Name DAN PORTS

Computer System Architecture
6.823 Quiz #1
October 6th, 2006
Professor Krste Asanovic
Dr. Joel Emer

Name: DAN PORTS

This is a closed book, closed notes exam.
80 Minutes
17 Pages

Notes:

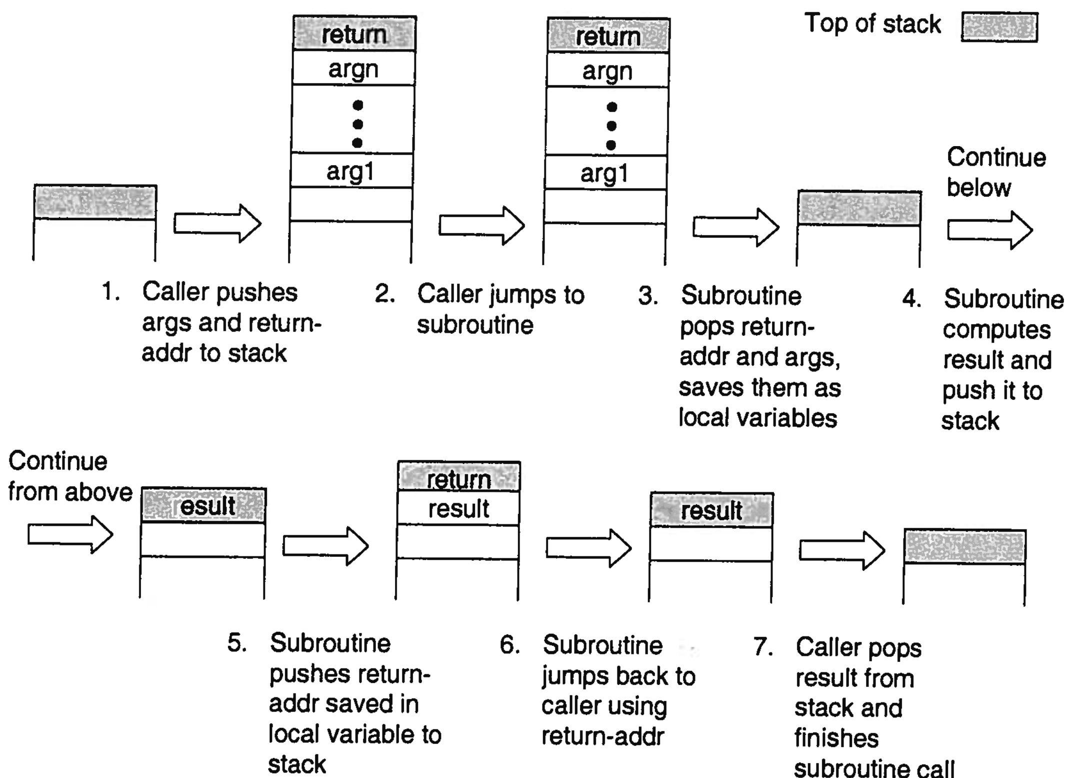
- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not yet taken the quiz.

Writing name on each sheet	2	2 Points
Question 1	20	23 Points
Question 2	27	32 Points
Question 3	23	23 Points
TOTAL	72	**80 Points**

Name Dan Ports

Question 1: Self-modifying Code on the 6.823 Stack ISA (23 points)

After having studied self-modifying code on the EDSACjr, Ben Bitdiddle decides to derive a subroutine calling convention for the 6.823 stack ISA similar to that of the EDSACjr presented in pset1 Problem M1.1.B. The following diagram explains the steps of a subroutine call according to his calling convention and the state of the stack after each step.



