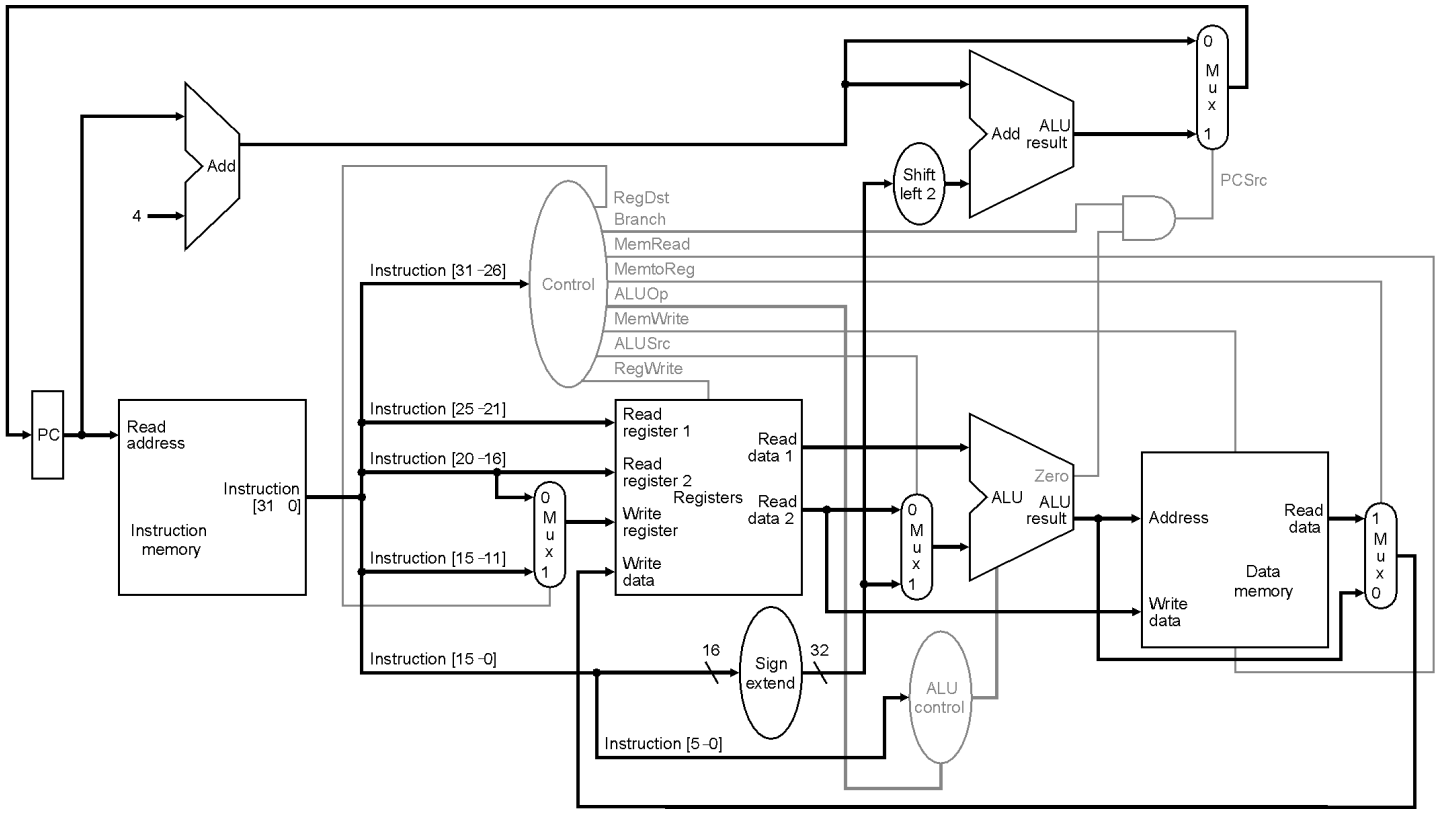
COMP3211/9211 : Computer Architecture
Tutorial 3 : Week 6/7
Solutions

Compiled by Marco Della Torre : marcodt@cse : August 30, 2004

(P = Patterson and Hennessy : Computer Organization and Design, 2nd Edition)

Warmup Exercises

Q02. [P5.1] Describe the effect that a single stuck-at-0 fault (i.e., regardless of what it should be, the signal is always 0) would have on the multiplexors in the single-cycle datapath in Fig 5.19 on p.360 [similar to Slide 31 of Lecture L08 on Friday Week 5]. Which instructions [of those we based our design upon], if any, would still work? Consider each of the following faults separately: RegDst = 0, ALUSrc = 0,MemtoReg = 0, Zero [Equal] = 0.



