Name ___________________________

Computer System Architecture
6.823 Quiz #1

March 6, 2015

Professors Daniel Sanchez and Joel Emer

This is a closed book, closed notes exam.

80 Minutes

18 Pages

Notes:

- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not yet taken the quiz.

Part A	_________	27 Points
Part B	_________	38 Points
Part C	_________	35 Points

TOTAL	______	**100 Points**

Name ____________________________

Part A: Self-Modifying Code (27 Points)

In this problem we will use and extend the EDSACjr instruction set from Handout 1, shown in Table A-1.

Opcode	**Description**
ADD *n*	Accum ← Accum + M[*n*]
SUB *n*	Accum ← Accum - M[*n*]
LD *n*	Accum ← M[*n*]
ST *n*	M[*n*] ← Accum
CLEAR	Accum ← 0
OR *n*	Accum ← Accum | M[*n*]
AND *n*	Accum ← Accum & M[*n*]
SHIFTR *n*	Accum ← Accum shiftr *n*
SHIFTL *n*	Accum ← Accum shiftl *n*
BGE *n*	If Accum ≥ 0 then PC ← *n*
BLT *n*	If Accum < 0 then PC ← *n*
END	Halt machine

Table A-1. EDSACjr Instruction Set

Question 1 (15 Points)

Write a program that loops over an n-item array and replaces each item with its absolute value, as shown in the following pseudo-code:

```
for (i = 0; i < n; i++)
    A[i] = |A[i]|
```

Part of the program is already written for you, and to simplify your job you can assume the loop will be executed only once. The memory map on the next page shows the memory contents before the program starts. Array A is stored in memory in a contiguous manner, starting from location A. Memory locations N, I, and ONE hold the values of n, i, and 1, respectively. If you need to, you can use additional memory locations for your own variables. You should label each variable and define its initial value.

Name ______________________________

Memory:

	…
A	A[0]
	A[1]
	…
	A[n-1]
	…
ONE	1
N	n
I	0
TMP	0

Program:

```
loop:   LD      I
        SUB     N
        BGE     done

I1:     LD      A
        BGE     cont
        ST      TMP
        CLEAR
        SUB     TMP
I2:     ST      A

cont:   LD      I1
        ADD     ONE
        ST      I1
        LD      I2
        ADD     ONE
        ST      I2









        LD      I
        ADD     ONE
        ST      I
        BGE     loop
done:   END
```

Question 2 (12 Points)

Tired of writing self-modifying code, Ben Bitdiddle decides to extend EDSACjr to support indirect addressing. However, because registers are expensive, Ben does not want to add an index register. Instead, he implements the indirect addressing instructions shown in Table A-2. To execute an indirect addressing instruction, the new architecture first reads the target address from memory and then loads/stores the data from/to memory.

Opcode	**Description**
ADDind *n*	Accum ← Accum + M[M[*n*]]
SUBind *n*	Accum ← Accum – M[M[*n*]]
LDind *n*	Accum ← M[M[*n*]]
STind *n*	M[M[*n*]]← Accum

Table A-2. Additional Indirect Addressing Instructions

Using the instructions in Table A-1 and Table A-2, rewrite the program from Question 1 without using self-modifying code. As before, you can use additional memory locations for your own variables. You should label each variable and define its initial value.

Name ______________________________

Memory:

	…
A	A[0]
	A[1]
	…
	A[n-1]
	…
ONE	1
N	n
I	0
IDX	A

Program:

```
loop:  LD       I
       SUB      N
       BGE      done

       LDind    IDX
       BGE      cont
       CLEAR
       SUBind   IDX
       STind    IDX

cont:  LD       IDX
       ADD      ONE
       ST       IDX










       LD       I
       ADD      ONE
       ST       I
       BGE      loop
done:  END
```

Name ______________________________

Part B: Caches and Virtual Memory (38 pts)

Question 1 (13 points)

Consider a reference stream that repetitively **loops over four addresses, A, B, C, and D (ABCDABCDABCD….)**. We will study how different replacement policies perform on this reference stream, using a small, 2-entry, fully-associative cache.