In this question, we will try to help Ben Bitddidle realize his calling convention. The following global variables are defined for you to use.

```
_goto_template:   GOTO 0    ; GOTO template, address field set as 0
_zero:            0         ; constant value 0
_one:             1         ; constant value 1
_two:             2         ; constant value 2
_three:           3         ; constant value 3
_four:            4         ; constant value 4
```

These global variables are located somewhere in main memory and can be accessed using their labels. You can also declare local variables and refer to them using labels in a similar fashion. In this question, assume that each instruction is 16 bits long (4 bits for the

Name DAN PORTS

opcode, 12 bits for the memory address). The memory is byte-addressable. The maximum size of memory supported in this machine is 4096 Bytes.

The following table gives the meaning of each instruction.

Instruction	Meaning
PUSH *A*	push M[*A*] onto stack
POP *A*	pop stack and place popped value in M[*A*]
ADD	pop two values from the stack; ADD them; push result onto stack
SUB	pop two values from the stack; SUBtract top value from the 2nd; push result onto stack
MUL	pop two values from the stack; MULtiply them; push result onto stack
ZERO	zeroes out the value at top of stack
INC	pop value from top of stack; increments value by one push new value back on the stack
BEQZ *A*	pop value from stack; if it's zero, next instruction will be M[*A*]; else, continue with next instruction
BNEZ *A*	pop value from stack; if it's not zero, next instruction will be M[*A*]; else, continue with next instruction
GOTO *A*	Next instruction to be executed will be M[*A*]
.fill *A*	Extend the address field *A* with 0s to fit the length of an instruction; (This is not exactly an executable instruction, it just provides you a way to specify the content at the current instruction location.)

Part A (9 points)

We will start with writing the macro "**GOTOind**". The meaning of this macro is to pop a value from the top of the stack and to continue execution at the location pointed by the popped value. This macro is useful when the subroutine jumps back to the caller's return address.

```
.macro GOTO ind
    PUSH _goto_template
    ADD
    POP _goto
_goto:  .fill 0      ✓
.end
```

9

Name DAN PORTS

Part B (9 points)

Now, we will write another macro "**GOTOAL** *A*". The meaning of this macro is to push the return address (i.e. the address just after the macro) onto the stack and to continue execution at location M[*A*]. This macro is useful when the caller calls the subroutine.

```
.macro GOTOAL (A)
    PUSH _end_of_macro
    PUSH _two              // increment by 2 since
    ADD                    // instructions are 16 bits
    PUSH A                 // and mem. is byte-addressable
    GOTO IND
_end_of_macro: .fill _end_of_macro
.end
```

— we want "GOTO A". not "GOTO M[A]".

✗ 6

Part C (5 points)

Ben's subroutine calling convention doesn't support recursive subroutine calls. Can you describe the problem in less than two sentences?

In step 3, the subroutine pops the return address and args from the stack and stores them in local variables; if the subroutine is recursively invoked it will store the new return addr and args in the same local variables, overwriting the old.