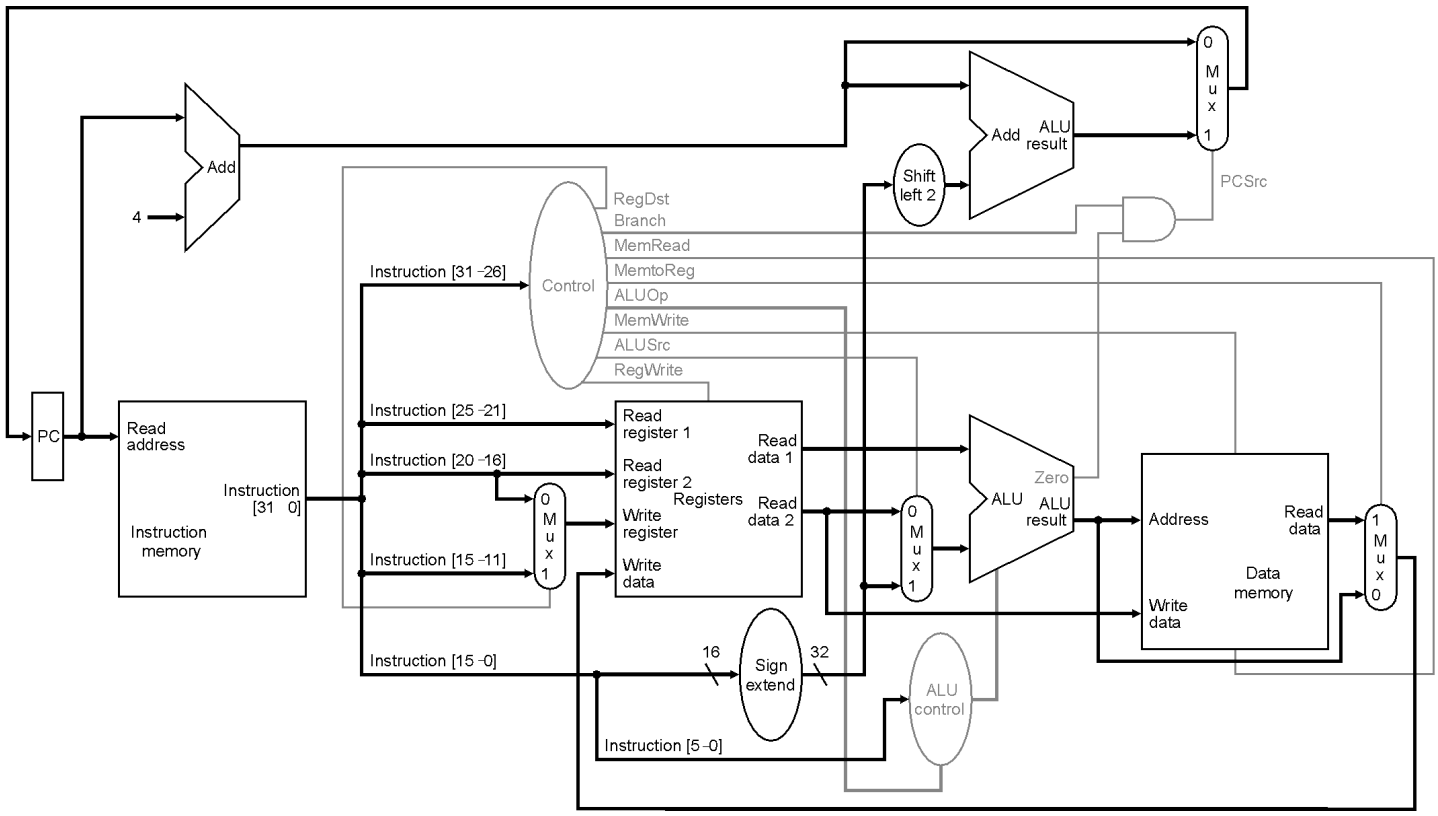
RegDst controls the multiplexor which determines which register is written when a result is to be stored. Therefore, if RegDst = 0, all R-format instructions will not work properly because the wrong destination register will be specified.

