addressing modes that require multiple memory accesses substantially complicate pipeline control and make it difficult to keep the pipeline flowing smoothly.

Perhaps the best example is the DEC Alpha and the DEC NVAX. In comparable technology, the newer instruction set architecture of the Alpha allowed an implementation whose performance is more than twice as fast as NVAX. In another example, Bhandarkar and Clark [1991] compared the MIPS M/2000 and the DEC VAX 8700 by counting clock cycles of the SPEC benchmarks; they concluded that although the MIPS M/2000 executes more instructions, the VAX on average executes 2.7 times as many clock cycles, so the MIPS is faster.

Nine-tenths of wisdom consists of being wise in time.

American proverb

4.14 Concluding Remarks

As we have seen in this chapter, both the datapath and control for a processor can be designed starting with the instruction set architecture and an understanding of the basic characteristics of the technology. In Section 4.3, we saw how the datapath for a MIPS processor could be constructed based on the architecture and the decision to build a single-cycle implementation. Of course, the underlying technology also affects many design decisions by dictating what components can be used in the datapath, as well as whether a single-cycle implementation even makes sense.

instruction latency The inherent execution time for an instruction.

Pipelining improves throughput but not the inherent execution time, or **instruction latency**, of instructions; for some instructions, the latency is similar in length to the single-cycle approach. Multiple instruction issue adds additional datapath hardware to allow multiple instructions to begin every clock cycle, but at an increase in effective latency. Pipelining was presented as reducing the clock cycle time of the simple single-cycle datapath. Multiple instruction issue, in comparison, clearly focuses on reducing clock cycles per instruction (CPI).

Pipelining and multiple issue both attempt to exploit instruction-level parallelism. The presence of data and control dependences, which can become hazards, are the primary limitations on how much parallelism can be exploited. Scheduling and speculation, both in hardware and in software, are the primary techniques used to reduce the performance impact of dependences.

The switch to longer pipelines, multiple instruction issue, and dynamic scheduling in the mid-1990s has helped sustain the 60% per year processor performance increase that started in the early 1980s. As mentioned in Chapter 1, these microprocessors preserved the sequential programming model, but they eventually ran into the power wall. Thus, the industry has been forced to try multiprocessors, which exploit parallelism at much coarser levels (the subject of Chapter 7). This trend has also caused designers to reassess the power-performance implications

of some of the inventions since the mid-1990s, resulting in a simplification of pipelines in the more recent versions of microarchitectures.

To sustain the advances in processing performance via parallel processors, Amdahl's law suggests that another part of the system will become the bottleneck. That bottleneck is the topic of the next chapter: the memory system.

4.15 Historical Perspective and Further Reading

This section, which appears on the CD, discusses the history of the first pipelined processors, the earliest superscalars, and the development of out-of-order and speculative techniques, as well as important developments in the accompanying compiler technology.

4.16 Exercises

Contributed by Milos Prvulovic of Georgia Tech

Exercise 4.1

Different instructions utilize different hardware blocks in the basic single-cycle implementation. The next three problems in this exercise refer to the following instruction:

	Instruction	Interpretation
a.	`add Rd,Rs,Rt`	`Reg[Rd]=Reg[Rs]+Reg[Rt]`
b.	`lw Rt,Offs(Rs)`	`Reg[Rt]=Mem[Reg[Rs]+Offs]`

4.1.1 [5] <4.1> What are the values of control signals generated by the control in Figure 4.2 for this instruction?

4.1.2 [5] <4.1> Which resources (blocks) perform a useful function for this instruction?

4.1.3 [10] <4.1> Which resources (blocks) produce outputs, but their outputs are not used for this instruction? Which resources produce no outputs for this instruction?

Different execution units and blocks of digital logic have different latencies (time needed to do their work). In Figure 4.2 there are seven kinds of major blocks. Latencies of blocks along the critical (longest-latency) path for an instruction determine the minimum latency of that instruction. For the remaining three problems in this exercise, assume the following resource latencies:

	I-Mem	Add	Mux	ALU	Regs	D-Mem	Control
a.	400ps	100ps	30ps	120ps	200ps	350ps	100ps
b.	500ps	150ps	100ps	180ps	220ps	1000ps	65ps

4.1.4 [5] <4.1> What is the critical path for a MIPS AND instruction?

4.1.5 [5] <4.1> What is the critical path for a MIPS load (LD) instruction?

4.1.6 [10] <4.1> What is the critical path for a MIPS BEQ instruction?

Exercise 4.2

The basic single-cycle MIPS implementation in Figure 4.2 can only implement some instructions. New instructions can be added to an existing ISA, but the decision whether or not to do that depends, among other things, on the cost and complexity such an addition introduces into the processor datapath and control. The first three problems in this exercise refer to this new instruction:

	Instruction	Interpretation
a.	`add3 Rd,Rs,Rt,Rx`	`Reg[Rd]=Reg[Rs]+Reg[Rt]+Reg[Rx]`
b.	`sll Rt,Rd,Shift`	`Reg[Rd]= Reg[Rt] << Shift` (shift left by Shift bits)

4.2.1 [10] <4.1> Which existing blocks (if any) can be used for this instruction?

4.2.2 [10] <4.1> Which new functional blocks (if any) do we need for this instruction?

4.2.3 [10] <4.1> What new signals do we need (if any) from the control unit to support this instruction?

When processor designers consider a possible improvement to the processor datapath, the decision usually depends on the cost/performance tradeoff. In the following three problems, assume that we are starting with a datapath from Figure 4.2, where I-Mem, Add, Mux, ALU, Regs, D-Mem, and Control blocks have latencies of 400ps, 100ps, 30ps, 120ps, 200ps, 350ps, and 100ps, respectively, and costs of 1000, 30, 10, 100, 200, 2000, and 500, respectively. The remaining three problems in this exercise refer to the following processor improvement:

	Improvement	Latency	Cost	Benefit
a.	Faster Add	–20ps for Add units	+20 per Add unit	Replaces existing Add units with faster ones.
b.	Larger Registers	+100ps for Regs	+200 for Regs	Fewer loads and stores needed to save and restore register values. This results in 5% fewer instructions.