1. Find out how the cache performs with LRU replacement. Fill the table below to show the cache contents over time, and note whether each access is a hit or a miss. Then, compute the long-term miss ratio (i.e., discounting cache warm-up). (3 points)

Access	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Address	A	B	C	D	A	B	C	D	A	B	C	D
Entry 1	-	A	A	C	C	A	A	C	C	A	A	C
Entry 2	-	-	B	B	D	D	B	B	D	D	B	B
Hit?	N	N	N	N	N	N	N	N	N	N	N	N

What is the long-term miss ratio under LRU? 100%

2. Find out how the cache performs under optimal replacement. **This cache cannot bypass accesses, i.e., on every miss, it must replace an existing block and insert the new block.** Fill the time diagram below, and find the long-term hit rate. (5 points)

Access	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Address	A	B	C	D	A	B	C	D	A	B	C	D
Entry 1	-	A	A	A	A	A	B	C	C	C	C	C
Entry 2	-	-	B	C	D	D	D	D	D	A	B	B
Hit?	N	N	N	N	Y	N	N	Y	N	N	Y	N

What is the long-term miss ratio under optimal replacement? 66% (2/3)

3. In the example, is there a simple policy that, without knowing the future, performs as well as the optimal one? If so, which one? (5 points)

   Yes, MRU (Most Recently Used)

Question 2 (9 points)

Consider a byte addressing system with **16-bit virtual and physical addresses**. The system has a cache with the following properties:

- 8 sets with 128 bytes per block
- 4-way set-associative organization
- Virtually-indexed, physically-tagged

1. Suppose we use **256-byte pages**. Where in the cache can **virtual address** 0xABCD live? Please use crosses (X) to mark its possible locations in the diagram below. (The binary representation of 0xABCD is 1010 1011 1100 1101.) (3 points)

Index	**Cache Contents**			
	Way 0	Way 1	Way 2	Way 3
0				
1				
2				
3				
4				
5				
6				
7	X	X	X	X

```
              bit  15                 0
Virtual address: 1010 1011 1100 1101
Index: 111 (bit 9-7)
```

2. As before, suppose we use **256-byte pages**. Where in the cache can **physical address** 0xABCD live? Please use crosses (X) to mark its possible locations. (3 points)

Index	**Cache Contents**			
	Way 0	Way 1	Way 2	Way 3
0				
1	X	X	X	X
2				
3	X	X	X	X
4				
5	X	X	X	X
6				
7	X	X	X	X

```
                  bit  15                          0
Physical address: 1010 1011 1100 1101
Virtual address:   xxxx xxxx 1100 1101
Index: xx1 (bit 9-7)
```

3. Suppose we use **1024-byte pages** instead. Where in the cache can **physical address** 0xABCD live? Please use crosses (X) to mark its possible locations. (3 points)

Index	**Cache Contents**			
	Way 0	Way 1	Way 2	Way 3
0				
1				
2				
3				
4				
5				
6				
7	X	X	X	X

```
                  bit  15                          0
Physical address: 1010 1011 1100 1101
Virtual address:   xxxx xx11 1100 1101
Index: 111 (bit 9-7)
```

Name ______________________________

Question 3 (16 points)

We'd like our memory system to support **two page sizes**: **256-byte small pages** and **1024-byte large pages**. A common approach to support multiple page sizes is to use separate TLBs, one for each page size. Instead, to reduce area overheads, we will use a single TLB to cache translations of both small and large pages, shown in Figure B-1. The TLB has 8 sets and 2 ways. The L bit denotes whether the cached PTE is for a large page.

V = valid bit **L = large page bit (set to 1 when a large page is stored)**
PPN = physical page number

	Way 0				Way 1			
	V	L	Tag	PPN	V	L	Tag	PPN
0								
1								
2								
3								
4								
5								
6								
7								

Figure B-1. TLB for multiple page sizes.

Each TLB access consists of three steps. First, the TLB checks for a small-page match, using the tag and index bits shown in Figure B-2. Second, if it does not find a small-page match, it checks for a large-page match, using the tag and index bits in Figure B-3. Third, if the second lookup misses as well, it results in a TLB miss and a page table walk.

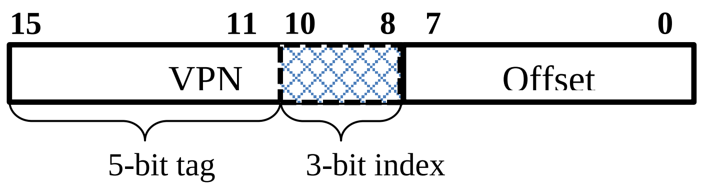

Figure B-2. Tag and index bits for small (256-byte) pages.

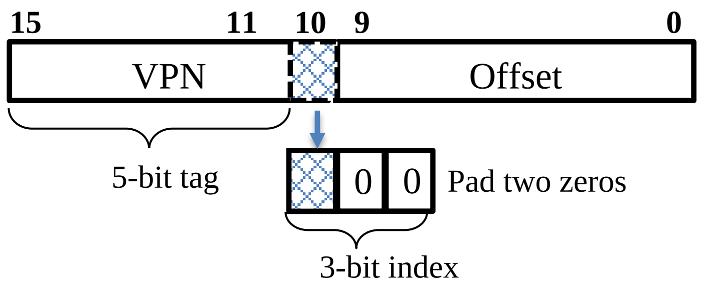

Figure B-3. Tag and index bits for large (1024-byte) pages.