✓ 5

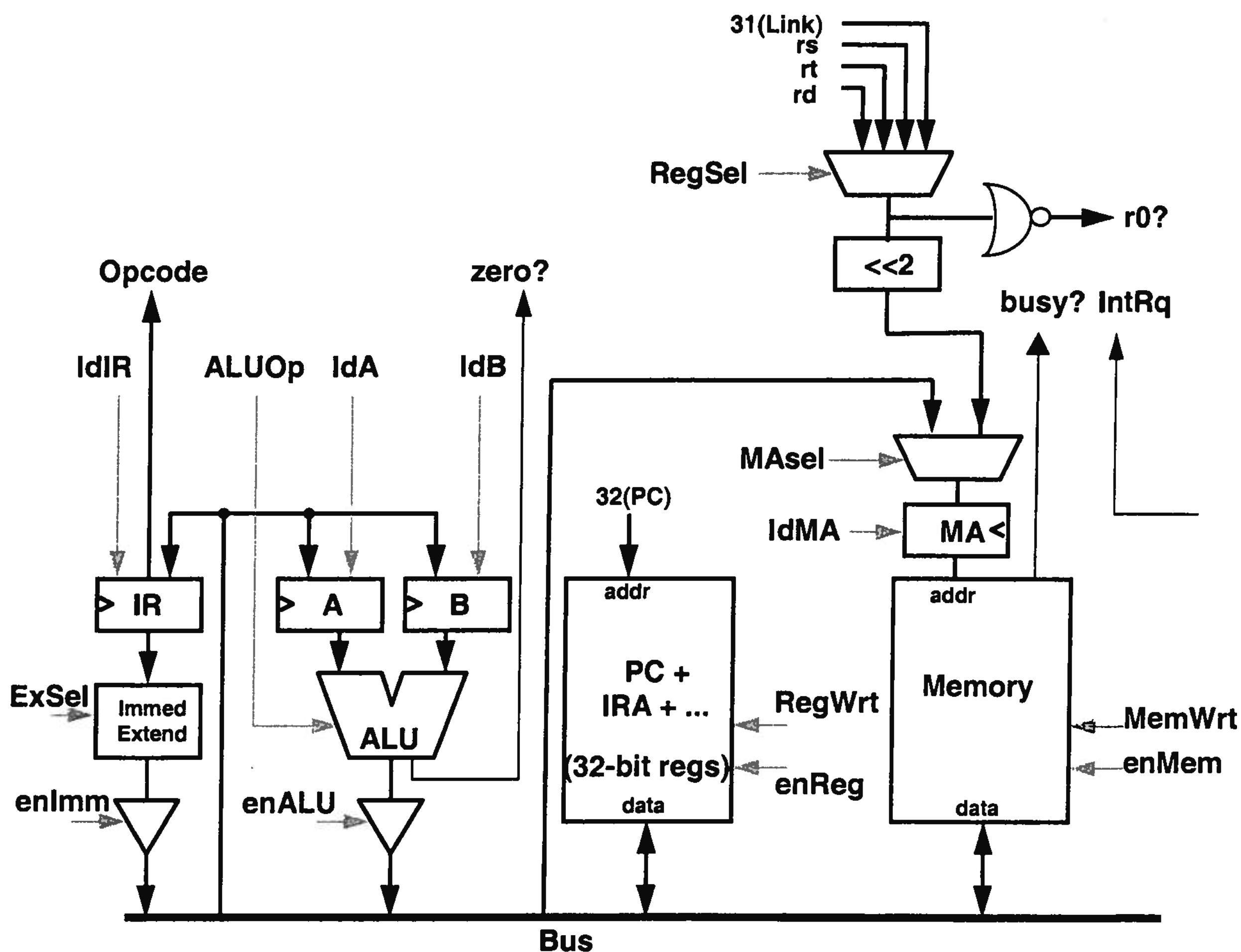
11

Name Dan Ports

Question 2: Microprogramming with Registers in Memory (32 points)

Some early microcoded machines used main memory to hold architectural registers to save on component costs. In this question, we are going to modify the MIPS microcoded machine (Handout #5) to place the 32 general-purpose registers (GPRs) in main memory.

The modified architecture is shown below. The register file is reduced to hold only the PC and a few other special registers (we'll ignore the other special registers in this question). The RegSel mux is moved to be in front of the MA register. The RegSel mux outputs a 5-bit register specifier that is either one of the rs, rt, rd fields of the current instruction in the IR, or the constant value 31. The RegSel mux output is shifted left two bits to give a word address starting at the beginning of memory, i.e., register 0 starts at byte address 0, register 1 starts at byte address 4, etc. (In this machine, byte addresses 0-127 are reserved for registers usage only). A new MAsel mux selects either the register address (reg) or the bus value (bus) to load into the MA register.



Name DAN PORTS

In this question, for the parts requiring you to implement macroinstructions with microcode sequences, solutions will be graded based on the quality of the implementation so please optimize the sequences to use the minimal number of microinstructions. Also, make sure that each microinstruction can actually complete in one cycle.

Please detach Appendices A and B and use them as references for answering this question.

