CSE 30321 – Computer Architecture I – Fall 2009
Homework 06 – Control Logic for Multi Cycle Processors
Assigned: October 29, 2009 **– Due:** November 5, 2009
This assignment can be done in groups of 1, 2, or 3. The assignment is worth 30 points.

Problem 1: (15 points)

Background:
For your final project, you'll need to add some instructions/additional functionality to the 6-instruction processor that you have worked with in the lab assignments. In this question, you'll do something similar – but for the MIPS ISA. Note: when you decide to add or support a new instruction, this will most often require more hardware (i.e. multiplexors, wires, etc.) as well as more control signals. Ideally, you'd like an instruction to improve execution time (by lowering CPI, clock rate, etc.) with as little additional hardware and control signal overhead as possible.

Part A:
We wish to add the instruction lui (load upper immediate) to the MIPS multi-cycle datapath. For the lui instruction, the immediate value is shifted left 16 bits and stored in a register. The lower 16 bits are zeroes.

- The instruction syntax is: lui $t, imm
- Thus, $t ← (imm << 16)
- The instruction encoding is: 0011 – 11xx – xxxt – tttt – iiii – iiii – iiii – iiii
    - (where 0 is the most significant bit and i is the least significant bit)

For the MIPS multi-cycle datapath, discuss/show the necessary modifications to the datapath and finite state machine. Note that you **do not** have to update all other states in the finite state machine if you add new control signals. (See the end of this handout for schematics of both the datapath and finite state machine.) The instruction should take 4 clock cycles.

Part B:
Now, modify the datapath so that the lui instruction takes just 3 clock cycles. Again, add any necessary datapaths and control signals to the multi-cycle datapath. You have to maintain the assumption that you don't know what the instruction is before the end of state 1 (the end of the second cycle). Show any necessary changes to the finite state machine. Note that you **do not** have to update all other states in the finite state machine if you add new control signals.

Problem 2: (15 points)

In Lecture 08, we did an in-class example where you were asked to implement a branch-if-less-than instruction – e.g. blt $s1, $s2, Label. There is no explicit branch-if-less-than or branch-if-greater than instruction in MIPS, and two separate instructions were needed to implement branch-if-less-than functionality.

The solution presented in class was to use the "set-on-less-than" instruction:

```
slt $t0, $s1, $s2       # $t0 ← 0 if $s1 >= $s2
                        # $t0 ← 1 if $s1 < $s2
```

Then, branch-if-less-than functionality could be implemented as follows:

```
slt $1, $s1, $s2
bne $1, $0, label
```

In this question, you are asked to explain how the MIPS datapath and finite state machine would need to be modified in order to implement an explicit blt (or bgt) instruction. Whether or not the data in one register is less than or greater than the data in another will be determined by examining the most significant bit of the ALUOut register after a comparison calculation is performed. (Thus, if the most significant bit of ALUOut is a 1, the result of the comparison is negative. If the bit is a 0, the result of the comparison is positive.)

Note – explicit modifications of the datapath and finite state machine diagrams are not required for this problem (although you can use them if you would like). However, please explain why you made the design decisions that you made – and your rationale for making them.

Bonus: (10 points, must be done individually)

Background:
The purpose of this question is to quantify performance improvements to the MIPS multi-cycle datapath to determine which design improvement makes the most sense from the standpoint of execution time.

Question:
Two important parameters control the performance of a processor: cycle time and cycles per instruction. There is an enduring trade-off between these two parameters in the design process of microprocessors. While some designers prefer to increase the processor frequency at the expense of large CPI, other designers follow a different school of thought in which reducing the CPI comes at the expense of lower clock frequencies.

Consider the following machines, and compare their performance using the SPEC CPUint 2000 data (for your convenience summarized below):