Name ______________________________

Assume virtual address 0xABBA translates to physical address 0x47BA.

1. If virtual address 0xABBA **belongs to a small (256-byte) page**, fill in the fields of the TLB entry, and mark all possible TLB locations it can be in. (3 points)

TLB entry

L	Tag	PPN
0	10101	0x47

0xABBA	= 1010 1011 1011 1010
0x47BA	= 0100 0111 1011 1010
VPN	= 1010 1011
PPN	= 0100 0111
Page offset	= 1011 1010
Index to TLB	= 011 (from VPN bit 8-10)
Tag	= 10101 (from VPN bit 11-15)

Possible locations

	Way 0	Way 1
0		
1		
2		
3	X	X
4		
5		
6		
7		

2. If virtual address 0xABBA **belongs to a large (1024-byte) page**, fill in the fields of the TLB entry, and mark all possible TLB locations it can be in. (3 points)

TLB entry

L	Tag	PPN
1	10101	0100 01

0xABBA	= 1010 1011 1011 1010
0x47BA	= 0100 0111 1011 1010
VPN	= 1010 10
PPN	= 0100 01
Page offset	= 11 1011 1010
Index to TLB	= 000 (from VPN bit 10 + 00)
Tag	= 10101 (from VPN bit 11-15)

Possible locations

	Way 0	Way 1
0	X	X
1		
2		
3		
4		
5		
6		
7		

Name ______________________________

3. What is the reach of this TLB? (TLB reach = maximum amount of memory accessible without TLB misses) (4 points)

2*2*1K page + 6*2*256 page = 7K

4. This TLB has a utilization problem for large pages. Explain why it happens and how to solve it. (6 points)

Problem: Padding zeros limits large pages to locate only in entry 0 and 4.

Do not pad index bits with 00 for large pages but use the first 3 bits from the VPN of large pages.

Fixed reach: 8*2*1K page = 16K

Name ____________________________

HAL 180 ISA and 6-Stage Pipelined Implementation

Inspired by how the IBM 360 uses condition codes, Ben Bitdiddle designs the HAL 180 architecture, which features two flag registers. Table C-1 describes these flags.

Name	**Description**
Sign Flag (SF)	Stores 1 if the result of the *last arithmetic or comparison instruction* was negative, 0 if it was non-negative
Zero Flag (ZF)	Stores 1 if the result of the *last arithmetic, logical, or comparison instruction* was zero, and 0 if it was non-zero

Table C-1. HAL 180 status flags.

Table C-2 summarizes the different instruction types and the flags they read or write. The SF and ZF columns have an "R" when the instruction reads the status flag, a "W" if it writes the flag (and does not read it), or a blank if the instruction does not affect the status flag. For example, `JL` (jump if less than) reads SF; `ADD` writes all flags; and `JMP` (unconditional jump) does not affect any flag. Some instructions, like `CMP`, write the status flags but do not return any result.

Instruction	**Description**	**SF**	**ZF**
Arithmetic Instructions			
ADD *s1, s2*	$s1 \leftarrow s1 + s2$	W	W
SUB *s1, s2*	$s1 \leftarrow s1 - s2$	W	W
MUL *s1, s2*	$s1 \leftarrow s1 \times s2$	W	W
Logical Instructions			
AND *s1, s2*	$s1 \leftarrow s1 \,\&\, s2$		W
OR *s1, s2*	$s1 \leftarrow s1 \mid s2$		W
XOR *s1, s2*	$s1 \leftarrow s1 \wedge s2$		W
Comparison Instructions			
CMP *s1, s2*	$temp \leftarrow s1 - s2$	W	W
Jump Instructions			
JMP *target*	jump to the address specified by *target*		
JL *target*	jump to *target* if SF == 1	R	
JG *target*	jump to *target* if SF == 0 and ZF == 0	R	R
Memory Instructions			
LD *s1, s2*	$s1 \leftarrow M[s2]$		
ST *s1, s2*	$M[s1] \leftarrow s2$		

Table C-2. HAL 180 instruction set.