Part A (8 points) 7

In the next page, we have included a microcode table for the new machine, showing the modified fetch sequence. Fill in the microcode sequence for the ADD instruction. FOR THIS PART ONLY, ignore the behavior of register 0.

For your convenience, the following is the Microcode sequence for ADD for the original machine from lecture 4.

State	Control points	μBr	Next State
ADD0	A ← Reg[rs]	N	*
ADD1	B ← Reg[rt]	N	*
ADD2	Reg[rd] ← A+B	J	FETCH0

Assuming single-cycle memory.

Name Dan Ports

State	PseudoCode	ld IR	Reg W	en Reg	ld A	ld B	ALUOp	en ALU	Reg Sel	MA Sel	ld MA	Mem W	en Mem	Ex Sel	en Imm	µBr	Next State
FETCH0:	MA <- PC; A <- PC	0	0	1	1	*	*	0	*	bus	1	*	0	*	0	N	*
	IR <- Mem	1	*	0	0	*	*	0	*	*	0	0	1	*	0	N	*
	PC <- A+4	0	1	1	0	*	INC_A_4	1	*		*	*	0	*	0	D	*
. . .																	
NOP0:	microbranch back to FETCH0	0	*	0	*	*	*	0	*	*	*	*	0	*	0	J	FETCH0
ADD:	MA ← rs	0	*	0	*	*	*	0	rs	reg	1	*	0	*	0	N	
	A ← mem	0	*	0	1	*	*	0	*	*	*	0	1	*	0	N	
can be combined	MA ← rt	0	*	0	0	*	*	0	rt	reg	1	*	0	*	0	N	
	B ← mem	0	*	0	0	1	*	0	*	*	*	0	1	*	0	N	
	MA ← rd	*	*	0	0	0	*	0	rd	reg	1	*	0	*	0	N	
	Mem ← A+B	*	*	0	*	*	ADD	1	*	*	*	1	1	*	0	J	FETCH0

Worksheet for Question 2, Part A, ADD Instruction

not optimized

Dan Ports

Name DAN PORTS

Part B (6 points) 6

The design presented so far doesn't comply with the MIPS ISA because of the behavior of register r0. Can you describe the problem and give an example instruction sequence that illustrates the problem?

R0 is always supposed to be zero, but this design doesn't enforce that! It's possible to put something else in R0

ADDI R0, R0, 42
ADD R1, R0, R1

R1 should remain unchanged, but this increments it by 42. ✓

Part C (12 points) 10

To fix the design, we add a new µBr option, R, which jumps to the Next State field if the output of the RegSel mux is zero, otherwise execution continues to the next microinstruction.

In the microcode table in the next page, fill in the microcode sequences for LW using the new R option to implement the correct behavior for register 0. You may assume that all of memory is initialized with the value 0 at machine reset.

Hint: At most one R µBr is required within each macroinstruction sequence.

For your convenience, the followings are the microcode sequences for LW for the original machine from lecture 4.

State	Control points	µBr	Next State
LW0	A ← Reg[rs]	N	*
LW1	B ← sExt16(Imm)	N	*
LW2	MA ← A+B	N	*
LW3	Reg[rt] ← Memory	J	FETCH0

Name DAN PORTS

State	PseudoCode	ld IR	Reg W	en Reg	ld A	ld B	ALUOp	en ALU	Reg Sel	MA Sel	ld MA	Mem W	en Mem	Ex Sel	en Imm	µBr	Next State
FETCH0:	MA <- PC; A <- PC	0	0	1	1	*	*	0	*	bus	1	*	0	*	0	N	*
	IR <- Mem	1	*	0	0	*	*	0	*	*	0	0	1	*	0	N	*
	PC <- A+4	0	1	1	0	*	INC_A_4	1	*		*	*	0	*	0	D	*
. . .																	
NOP0:	microbranch back to FETCH0	0	*	0	*	*	*	0	*	*	*	*	0	*	0	J	FETCH0
LW:	MA ← rs; B ← SExt16(imm)	0	*	0	*	1	*	0	rs	reg	1	*	0	sext16	1	N	
	A ← mem; if rd = 0 then FETCH0	0	*	0	1	0	*	0	rd	*	0	0	1	*	0	Z	FETCH0
	MA ← A+B	0	*	0	*	*	ADD	0 (1)	*	bus	1	*	0	*	0	N	
	A ← mem; MA ← rd	0	*	0	1	*	*	0	rd	reg	1	0	1	*	0	N	
	Mem ← A	*	*	0	*	*	COPY_A	1	*	*	0	1	1	*	0	J	FETCH0