4.2.4 [10] <4.1> What is the clock cycle time with and without this improvement?

4.2.5 [10] <4.1> What is the speed-up achieved by adding this improvement?

4.2.6 [10] <4.1> Compare the cost/performance ratio with and without this improvement.

Exercise 4.3

Problems in this exercise refer to the following logic block:

	Logic Block
a.	Small I-Memory with four 8-bit words
b.	Small Registers unit with two 8-bit registers

4.3.1 [5] <4.1, 4.2> Does this block contain logic only, flip-flops only, or both?

4.3.2 [20] <4.1, 4.2> Show how this block can be implemented. Use only AND, OR, NOT, and D-elements.

4.3.3 [10] <4.1, 4.2> Repeat Exercise 4.3.2, but the AND and OR gates you use must all be 2-input gates.

Cost and latency of digital logic depends on the kinds of basic logic elements (gates) that are available and on the properties of these gates. The remaining three problems in this exercise refer to these gates, latencies, and costs:

	NOT		2-input AND or OR		Each additional input for AND/OR		D-element	
	Latency	Cost	Latency	Cost	Latency	Cost	Latency	Cost
a.	20ps	1	30ps	2	+0ps	+1	40ps	6
b.	50ps	1	100ps	2	+40ps	+1	160ps	2

4.3.4 [5] <4.1, 4.2> What is the latency of your implementation from Exercise 4.3.2?

4.3.5 [5] <4.1, 4.2> What is the cost of your implementation from Exercise 4.3.2?

4.3.6 [20] <4.1, 4.2> Change your design to minimize the latency, then to minimize the cost. Compare the cost and latency of these two optimized designs.

Exercise 4.4

When implementing a logic expression in digital logic, one must use the available logic gates to implement an operator for which a gate is not available. Problems in this exercise refer to the following logic expressions:

	Control signal 1	Control signal 2
a.	(((A OR B) OR C) OR (A AND C)) OR (A AND B)	(A OR B) OR C
b.	(((A OR B) XOR B) OR (A OR C)) OR (A AND B)	A AND B

4.4.1 [5] <4.2> Implement the logic for the Control signal 1. Your circuit should directly implement the given expression (do not reorganize the expression to "optimize" it), using NOT gates and 2-input AND, OR, and XOR gates.

4.4.2 [10] Assuming that all gates have equal latencies, what is the length (in gates) of the critical path in your circuit from Exercise 4.4.1?

4.4.3 [10] <4.2> When multiple logic expressions are implemented, it is possible to reduce implementation cost by using the some signals in more than one expression. Repeat Exercise 4.4.1, but implement both Control signal 1 and Control signal 2, and try to "share" circuitry between expressions whenever possible.

For the remaining three problems in this exercise, we assume that the following basic digital logic elements are available, and that their latency and cost are as follows:

	NOT		2-input AND		2-input OR		2-input XOR	
	Latency	Cost	Latency	Cost	Latency	Cost	Latency	Cost
a.	20ps	1	30ps	2	34ps	3	40ps	6
b.	50ps	1	100ps	2	120ps	2	150ps	2

4.4.4 [10] <4.2> What is the length of the critical path in your circuit from 4.4.3?

4.4.5 [10] <4.2> What is the cost of your circuit from Exercise 4.4.3?

4.4.6 [10] <4.2> What fraction of the cost was saved in your circuit from Exercise 4.4.3 by implementing these two control signals together instead of separately?

Exercise 4.5

The goal of this exercise is to help you familiarize yourself with the design and operation of sequential logical circuits. Problems in this exercise refer to this ALU operation:

	ALU operation
a.	Add-one (X+1)
b.	Shift left by 2 bits (X<<2)

4.5.1 [20] <4.2> Design a circuit with 1-bit data inputs and a 1-bit data output that accomplishes this operation serially, starting with the least-significant bit. In a serial implementation, the circuit is processing input operands bit by bit, generating output bits one by one. For example, a serial AND circuit is simply an AND gate; in cycle N we give it the Nth bit from each of the operand and we get the Nth bit of the result. In addition to data inputs, the circuit has a Clk (clock) input and a "Start" input that is set to 1 only in the very first cycle of the operation. In your design, you can use D-elements and NOT, AND, OR, and XOR gates.

4.5.2 [20] <4.2> Repeat Exercise 4.5.1, but now design a circuit that accomplishes this operation 2 bits at a time.

In the rest of this exercise, we assume that the following basic digital logic elements are available, and that their latency and cost are as follows:

	NOT		AND		OR		XOR		D-element	
	Latency	Cost	Latency	Cost	Latency	Cost	Latency	Cost	Latency	Cost
a.	20ps	1	30ps	2	20ps	2	30ps	4	40ps	6
b.	40ps	1	50ps	2	60ps	2	80ps	3	80ps	12

The time given for a D-element is its setup time. The data input of a flip-flop must have the correct value one setup-time before the clock edge (end of clock cycle) that stores that value into the flip-flop.

4.5.3 [10] <4.2> What is the cycle time for the circuit you designed in Exercise 4.5.1? How long does it take to perform the 32-bit operation?

4.5.4 [10] <4.2> What is the cycle time for the circuit you designed in Exercise 4.5.2? What is the speed-up achieved by using this circuit instead of the one from Exercise 4.5.1 for a 32-bit operation?

4.5.5 [10] <4.2> Compute the cost for the circuit you designed in Exercise 4.5.1, and then for the circuit you designed in Exercise 4.5.2.

4.5.6 [5] <4.2> Compare cost/performance ratios for the two circuits you designed in Exercises 4.5.1 and 4.5.2. For this problem, performance of a circuit is the inverse of the time needed to perform a 32-bit operation.