Ben also designs a 6-stage pipelined implementation of the HAL 180. In this pipeline, the ALU takes three pipeline stages (E1, E2, and E3), and status flags are updated in stage E3. Table C-3 describes each stage, and Figure C-4 shows the datapath of this 6-stage pipelined architecture, highlighting the differences with a conventional MIPS pipeline. **Note that this implementation does not have any data bypass paths.**

Stage	Description
Fetch and Decode Stage (FD)	Fetch an instruction from the instruction memory, decode the instruction, and fetch the register values from the register file. The status flag checking for conditional jumps is also done in this stage.
Execute Stage 1 (E1)	The first stage of the execution phase. Generate partial results and store them in the pipeline registers.
Execute Stage 2 (E2)	The second stage of the execution phase. Generate partial results and store them in the pipeline registers.
Execute Stage 3 (E3)	The final stage of the execution phase. Final results are generated and flag registers get updated if necessary.
Memory Stage (M)	Perform load/store from/to the data memory if necessary.
Writeback Stage (WB)	Write to the register file if necessary.

Table C-3. HAL 180 pipeline stages.

Name ______________________________

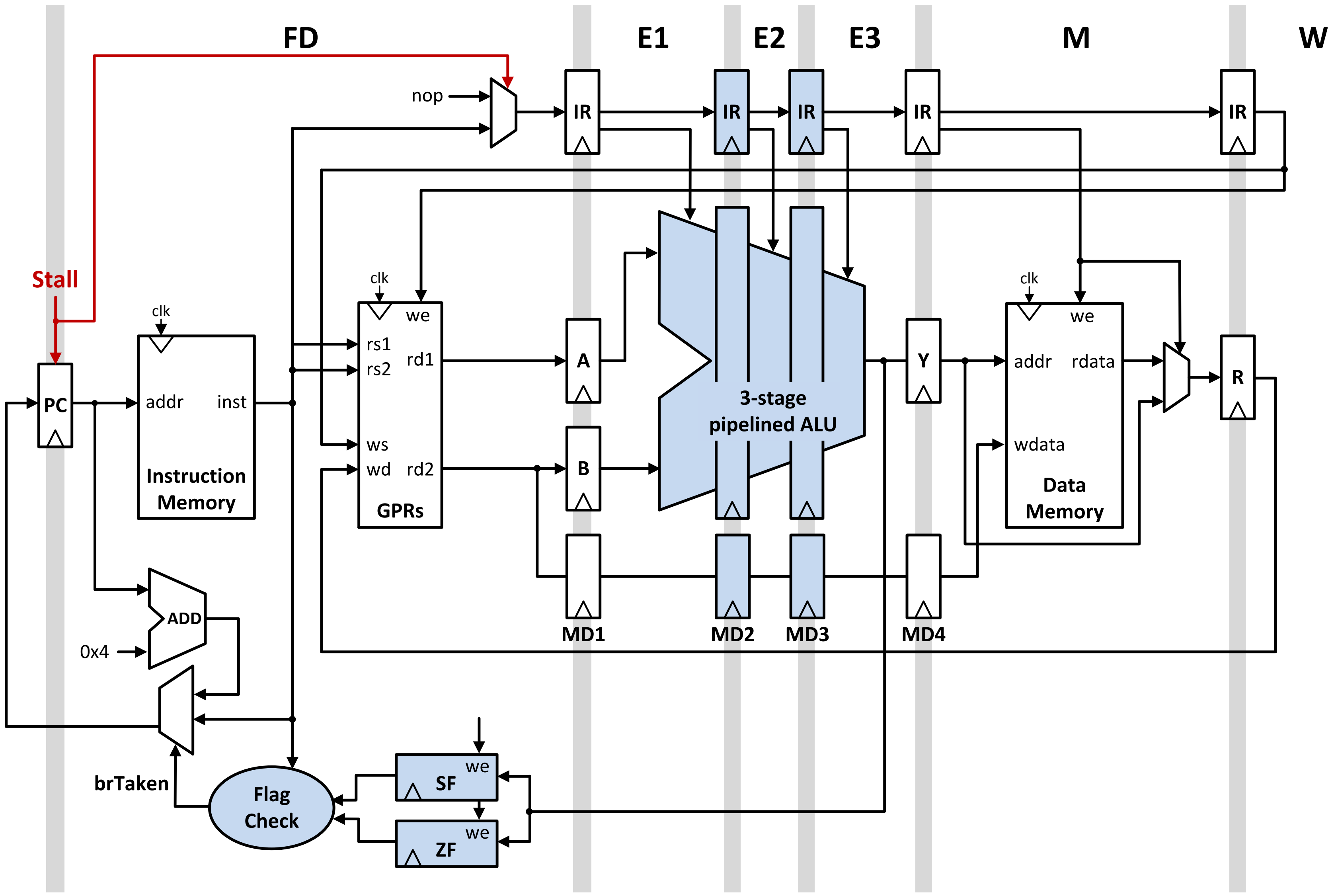

Figure C-4. HAL 180 6-Stage pipelined implementation.