ALUSrc is used to select between data from a register and a sign-extended version of bits 15-0. Therefore, if ALUSrc = 0, then all I-format instructions except branch will not work because we will not be able to get the sign-extended 16 bits into the ALU.

MemtoReg controls the source of the data written back to a register. If MemtoReg = 0, then loads will not work because the memory cannot be selected as a source for the data.

If Branch and Zero are both high, then a branch instruction is being executed and should be taken (PCSrc = 1).
If Zero is always 0, a branch instruction will never be taken, since PCSrc will be set to 0 and therefore the PC is simply incremented by 4, regardless of the instruction.

Q03. [P5.5] We wish to add the instruction addi (add immediate) to the single-cycle datapath described in this chapter (P5). Add any necessary datapaths and control signals to the single-cycle datapath of Figure 5.19 on page 361 (copy below).



No additional datapaths or control signals are required to implement the addi instruction.

The original Figure 5.20 and the new additions are shown below.

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

FIGURE 5.20 The setting of the control lines is completely determined by the opcode fields of the instruction. The first row of the table corresponds to the R-format instructions (add, sub, and, or, and slt). For all these instructions, the source register fields are rs and rt and the destination register field is rd; this defines how the signals ALUSrc and RegDst are set. Furthermore, an R-type instruction writes a register (RegWrite = 1), but neither reads nor writes data memory. When the Branch control signal is 0, the PC is unconditionally replaced with PC + 4; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high. The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct field. The second and third rows of this table give the control signal settings for lw and sw. These ALUSrc and ALUOp fields are set to perform the address calculation. The MemRead and MemWrite are set to perform the memory access. Finally, RegDst and RegWrite are set for a load to cause the result to be stored into the rt register. The branch instruction is similar to an R-format operation, since it sends the rs and rt registers to the ALU. The ALUOp field for branch is set for a subtract (ALU control = 01), which is used to test for equality. Notice that the MemtoReg field is irrelevant when the RegWrite signal is 0—since the register is not being written, the value of the data on the register data write port is not used. Thus, the entry MemtoReg in the last two rows of the table is replaced with X for don't care. Don't cares can also be added to RegDst when RegWrite is 0. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

The following row is now added

Instruction	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
addi	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**

Discussion Questions

Q04. During Lecture L08, on Friday of Week 5, the control for the JUMP instruction was discussed. Modify the datapath in order to implement this instruction.