Exercise 4.6

Problems in this exercise assume that logic blocks needed to implement a processor's datapath have the following latencies:

	I-Mem	Add	Mux	ALU	Regs	D-Mem	Sign-extend	Shift-left-2
a.	400ps	100ps	30ps	120ps	200ps	350ps	20ps	2ps
b.	500ps	150ps	100ps	180ps	220ps	1000ps	90ps	20ps

4.6.1 [10] <4.3> If the only thing we need to do in a processor is fetch consecutive instructions (Figure 4.6), what would the cycle time be?

4.6.2 [10] <4.3> Consider a datapath similar to the one in Figure 4.11, but for a processor that only has one type of instruction: unconditional PC-relative branch. What would the cycle time be for this datapath?

4.6.3 [10] <4.3> Repeat Exercise 4.6.2, but this time we need to support only *conditional* PC-relative branches.

The remaining three problems in this exercise refer to the following logic block (resource) in the datapath:

	Resource
a.	Add 4 (to the PC)
b.	Data Memory

4.6.4 [10] <4.3> Which kinds of instructions require this resource?

4.6.5 [20] <4.3> For which kinds of instructions (if any) is this resource on the critical path?

4.6.6 [10] <4.3> Assuming that we only support `beq` and `add` instructions, discuss how changes in the given latency of this resource affect the cycle time of the processor. Assume that the latencies of other resources do not change.

Exercise 4.7

In this exercise we examine how latencies of individual components of the datapath affect the clock cycle time of the entire datapath, and how these components are utilized by instructions. For problems in this exercise, assume the following latencies for logic blocks in the datapath:

	I-Mem	Add	Mux	ALU	Regs	D-Mem	Sign-extend	Shift-left-2
a.	400ps	100ps	30ps	120ps	200ps	350ps	20ps	0ps
b.	500ps	150ps	100ps	180ps	220ps	1000ps	90ps	20ps

4.7.1 [10] <4.3> What is the clock cycle time if the only type of instructions we need to support are ALU instructions (`add`, `and`, etc.)?

4.7.2 [10] <4.3> What is the clock cycle time if we only had to support `lw` instructions?

4.7.3 [20] <4.3> What is the clock cycle time if we must support `add`, `beq`, `lw`, and `sw` instructions?

For the remaining problems in this exercise, assume that there are no pipeline stalls and that the breakdown of executed instructions is as follows:

	add	addi	not	beq	lw	sw
a.	30%	15%	5%	20%	20%	10%
b.	25%	5%	5%	15%	35%	15%

4.7.4 [10] <4.3> In what fraction of all cycles is the data memory used?

4.7.5 [10] <4.3> In what fraction of all cycles is the input of the sign-extend circuit needed? What is this circuit doing in cycles in which its input is not needed?

4.7.6 [10] <4.3> If we can improve the latency of one of the given datapath components by 10%, which component should it be? What is the speed-up from this improvement?

Exercise 4.8

When silicon chips are fabricated, defects in materials (e.g., silicon) and manufacturing errors can result in defective circuits. A very common defect is for one wire to affect the signal in another. This is called a cross-talk fault. A special

class of cross-talk faults is when a signal is connected to a wire that has a constant logical value (e.g., a power supply wire). In this case we have a stuck-at-0 or a stuck-at-1 fault, and the affected signal always has a logical value of 0 or 1, respectively.

The following problems refer to the following signal from Figure 4.24:

	Signal
a.	Instruction Memory, output Instruction, bit 7
b.	Control unit, output MemtoReg

4.8.1 [10] <4.3, 4.4> Let us assume that processor testing is done by filling the PC, registers, and data and instruction memories with some values (you can choose which values), letting a single instruction execute, then reading the PC, memories, and registers. These values are then examined to determine if a particular fault is present. Can you design a test (values for PC, memories, and registers) that would determine if there is a stuck-at-0 fault on this signal?

4.8.2 [10] <4.3, 4.4> Repeat Exercise 4.8.1 for a stuck-at-1 fault. Can you use a single test for both stuck-at-0 and stuck-at-1? If yes, explain how; if no, explain why not.

4.8.3 [60] <4.3, 4.4> If we know that the processor has a stuck-at-1 fault on this signal, is the processor still usable? To be usable, we must be able to convert any program that executes on a normal MIPS processor into a program that works on this processor. You can assume that there is enough free instruction memory and data memory to let you make the program longer and store additional data. Hint: the processor is usable if every instruction "broken" by this fault can be replaced with a sequence of "working" instructions that achieve the same effect.

The following problems refer to the following fault:

	Fault
a.	Stuck-at-1
b.	Becomes 0 if Instruction [31-26] has all bits at 0, no fault otherwise

4.8.4 [10] <4.3, 4.4> Repeat Exercise 4.8.1, but now the fault to test for is whether the "MemRead" control signal has this fault.

4.8.5 [10] <4.3, 4.4> Repeat Exercise 4.8.1, but now the fault to test for is whether the "Jump" control signal has this fault.

4.8.6 [40] <4.3, 4.4> Using a single test described Exercise 4.8.1, we can test for faults in several different signals, but typically not all of them. Describe a series of tests to look for this fault in all Mux outputs (every output bit from each of the five Muxes)? Try to do this with as few single-instruction tests as possible.

Exercise 4.9

In this exercise we examine the operation of the single-cycle datapath for a particular instruction. Problems in this exercise refer to the following MIPS instruction:

	Instruction
a.	`lw $1,40($6)`
b.	`Label: bne $1,$2,Label`

4.9.1 [10] <4.4> What is the value of the instruction word?

4.9.2 [10] <4.4> What is the register number supplied to the register file's "Read register 1" input? Is this register actually read? How about "Read register 2"?

4.9.3 [10] <4.4> What is the register number supplied to the register file's "Write register" input? Is this register actually written?

Different instructions require different control signals to be asserted in the datapath. The remaining problems in this exercise refer to the following two control signals from Figure 4.24:

	Control signal 1	Control signal 2
a.	RegDst	MemRead
b.	RegWrite	MemRead

4.9.4 [20] <4.4> What is the value of these two signals for this instruction?