Part C: Status Flags (35 Points)

Question 1 (12 points)

Write the HAL 180 assembly for the following program. For maximum credit, use the minimum number of comparison and jump instructions.

```
if (a < b) {
     c = c XOR b;
} else if (a > b) {
     c = c XOR a;
} else {
     c = 0;
}
a = 0;
b = 0;
```

Assume variables a, b, and c are stored in registers **R1, R2, and R3** respectively.

```
            CMP          R1, R2
            JL           _L1
            JG           _L2
            XOR          R3, R3
            JMP          _L3
_L1:        XOR          R3, R2
            JMP          _L3
_L2:        XOR          R3, R1
_L3:        XOR          R1, R1
            XOR          R2, R2
```

(You get full grades if you have better answer)

Name ______________________________

Question 2 (10 points)

Ben's HAL 180 6-stage pipeline (Figure C-4) stalls to avoid data hazards through registers, but does not yet handle hazards due to status flags. To illustrate why this is problematic, consider the following instruction sequence:

I0:		ADD	R1, R2	Set	SF = 1	ZF = 0
I1:		JG	_L2	Not	Taken	
I2:		XOR	R1, R3	Set	ZF = 0	
I3:		JL	_L2	Taken		
I4:	_L1:	SUB	R1, R2			
I5:	_L2:	ADD	R3, R1			

Assume that when the program start, R1 = -1, R2 = -2, R3 = -3, and all the status flags are zero. Fill out the following instruction flow diagram to incur the minimum amount of stalls while maintaining correct operation (i.e., use stalls to respect both data and status flag dependences). Use "X"s to denote pipeline bubbles.

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
FD	I0	I1	I1	I1	I1	I2	I2	I3	I5	I5
E1		I0	X	X	X	I1	X	I2	I3	X
E2			I0	X	X	X	I1	X	I2	I3
E3				I0	X	X	X	I1	X	I2
M					I0	X	X	X	I1	X
W						I0	X	X	X	I1

Name ______________________________

Question 3 (6 points)

Let's fix Ben's implementation by extending the existing stall control signal, which already works for register hazards, to also stall on status flag hazards.

First, derive the stall conditions for the different jumps: $\texttt{JMP}_{\texttt{stall}}$, $\texttt{JL}_{\texttt{stall}}$, and $\texttt{JG}_{\texttt{stall}}$. Use $\texttt{Opcode}_{\texttt{X}}\texttt{(Y)}$ to indicate the condition when the instruction in `X` stage is `Y`. `Y` can be a specific instruction or an instruction class (see Table C-2). For example:

$\texttt{Opcode}_{\texttt{FD}}\texttt{(JG)}$:	if the instruction in the `FD` stage is a `JG` instruction.
$\texttt{Opcode}_{\texttt{E1}}\texttt{(Logic)}$:	if the instruction in the E1 stage belongs to the logical instruction class (e.g. `OR`).
$\texttt{Opcode}_{\texttt{E2}}\texttt{(CMP|Arith)}$:	if the instruction in the E2 stage is a `CMP` instruction or belongs to the arithmetic instruction class.

$\texttt{JMP}_{\texttt{stall}}$ = 0

$\texttt{JG}_{\texttt{stall}}$ = $\texttt{Opcode}_{\texttt{E1}}\texttt{(logic|Arith|CMP)|Opcode}_{\texttt{E2}}\texttt{(logic|Arith|CMP)|}$
$\texttt{Opcode}_{\texttt{E3}}\texttt{(logic|Arith|CMP)}$

$\texttt{JL}_{\texttt{stall}}$ = $\texttt{Opcode}_{\texttt{E1}}\texttt{(Arith|CMP)|Opcode}_{\texttt{E2}}\texttt{(Arith|CMP)|}$
$\texttt{Opcode}_{\texttt{E3}}\texttt{(Arith|CMP)}$

Finally, write down the new stall signal (`stall'`) by using the old stall signal (`stall`) and stall conditions you derive.

`stall'` = $\texttt{stall | (Opcode}_{\texttt{FD}}\texttt{(JL) \& JL}_{\texttt{stall}}\texttt{) |}$
$\texttt{(Opcode}_{\texttt{FD}}\texttt{(JG) \& JG}_{\texttt{stall}}\texttt{)|}$

Question 4 (7 points)

Does this 6-stage pipeline add more challenges to precise exception handling? If so, please explain.

Yes. Since the status flags are set in E3 stages, you will need some mechanism to roll back in order to handle exceptions after E3 stages.