Answer is covered in P&H, Figure 5.29, page 372

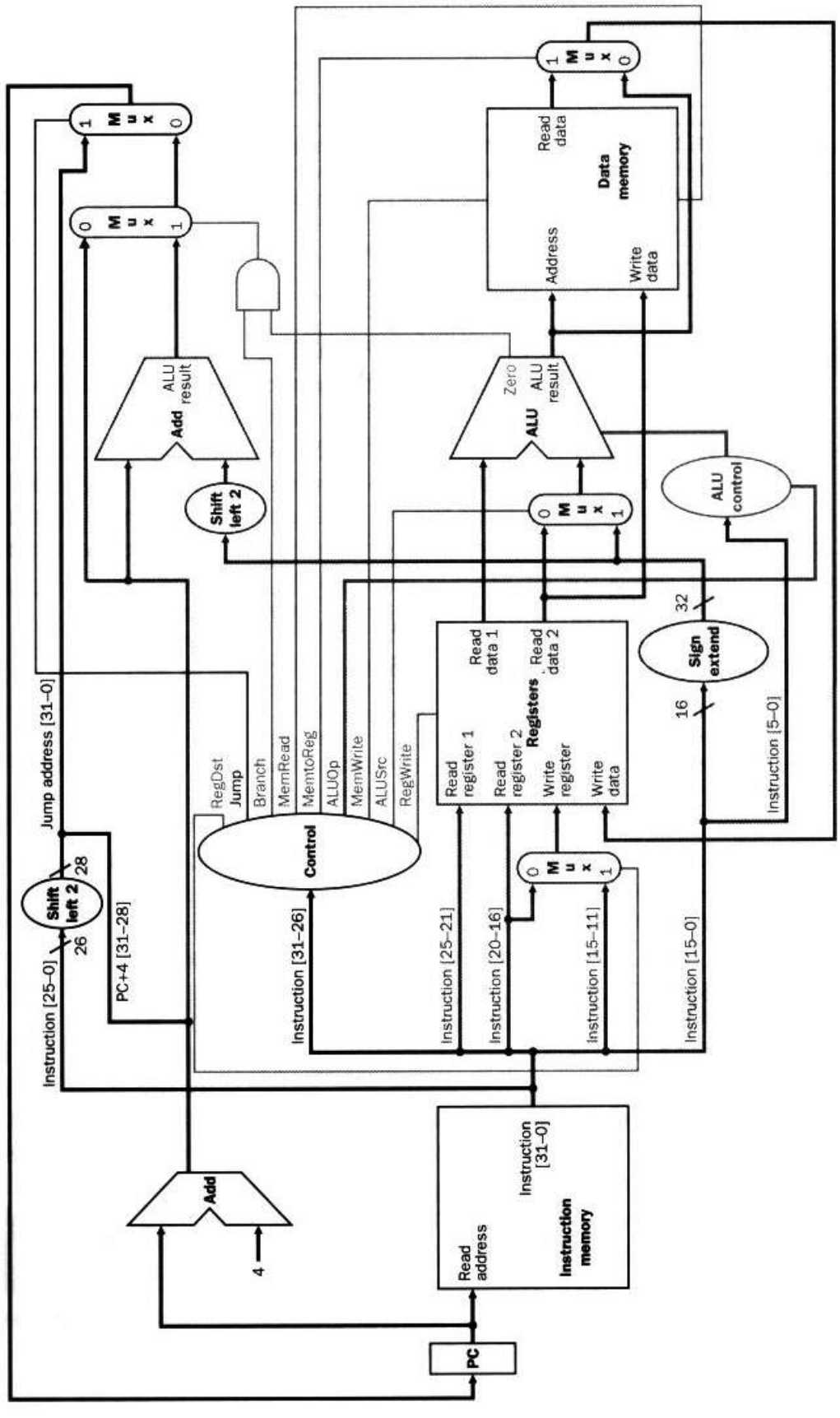


FIGURE 5.29 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address.

Q05. [P5.12] Consider the following idea: Let's modify the instruction set architecture and remove the ability to specify an offset for memory access instructions. Specifically, all load-store instructions with non-zero offsets would become pseudo-instructions and would be implemented using two instructions. For example:

```
addi   $at, $t1, 104        # add the offset to a temporary
lw     $t0, $at             # new way of doing lw $t0, 104 ($t1)
```

What changes would you make to the single-cycle datapath and control if this simplified architecture were to be used?

The key is recognizing that we no longer have to go through the ALU and then to memory. We would not want to add zero using the ALU, instead we want to provide a path directly from the Read data 1 output of the register file to the read/write address lines of the memory (assuming the instruction format does not change). The output of the ALU would no longer connect to memory. The control does not need to change, but some of the control signals are now "don't cares". Assuming we are not implementing addi or addiu, it is possible to remove AluSrc and the multiplexor it controls by having just the data from Read data 2 going into the ALU. This results in additional optimizations to ALU control.

Q06. The efficiency of a single-cycle datapath was questioned during lectures.
On Slide 12 of Lecture L08 the timing for various datapath components was given. Let us assume the clock period is set to 11.5ns (the critical path length for a load instruction). If loads constitute 24% of all instructions, 12% are stores, 44% R-format instructions, and 20% are branches, compute the fraction of time a single cycle datapath spends fetching and executing instructions. That is, what fraction of time is the datapath **NOT** simply waiting for the next triggering clock edge?