4.9.5 [20] <4.4> For the datapath from Figure 4.24, draw the logic diagram for the part of the control unit that implements just the first signal. Assume that we only need to support `lw`, `sw`, `beq`, `add`, and `j` (jump) instructions.

4.9.6 [20] <4.4> Repeat Exercise 4.9.5, but now implement both of these signals.

Exercise 4.10

In this exercise we examine how the clock cycle time of the processor affects the design of the control unit, and vice versa. Problems in this exercise assume that the logic blocks used to implement the datapath have the following latencies:

	I-Mem	Add	Mux	ALU	Regs	D-Mem	Sign-extend	Shift-left-2	ALU Ctrl
a.	400ps	100ps	30ps	120ps	200ps	350ps	20ps	0ps	50ps
b.	500ps	150ps	100ps	180ps	220ps	1000ps	90ps	20ps	55ps

4.10.1 [10] <4.2, 4.4> To avoid lengthening the critical path of the datapath shown in Figure 4.24, how much time can the control unit take to generate the MemWrite signal.

4.10.2 [20] <4.2, 4.4> Which control signal in Figure 4.24 has the most slack and how much time does the control unit have to generate it if it wants to avoid being on the critical path?

4.10.3 [20] <4.2, 4.4> Which control signal in Figure 4.24 is the most critical to generate quickly and how much time does the control unit have to generate it if it wants to avoid being on the critical path?

The remaining problems in this exercise assume that the time needed by the control unit to generate individual control signals is as follows:

	RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
a.	720ps	730ps	600ps	400ps	700ps	200ps	710ps	200ps	800ps
b.	1600ps	1600ps	1400ps	500ps	1400ps	400ps	1500ps	400ps	1700ps

4.10.4 [20] <4.4> What is the clock cycle time of the processor?

4.10.5 [20] <4.4> If you can speed up the generation of control signals, but the cost of the entire processor increases by $1 for each 5ps improvement of a single control signal, which control signals would you speed up and by how much to maximize performance? What is the cost (per processor) of this performance improvement?

4.10.6 [30] <4.4> If the processor is already too expensive, instead of paying to speed it up as we did in 4.10.5, we want to minimize its cost without further slowing it down. If you can use slower logic to implement control signals, saving $1 of the processor cost for each 5ps you add to the latency of a single control signal, which control signals would you slow down and by how much to reduce the processor's cost without slowing it down?

Exercise 4.11

In this exercise we examine in detail how an instruction is executed in a single-cycle datapath. Problems in this exercise refer to a clock cycle in which the processor fetches the following instruction word:

	Instruction word
a.	10001100010000110000000000010000
b.	00010000001000110000000000001100

4.11.1 [5] <4.4> What are the outputs of the sign-extend and the jump "Shift left 2" unit (in the upper left of Figure 4.24) for this instruction word?

4.11.2 [10] <4.4> What are the values of ALU control unit's inputs for this instruction?

4.11.3 [10] <4.4> What is the new PC address after this instruction is executed? Highlight the path through which this value is determined.

The remaining problems in this exercise assume that data memory is all-zeros and that the processor's registers have the following values at the beginning of the cycle in which the above instruction word is fetched:

	$0	$1	$2	$3	$4	$5	$6	$8	$12	$31
a.	0	1	2	3	–4	5	6	8	1	–32
b.	0	–16	–2	–3	4	–10	–6	–1	8	–4

4.11.4 [10] <4.4> For each Mux, show the values of its data output during the execution of this instruction and these register values.

4.11.5 [10] <4.4> For the ALU and the two add units, what are their data input values?

4.11.6 [10] <4.4> What are the values of all inputs for the "Registers" unit?

Exercise 4.12

In this exercise, we examine how pipelining affects the clock cycle time of the processor. Problems in this exercise assume that individual stages of the datapath have the following latencies:

	IF	ID	EX	MEM	WB
a.	300ps	400ps	350ps	500ps	100ps
b.	200ps	150ps	120ps	190ps	140ps

4.12.1 [5] <4.5> What is the clock cycle time in a pipelined and nonpipelined processor?

4.12.2 [10] <4.5> What is the total latency of a `lw` instruction in a pipelined and nonpipelined processor?

4.12.3 [10] <4.5> If we can split one stage of the pipelined datapath into two new stages, each with half the latency of the original stage, which stage would you split and what is the new clock cycle time of the processor?

The remaining problems in this exercise assume that instructions executed by the processor are broken down as follows:

	ALU	beq	lw	sw
a.	50%	25%	15%	10%
b.	30%	25%	30%	15%

4.12.4 [10] <4.5> Assuming there are no stalls or hazards, what is the utilization (% of cycles used) of the data memory?

4.12.5 [10] <4.5> Assuming there are no stalls or hazards, what is the utilization of the write-register port of the "Registers" unit?

4.12.6 [30] <4.5> Instead of a single-cycle organization, we can use a multi-cycle organization where each instruction takes multiple cycles but one instruction finishes before another is fetched. In this organization, an instruction only goes through stages it actually needs (e.g., ST only takes four cycles because it does not need the WB stage). Compare clock cycle times and execution times with single-cycle, multi-cycle, and pipelined organization.

Exercise 4.13

In this exercise, we examine how data dependences affect execution in the basic five-stage pipeline described in Section 4.5. Problems in this exercise refer to the following sequence of instructions:

	Instruction sequence

a.	`lw $1,40($6)`
	`add $6,$2,$2`
	`sw $6,50($1)`

b.	`lw $5,-16($5)`
	`sw $5,-16($5)`
	`add $5,$5,$5`

4.13.1 [10] <4.5> Indicate dependences and their type.

4.13.2 [10] <4.5> Assume there is no forwarding in this pipelined processor. Indicate hazards and add `nop` instructions to eliminate them.

4.13.3 [10] <4.5> Assume there is full forwarding. Indicate hazards and add `nop` instructions to eliminate them. The remaining problems in this exercise assume the following clock cycle times:

	Without forwarding	With full forwarding	With ALU-ALU forwarding only
a.	300ps	400ps	360ps
b.	200ps	250ps	220ps

4.13.4 [10] <4.5> What is the total execution time of this instruction sequence without forwarding and with full forwarding? What is the speed-up achieved by adding full forwarding to a pipeline that had no forwarding?