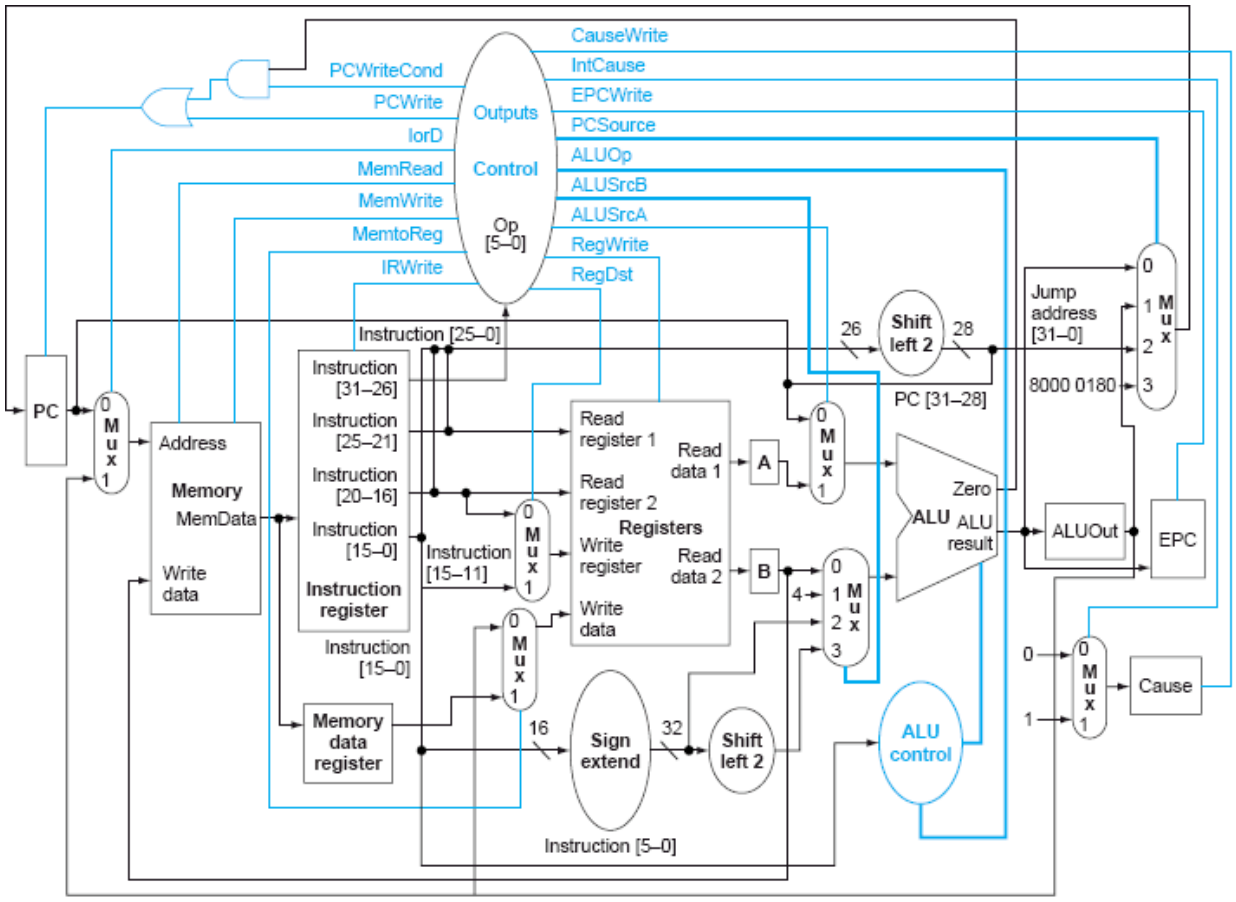
ALU	48%
Store	10%
Load	27%
Branch	15%

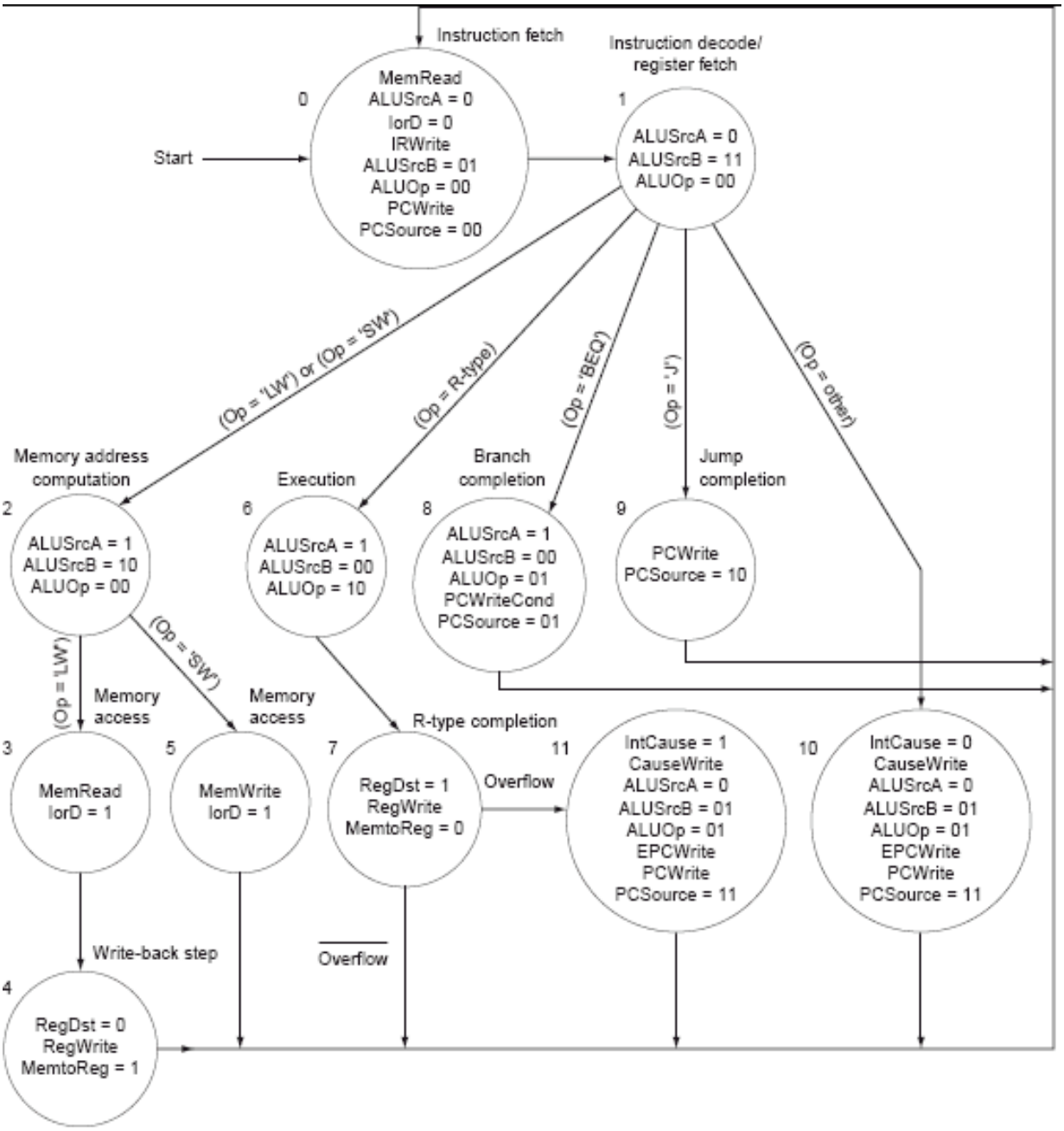
M1: The multicycle datapath that we have worked with in class (see handout) with a 4 GHz clock.

M2: A machine like the multicycle datapath of Problem 1, except that register updates are done in the same clock cycle as a memory read or ALU operation. (Thus, in the FSM of Problem 1, states 6 and 7 and states 3 and 4 are combined.) This machine has a 3.2 GHz clock, since the register update increases the length of the critical path.

M3: A machine like M2 except that effective address calculations are done in the same clock cycle as a memory access. Thus, states 2, 3 and 4 can be combined, as can 2 and 5, as well as 6 and 7. This machine has a 2.8 GHz clock because of the long cycle created by combining address calculation and memory access.

Find out which of the machines is the fastest. Are there instructions mixes that would make another machine faster, and if so, what are they? (One example is sufficient.)


CauseWrite
IntCause
EPCWrite
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op
[5–0]
Instruction [25–0]
26
Shift left 2
28
PC [31–28]
Jump address [31–0]
8000 0180
PC
Address
Memory
MemData
Write data
Instruction [31–26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–0]
Instruction register
Instruction [15–11]
Instruction [15–0]
Memory data register
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 1
A
B
4
Zero
ALU
ALU result
ALUOut
EPC
Cause
16
Sign extend
32
Shift left 2
ALU control
Instruction [5–0]
Mux


Instruction fetch
Instruction decode/
register fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Start
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
(Op = other)
Memory address
computation
Execution
Branch
completion
Jump
completion
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory
access
Memory
access
R-type completion
3
MemRead
IorD = 1
5
MemWrite
IorD = 1
7
RegDst = 1
RegWrite
MemtoReg = 0
Overflow
11
IntCause = 1
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11
10
IntCause = 0
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11
Write-back step
Overflow
4
RegDst = 0
RegWrite
MemtoReg = 1