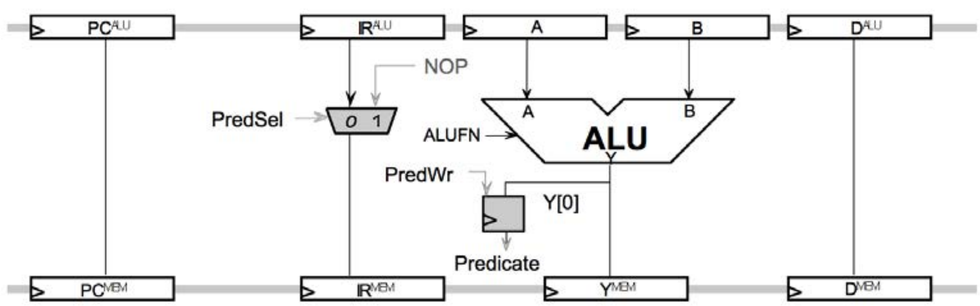
```
       // x in R0, y in R1
       CMPEQ(R0, R1, R2)
       BT(R2, end)
loop:  CMPLT(R1, R0, R2)
       BF(R2, else)
       SUB(R0, R1, R0)
       BR(cond)
else:  SUB(R1, R0, R1)
cond:  CMPEQ(R1, R0, R2)
       BF(R2, loop)
end:   …
```

Code with predication

```
       // x in R0, y in R1
       CMPEQ(R0, R1, R2)
       BT(R2, end)
loop:  CMPLT(R1, R0, R2)
       SUB(R0, R1, R0) [predTrue]
       SUB(R1, R0, R1) [Pred False]
       CMPEQ(R1, R0, R2)
       BF(R2, loop).

end:   …
```

We implement predication in the pipelined Beta with minor changes to the ALU stage:



Comparison instructions write the 1-bit predicate register (the PredWr control signal ensures that only comparison instructions update the register). The PredSel mux annuls ALU instructions if they are predicated and should not execute according to the value of the predicate register.

(C) Write the Boolean expression for the PredSel control signal. You can use AND, OR, NOT, Predicate, and comparisons with PredBits (e.g., PredBits == 0b10).

PredSel = (IR^{ALU}[31:30] == 0b10) AND Predict[1] == 1 AND Predict[0] != Predict.

(D) How fast is this modified code? Fill in the following table:

	Iterations where x > y	**Iterations where x < y**
Instructions per iteration	4	4
+ Cycles lost to data hazards	0	0
+ Cycles lost to annulments	2	2
= Total cycles per iteration	6	6