4.13.5 [10] <4.5> Add `nop` instructions to this code to eliminate hazards if there is ALU-ALU forwarding only (no forwarding from the MEM to the EX stage)?

4.13.6 [10] <4.5> What is the total execution time of this instruction sequence with only ALU-ALU forwarding? What is the speed-up over a no-forwarding pipeline?

Exercise 4.14

In this exercise, we examine how resource hazards, control hazards, and ISA design can affect pipelined execution. Problems in this exercise refer to the following fragment of MIPS code:

	Instruction sequence

a.	`lw $1,40($6)`
	`beq $2,$0,Label ; Assume $2 == $0`
	`sw $6,50($2)`
	`Label: add $2,$3,$4`
	`sw $3,50($4)`

b.	`lw $5,-16($5)`
	`sw $4,-16($4)`
	`lw $3,-20($4)`
	`beq $2,$0,Label ; Assume $2 != $0`
	`add $5,$1,$4`

4.14.1 [10] <4.5> For this problem, assume that all branches are perfectly predicted (this eliminates all control hazards) and that no delay slots are used. If

we only have one memory (for both instructions and data), there is a structural hazard every time we need to fetch an instruction in the same cycle in which another instruction accesses data. To guarantee forward progress, this hazard must always be resolved in favor of the instruction that accesses data. What is the total execution time of this instruction sequence in the five-stage pipeline that only has one memory? We have seen that data hazards can be eliminated by adding `nop`s to the code. Can you do the same with this structural hazard? Why?

4.14.2 [20] <4.5> For this problem, assume that all branches are perfectly predicted (this eliminates all control hazards) and that no delay slots are used. If we change load/store instructions to use a register (without an offset) as the address, these instructions no longer need to use the ALU. As a result, MEM and EX stages can be overlapped and the pipeline has only four stages. Change this code to accommodate this changed ISA. Assuming this change does not affect clock cycle time, what speed-up is achieved this instruction sequence?

4.14.3 [10] <4.5> Assuming stall-on-branch and no delay slots, what speed-up is achieved on this code if branch outcomes are determined in the ID stage, relative to the execution where branch outcomes are determined in the EX stage?

The remaining problems in this exercise assume that individual pipeline stages have the following latencies:

	IF	ID	EX	MEM	WB
a.	100ps	120ps	90ps	130ps	60ps
b.	180ps	100ps	170ps	220ps	60ps

4.14.4 [10] <4.5> Given these pipeline stage latencies, repeat the speed-up calculation from 4.14.2, but take into account the (possible) change in clock cycle time. When EX and MEM are done in a single stage, most of their work can be done in parallel. As a result, the resulting EX/MEM stage has a latency that is the larger of the original two, plus 20ps needed for the work that could not be done in parallel.

4.14.5 [10] <4.5> Given these pipeline stage latencies, repeat the speed-up calculation from Exercise 4.14.3, taking into account the (possible) change in clock cycle time. Assume that the latency ID stage increases by 50% and the latency of the EX stage decreases by 10ps when branch outcome resolution is moved from EX to ID.

4.14.6 [10] <4.5> Assuming stall-on-branch and no delay slots, what is the new clock cycle time and execution time of this instruction sequence if `beq` address

computation is moved to the MEM stage? What is the speed-up from this change? Assume that the latency of the EX stage is reduced by 20ps and the latency of the MEM stage is unchanged when branch outcome resolution is moved from EX to MEM.

Exercise 4.15

In this exercise, we examine how the ISA affects pipeline design. Problems in this exercise refer to the following new instruction:

a.	`bezi (Rs),Label`	if Mem[Rs] = 0 then PC=PC+Offs
b.	`swi Rd,Rs(Rt)`	Mem[Rs+Rt]=Rd

4.15.1 [20] <4.5> What must be changed in the pipelined datapath to add this instruction to the MIPS ISA?

4.15.2 [10] <4.5> Which new control signals must be added to your pipeline from Exercise 4.15.1?

4.15.3 [20] <4.5, 4.13> Does support for this instruction introduce any new hazards? Are stalls due to existing hazards made worse?

4.15.4 [10] <4.5, 4.13> Give an example of where this instruction might be useful and a sequence of existing MIPS instruction that are replaced by this instruction.

4.15.5 [10] <4.5, 4.11, 4.13> If this instruction already exists in a legacy ISA, explain how it would be executed in a modern processor like AMD Barcelona.

The last problem in this exercise assumes that each use of the new instruction replaces the given number of original instructions, that the replacement can be made once in the given number of original instructions, and that each time the new instruction is executed the given number of extra stall cycles is added to the program's execution time:

	Replaces	Once in every	Extra Stall Cycles
a.	2	20	1
b.	3	60	0

4.15.6 [10] <4.5> What is the speed-up achieved by adding this new instruction? In your calculation, assume that the CPI of the original program (without the new instruction) is 1.

Exercise 4.16

The first three problems in this exercise refer to the following MIPS instruction:

	Instruction
a.	`lw $1,40($6)`
b.	`add $5,$5,$5`

4.16.1 [5] <4.6> As this instruction executes, what is kept in each register located between two pipeline stages?

4.16.2 [5] <4.6> Which registers need to be read, and which registers are actually read?

4.16.3 [5] <4.6> What does this instruction do in EX and MEM stages?

The remaining three problems in this exercise refer to the following loop. Assume that perfect branch prediction is used (no stalls due to control hazards), that there are no delay slots, and that the pipeline has full forwarding support. Also assume that many iterations of this loop are executed before the loop exits.