Worksheet for Question 2, Part C, LW Instruction

DAN PORTS

Name DAN PORTS

Part D (6 points) 4

Give one advantage and two disadvantages of using memory to hold architectural registers.

Advantage

- need only a much smaller (and cheaper) register file for PC ✓

Disadvantages

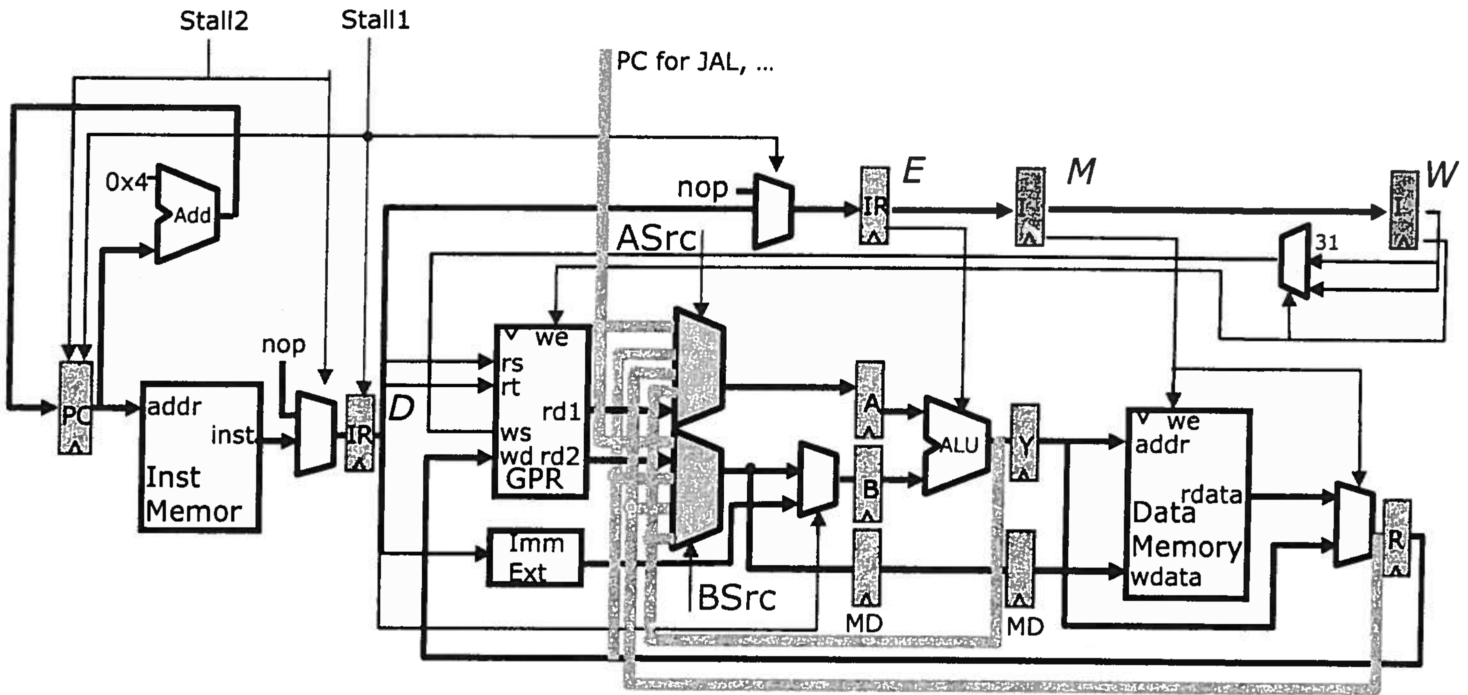
- if memory requires > 1 cycle time to access, accessing registers will be much slower ✓