From the lecture notes:

PC: t_{setup} = 0.5ns	**Mem: t_{setup} = 1.0ns**
PC: t_{pd} = 0.5ns	**Mem: t_{access} = 2.0ns**
RegFile: t_{setup} = 1.0ns	**Add/Sub: t_{comb} = 4.0ns**
RegFile: t_{access} = 2.0ns	**OR: t_{comb} = 0.5ns**

Instructions result in the delays outlined below.

load: $t_{pd} + t_{access(instr\ mem)} + t_{access(reg)} + t_{comb(add/sub)} + t_{access(data\ mem)} + t_{setup(reg)}$ = 11.5ns
store: $t_{pd} + t_{access(instr\ mem)} + t_{access(reg)} + t_{comb(add/sub)} + t_{setup(data\ mem)}$ = 9.5ns
add: $t_{pd} + t_{access(instr\ mem)} + t_{access(reg)} + t_{comb(add/sub)} + t_{setup(reg)}$ = 9.5ns
beq: $t_{pd} + t_{access(instr\ mem)} + t_{access(reg)} + t_{comb(add/sub)} + t_{setup(PC)}$ = 9.0ns

The clock cycle time is 11.5ns, due to the load instruction.
The expected waiting time can be calculated using the given instruction frequencies.

```
load:        0ns   * 24%
store:       2ns   * 12%
R-Format:    2ns   * 44%
branches:    2.5ns * 20%
```

Expected wait = 1.62 ns/cycle (14.1% of the total execution time is spent waiting for the next clock edge)
That is, the datapath spends on average 9.88ns/cycle fetching and executing instructions (85.9% of the total time)

Q07. [P5.11] Determine whether any of the control signals (other than MemToReg) in the single-cycle implementation can be eliminated and replaced by another control signal. Why or Why not?

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Using Figure 5.20 (above), we discover that MemtoReg could be replaced by ALUSrc, RegDst could be replaced by ALUOp1, and either Branch or ALUOp0 could be replaced in favor of the other (their signals are identical). Note that in reality there would likely be additional rows present in the truth table to support other instructions, and it is quite likely that no control signals could be eliminated.

Follow-up Questions

Q08. Read P&H pp. 373 - 375 on performance of single cycle machines.

[P5.13] If the modifications described in Q05 are implemented, there are some definite trade-offs with regard to performance. Specifically, the cycle time may be affected, and all load-store instructions with nonzero offsets would now require an extra addi instruction (a good compiler might find ways to reduce the need for extra addi instructions, but you can ignore this). If there are too many load-store instructions with nonzero offsets, it is likely that the modification would not improve performance. Assuming delays as specified on page 373, what is the highest percentage of load-store instructions with offsets that could be tolerated (i.e., that would still result in the modification having a positive impact on performance)?

Given delays (on page 373) are:

Memory Units:	2ns
ALU and Adders:	2ns
Register File:	1ns

Base on these delays, the overall execution time for a load can be calculated.

load:	Instruction	
Memory Read	(2ns) +	
	Register Read	(1ns) +
	ALU	(2ns) +
	Data Memory Read	(2ns) +
	Register Write	(1ns)
	Total	8ns

After the modifications, a load instruction does not need to access the ALU, so

$load_{mod}$:	Instruction Memory Read	(2ns) +
	Register Read	(1ns) +
	Data Memory Read	(2ns) +
	Register Write	(1ns)
	Total	6ns

Therefore the load instruction completes 33% faster with the modifications. The cycle time can also reduced from 8ns to 6ns.

However, since an addi instruction must also be added, each load is accompanied by an additional delay of:

addi:	Instruction Memory Read	(2ns) +
	Register Read	(1ns) +
	ALU	(2ns) +
	Register Write	(1ns)
	Total	6ns

Consider executing 10 instructions, one of which is a load with an offset.
Previously this would have taken 10*8 = 80 ns.
Now with the additional addi instruction used to compute the offset, it takes (10 + 1)*6 = 66 ns.
So clearly we can tolerate more than 10%.
Algebraically,

$8*i = 6*(t + 1)*i$

where i is the number of instructions and t is the tolerance we are solving for.
Thus, for the above example t = 0.33, i.e. we can tolerate 33% of the instructions being loads and stores with offsets.
This shouldn't be surprising, because we sped up the cycle time by a factor of 8/6 = 1.33.