	Loop

a.	`Loop: lw $1,40($6)`
	`add $5,$5,$8`
	`add $6,$6,$8`
	`sw $1,20($5)`
	`beq $1,$0,Loop`

b.	`Loop: add $1,$2,$3`
	`sw $0,0($1)`
	`sw $0,4($1)`
	`add $2,$2,$4`
	`beq $2,$0,Loop`

4.16.4 [10] <4.6> Show a pipeline execution diagram for the third iteration of this loop, from the cycle in which we fetch the first instruction of that iteration up to (but not including) the cycle in which we can fetch the first instruction of the next iteration. Show all instructions that are in the pipeline during these cycles (not just those from the third iteration).

4.16.5 [10] <4.6> How often (as a percentage of all cycles) do we have a cycle in which all five pipeline stages are doing useful work?

4.16.6 [10] <4.6> At the start of the cycle in which we fetch the first instruction of the third iteration of this loop, what is stored in the IF/ID register?

Exercise 4.17

Problems in this exercise assume that instructions executed by a pipelined processor are broken down as follows:

	add	beq	lw	sw
a.	50%	25%	15%	10%
b.	30%	15%	35%	20%

4.17.1 [5] <4.6> Assuming there are no stalls and that 60% of all conditional branches are taken, in what percentage of clock cycles does the branch adder in the EX stage generate a value that is actually used?

4.17.2 [5] <4.6> Assuming there are no stalls, how often (percentage of all cycles) do we actually need to use all three register ports (two reads and a write) in the same cycle?

4.17.3 [5] <4.6> Assuming there are no stalls, how often (percentage of all cycles) do we use the data memory?

Each pipeline stage in Figure 4.33 has some latency. Additionally, pipelining introduces registers between stages (Figure 4.35), and each of these adds an additional latency. The remaining problems in this exercise assume the following latencies for logic within each pipeline stage and for each register between two stages:

	IF	ID	EX	MEM	WB	Pipeline register
a.	100ps	120ps	90ps	130ps	60ps	10ps
b.	180ps	100ps	170ps	220ps	60ps	10ps

4.17.4 [5] <4.6> Assuming there are no stalls, what is the speed-up achieved by pipelining a single-cycle datapath?

4.17.5 [10] <4.6> We can convert all load/store instructions into register-based (no offset) and put the memory access in parallel with the ALU. What is the clock cycle time if this is done in the single-cycle and in the pipelined datapath? Assume that the latency of the new EX/MEM stage is equal to the longer of their latencies.

4.17.6 [10] <4.6> The change in Exercise 4.17.5 requires many existing `lw/sw` instructions to be converted into two-instruction sequences. If this is needed for 50% of these instructions, what is the overall speed-up achieved by changing from the five-stage pipeline to the four-stage pipeline where EX and MEM are done in parallel?

Exercise 4.18

The first three problems in this exercise refer to the execution of the following instruction in the pipelined datapath from Figure 4.51, and assume the following clock cycle time, ALU latency, and Mux latency:

	Instruction	Clock cycle time	ALU Latency	Mux Latency
a.	`add $1,$2,$3`	100ps	80ps	10ps
b.	`slt $2,$1,$3`	80ps	50ps	20ps

4.18.1 [10] <4.6> For each stage of the pipeline, what are the values of control signals asserted by this instruction in that pipeline stage?

4.18.2 [10] <4.6, 4.7> How much time does the control unit have to generate the ALUSrc control signal? Compare this to a single-cycle organization.

4.18.3 What is the value of the PCSrc signal for this instruction? This signal is generated early in the MEM stage (only a single AND gate). What would be a reason in favor of doing this in the EX stage? What is the reason against doing it in the EX stage?

The remaining problems in this exercise refer to the following signals from Figure 4.48:

	Signal 1	Signal 2
a.	RegDst	RegWrite
b.	MemRead	RegWrite

4.18.4 [5] <4.6> For each of these signals, identify the pipeline stage in which it is generated and the stage in which it is used.

4.18.5 [5] <4.6> For which MIPS instruction(s) are both of these signals set to 1?

4.18.6 [10] <4.6> One of these signals goes back through the pipeline. Which signal is it? Is this a time-travel paradox? Explain.

Exercise 4.19

This exercise is intended to help you understand the cost/complexity/performance tradeoffs of forwarding in a pipelined processor. Problems in this exercise refer to pipelined datapaths from Figure 4.45. These problems assume that, of all instructions executed in a processor, the following fraction of these instructions

has a particular type of RAW data dependence. The type of RAW data dependence is identified by the stage that produces the result (EX or MEM) and the instruction that consumes the result (1st instruction that follows the one that produces the result, 2nd instruction that follows, or both). We assume that the register write is done in the first half of the clock cycle and that register reads are done in the second half of the cycle, so "EX to 3rd" and "MEM to 2nd" dependences are not counted because they can not result in data hazards. Also, assume that the CPI of the processor is 1 if there are no data hazards.

	EX to 1st only	EX to 1st and 2nd	EX to 2nd only	MEM to 1st
a.	10%	10%	5%	25%
b.	15%	5%	10%	20%

4.19.1 [10] <4.7> If we use no forwarding, what fraction of cycles are we stalling due to data hazards?

4.19.2 [5] <4.7> If we use full forwarding (forward all results that can be forwarded), what fraction of cycles are we stalling due to data hazards?

4.19.3 [10] <4.7> Let us assume that we can not afford to have three-input Muxes that are needed for full forwarding. We have to decide if it is better to forward only from the EX/MEM pipeline register (next-cycle forwarding) or only from the MEM/WB pipeline register (two-cycle forwarding). Which of the two options results in fewer data stall cycles?

The remaining three problems in this exercise refer to the following latencies for individual pipeline stages. For the EX stage, latencies are given separately for a processor without forwarding and for a processor with different kinds of forwarding.

	IF	ID	EX (no FW)	EX (full FW)	EX (FW from EX/MEM only)	EX (FW from MEM/WB only)	MEM	WB
a.	100ps	50ps	75ps	110ps	100ps	100ps	100ps	60ps
b.	250ps	300ps	200ps	350ps	320ps	310ps	300ps	200ps

4.19.4 [10] <4.7> For the given hazard probabilities and pipeline stage latencies, what is the speed-up achieved by adding full forwarding to a pipeline that had no forwarding?