- can't access RF and memory simultaneously X

Name Dall Ports

Question 3: A 5-Stage Princeton-style Pipeline (23points)

We have reproduced the fully bypassed 5-stage MIPS processor pipeline from Lecture 6 below. It has a Harvard-style architecture, i.e., separate instruction and data memories. In this problem we will ignore jump and branch instructions and self-modifying code.



Please make use of the signal names provided in the table below when writing your stall equations. Also, you are allowed to use any internal signals (e.g., opcode, PC, IR, zero?, data, etc.) but not other control signals (ASrc, BSrc, etc.).

C_{dest}	C_{re}
ws = *Case* opcode **ALU ⇒ rd** **ALUi, LW ⇒ rt** **JAL, JALR ⇒ R31** **we = *Case* opcode** **ALU, ALUi, LW ⇒ (ws ≠ 0)** **JAL, JALR ⇒ on** **... ⇒ off**	**re1 = *Case* opcode** **ALU, ALUi, LW, SW, BZ, JR, JALR ⇒ on** **J, JAL ⇒ off** **re2 = *Case* opcode** **ALU, SW ⇒ on** **... ⇒ off**

For example, if the stall signal should be set when the instruction at execution stage is an ADD and the instruction at decode stage is reading the second source register or the instruction at memory stage is SUB, then we can define the stall signal as:

Name Dan Ports

stall = ((opcode$_E$ == ADD) • re2$_D$) + (opcode$_M$ == SUB)

Part A (5 points)

Ignore the stall2 signal and the multiplexer in front of IR for this part (inst from Inst Memory is connected directly to IR).

Do we need to stall this pipeline even though it is a fully-bypassing Harvard-style machine? If so, explain why and give an example instruction sequence which causes a stall.

Yes, A load followed by an operation requiring the loaded value requires a stall. The second instruction requires the output of the first when it is in the D stage, but the output isn't available until the next cycle when the load accesses memory.

LW R1, 0(R2)

ADD R3, R1, R2

✓ 5

Part B (5 Points)

Write down the correct equation for the stall1 signal.

stall1 = (OPCODE$_{ALU}$ = LW) • we$_{ALU}$

• ((ws$_{ALU}$ = rs1$_D$) • re1$_D$

+ ws ALU = rs2$_D$) • re2$_D$) ✓ 5

10

Name Dan Porr?

Part C (8 Points)

We have now changed the MIPS processor so that it has a Princeton-style architecture, i.e., it has a shared instruction and data memory (although the instruction and data memories are shown separately, they are physically shared). The shared memory has only one read/write port. When there is a structural hazard to the shared memory, the priority goes to the instruction at the memory stage.

For this part and the next, DO NOT ignore the stall2 signal and the multiplexer in front of IR.

Complete the timing diagram for the given instruction sequence in the style of lecture 6.

I1	ADD	R1, R2, R3
I2	ADD	R1, R1, R4
I3	LW	R2, 0(R1)
I4	ADD	R2, R2, R5
I5	SW	R2, 0(R1)

Stall1 (above t5), stall2 (above t6)

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
Fetch	I1	I2	I3	I4	I5	NOP	I5			
Decode		I1	I2	I3	I4	I4	NOP	I5		
Execute			I1	I2	I3	NOP	I4	NOP	I5	
Memory				I1	I2	I3	NOP	I4	NOP	I5
Writeback					I1	I2	I3	NOP	I4	NOP

✓ 8

Part D (5 points)

Design the stall logic for the stall2 signal.

$Stall_2 = OPCODE_M = LW$
$+ OPCODE_M = SW$ ✓ 5

13