4.19.5 [10] <4.7> What would be the additional speed-up (relative to a processor with forwarding) if we added time-travel forwarding that eliminates all data

hazards? Assume that the yet-to-be-invented time-travel circuitry adds 100ps to the latency of the full-forwarding EX stage.

4.19.6 [20] <4.7> Repeat Exercise 4.19.3 but this time determine which of the two options results in shorter time per instruction.

Exercise 4.20

Problems in this exercise refer to the following instruction sequences:

	Instruction sequence

a.	`lw $1,40($2)`
	`add $2,$3,$3`
	`add $1,$1,$2`
	`sw $1,20($2)`

b.	`add $1,$2,$3`
	`sw $2,0($1)`
	`lw $1,4($2)`
	`add $2,$2,$1`

4.20.1 [5] <4.7> Find all data dependences in this instruction sequence.

4.20.2 [10] <4.7> Find all hazards in this instruction sequence for a five-stage pipeline with and then without forwarding.

4.20.3 [10] <4.7> To reduce clock cycle time, we are considering a split of the MEM stage into two stages. Repeat Exercise 4.20.2 for this six-stage pipeline.

The remaining three problems in this exercise assume that, before any of the above is executed, all values in data memory are 0s and that registers `$0` through `$3` have the following initial values:

	$0	$1	$2	$3
a.	0	1	31	1000
b.	0	–2	63	2500

4.20.4 [5] <4.7> Which value is the first one to be forwarded and what is the value it overrides?

4.20.5 [10] <4.7> If we assume forwarding will be implemented when we design the hazard detection unit, but then we forget to actually implement forwarding, what are the final register values after this instruction sequence?

4.20.6 [10] <4.7> For the design described in Exercise 4.20.5, add `nops` to this instruction sequence to ensure correct execution in spite of missing support for forwarding.

Exercise 4.21

This exercise is intended to help you understand the relationship between forwarding, hazard detection, and ISA design. Problems in this exercise refer to the following sequences of instructions, and assume that it is executed on a five-stage pipelined datapath:

	Instruction sequence

a.	`lw $1,40($6)`
	`add $2,$3,$1`
	`add $1,$6,$4`
	`sw $2,20($4)`
	`and $1,$1,$4`

b.	`add $1,$5,$3`
	`sw $1,0($2)`
	`lw $1,4($2)`
	`add $5,$5,$1`
	`sw $1,0($2)`

4.21.1 [5] <4.7> If there is no forwarding or hazard detection, insert `nops` to ensure correct execution.

4.21.2 [10] <4.7> Repeat Exercise 4.21.1 but now use `nops` only when a hazard cannot be avoided by changing or rearranging these instructions. You can assume register R7 can be used to hold temporary values in your modified code.

4.21.3 [10] <4.7> If the processor has forwarding, but we forgot to implement the hazard detection unit, what happens when this code executes?

4.21.4 [20] <4.7> If there is forwarding, for the first five cycles during the execution of this code, specify which signals are asserted in each cycle by hazard detection and forwarding units in Figure 4.60.

4.21.5 [10] <4.7> If there is no forwarding, what new inputs and output signals do we need for the hazard detection unit in Figure 4.60? Using this instruction sequence as an example, explain why each signal is needed.

4.21.6 [20] <4.7> For the new hazard detection unit from Exercise 4.21.5, specify which output signals it asserts in each of the first five cycles during the execution of this code.

Exercise 4.22

This exercise is intended to help you understand the relationship between delay slots, control hazards, and branch execution in a pipelined processor. In this exercise, we assume that the following MIPS code is executed on a pipelined processor with a five-stage pipeline, full forwarding, and a predict-taken branch predictor:

a.
```
Label1: lw  $1,40($6)
        beq $2,$3,Label2 ; Taken
        add $1,$6,$4
Label2: beq $1,$2,Label1 ; Not taken
        sw  $2,20($4)
        and $1,$1,$4
```

b.
```
        add $1,$5,$3
Label1: sw  $1,0($2)
        add $2,$2,$3
        beq $2,$4,Label1 ; Not taken
        add $5,$5,$1
        sw  $1,0($2)
```

4.22.1 [10] <4.8> Draw the pipeline execution diagram for this code, assuming there are no delay slots and that branches execute in the EX stage.

4.22.2 [10] <4.8> Repeat Exercise 4.22.1, but assume that delay slots are used. In the given code, the instruction that follows the branch is now the delay slot instruction for that branch.

4.22.3 [20] <4.8> One way to move the branch resolution one stage earlier is to not need an ALU operation in conditional branches. The branch instructions would be "`bez Rd,Label`" and "`bnez Rd,Label`", and it would branch if the register has and does not have a 0 value, respectively. Change this code to use these branch instruction instead of `beq`. You can assume that register `$8` is available for you to use as a temporary register, and that a `seq` (set if equal) R-type instruction can be used.

Section 4.8 describes how the severity of control hazards can be reduced by moving branch execution into the ID stage. This approach involves a dedicated comparator in the ID stage, as shown in Figure 4.62. However, this approach potentially adds to the latency of the ID stage, and requires additional forwarding logic and hazard detection.

4.22.4 [10] <4.8> Using the first branch instruction in the given code as an example, describe the hazard detection logic needed to support branch execution in the ID stage as in Figure 4.62. Which type of hazard is this new logic supposed to detect?

4.22.5 [10] <4.8> For the given code, what is the speed-up achieved by moving branch execution into the ID stage? Explain your answer. In your speed-up calculation, assume that the additional comparison in the ID stage does not affect clock cycle time.

4.22.6 [10] <4.8> Using the first branch instruction in the given code as an example, describe the forwarding support that must be added to support branch execution in the ID stage. Compare the complexity of this new forwarding unit to the complexity of the existing forwarding unit in Figure 4.62.

Exercise 4.23

The importance of having a good branch predictor depends on how often conditional branches are executed. Together with branch predictor accuracy, this will determine how much time is spent stalling due to mispredicted branches. In this exercise, assume that the breakdown of dynamic instructions into various instruction categories is as follows:

	R-Type	beq	jmp	lw	sw
a.	50%	15%	10%	15%	10%
b.	30%	10%	5%	35%	20%

Also, assume the following branch predictor accuracies:

	Always-taken	Always not-taken	2-bit
a.	40%	60%	80%
b.	60%	40%	95%

4.23.1 [10] <4.8> Stall cycles due to mispredicted branches increase the CPI. What is the extra CPI due to mispredicted branches with the always-taken predictor? Assume that branch outcomes are determined in the EX stage, that there are no data hazards, and that no delay slots are used.

4.23.2 [10] <4.8> Repeat Exercise 4.23.1 for the "always not-taken" predictor.

4.23.3 [10] <4.8> Repeat Exercise 4.23.1 for the 2-bit predictor.

4.23.4 [10] <4.8> With the 2-bit predictor, what speed-up would be achieved if we could convert half of the branch instructions in a way that replaces a branch instruction with an ALU instruction? Assume that correctly and incorrectly predicted instructions have the same chance of being replaced.

4.23.5 [10] <4.8> With the 2-bit predictor, what speed-up would be achieved if we could convert half of the branch instructions in a way that replaced each branch instruction with two ALU instructions? Assume that correctly and incorrectly predicted instructions have the same chance of being replaced.

4.23.6 [10] <4.8> Some branch instructions are much more predictable than others. If we know that 80% of all executed branch instructions are easy-to-predict loop-back branches that are always predicted correctly, what is the accuracy of the 2-bit predictor on the remaining 20% of the branch instructions?

Exercise 4.24

This exercise examines the accuracy of various branch predictors for the following repeating pattern (e.g., in a loop) of branch outcomes:

	Branch outcomes
a.	T, T, NT, T
b.	T, T, T, NT, NT

4.24.1 [5] <4.8> What is the accuracy of always-taken and always-not-taken predictors for this sequence of branch outcomes?

4.24.2 [5] <4.8> What is the accuracy of the two-bit predictor for the first four branches in this pattern, assuming that the predictor starts off in the bottom left state from Figure 4.63 (predict not taken).

4.24.3 [10] <4.8> What is the accuracy of the two-bit predictor if this pattern is repeated forever?

4.24.4 [30] <4.8> Design a predictor that would achieve a perfect accuracy if this pattern is repeated forever. Your predictor should be a sequential circuit with one output that provides a prediction (1 for taken, 0 for not taken) and no inputs other than the clock and the control signal that indicates that the instruction is a conditional branch.

4.24.5 [10] <4.8> What is the accuracy of your predictor from Exercise 4.24.4 if it is given a repeating pattern that is the exact opposite of this one?

4.24.6 [20] <4.8> Repeat Exercise 4.24.4, but now your predictor should be able to eventually (after a warm-up period during which it can make wrong predictions) start perfectly predicting both this pattern and its opposite. Your predictor should have an input that tells it what the real outcome was. Hint: this input lets your predictor determine which of the two repeating patterns it is given.

Exercise 4.25

This exercise explores how exception handling affects pipeline design. The first three problems in this exercise refer to the following two instructions:

	Instruction 1	Instruction 2
a.	`add $0,$1,$2`	`bne $1,$2,Label`
b.	`lw $2,40($3)`	`nand $1,$2,$3`

4.25.1 [5] <4.9> Which exceptions can each of these instructions trigger? For each of these exceptions, specify the pipeline stage in which it is detected.

4.25.2 [10] <4.9> If there is a separate handler address for each exception, show how the pipeline organization must be changed to be able to handle this exception. You can assume that the addresses of these handlers are known when the processor is designed.

4.25.3 [10] <4.9> If the second instruction from this table is fetched right after the instruction from the first table, describe what happens in the pipeline when the first instruction causes the first exception you listed in Exercise 4.25.1.

Show the pipeline execution diagram from the time the first instruction is fetched until the time the first instruction of the exception handler is completed.

The remaining three problems in this exercise assume that exception handlers are located at the following addresses:

	Overflow	Invalid data address	Undefined instruction	Invalid instruction address	Hardware malfunction
a.	0xFFFFF000	0xFFFFF100	0xFFFFF200	0xFFFFF300	0xFFFFF400
b.	0x00000008	0x00000010	0x00000018	0x00000020	0x00000028

4.25.4 [5] <4.9> What is the address of the exception handler in Exercise 4.25.3? What happens if there is an invalid instruction at that address in instruction memory?

4.25.5 [20] <4.9> In vectored exception handling, the table of exception handler addresses is in data memory at a known (fixed) address. Change the pipeline to implement this exception handling mechanism. Repeat Exercise 4.25.3 using this modified pipeline and vectored exception handling.

4.25.6 [15] <4.9> We want to emulate vectored exception handling (described in Exercise 4.25.5) on a machine that has only one fixed handler address. Write the code that should be at that fixed address. Hint: this code should identify the exception, get the right address from the exception vector table, and transfer execution to that handler.

Exercise 4.26

This exercise explores how exception handling affects control unit design and processor clock cycle time. The first three problems in this exercise refer to the following MIPS instruction that triggers an exception:

	Instruction	Exception
a.	`add $0,$1,$2`	Arithmetic overflow
b.	`lw $2,40($3)`	Invalid data memory address

4.26.1 [10] <4.9> For each stage of the pipeline, determine the values of exception-related control signals from Figure 4.66 as this instruction passes through that pipeline stage.

4.26.2 [5] <4.9> Some of the control signals generated in the ID stage are stored into the ID/EX pipeline register, and some go directly into the EX stage. Explain why, using this instruction as an example.

4.26.3 [10] <4.9> We can make the EX stage faster if we check for exceptions in the stage after the one in which the exceptional condition occurs. Using this instruction as an example, describe the main disadvantage of this approach.

The remaining three problems in this exercise assume that pipeline stags have the following latencies:

	IF	ID	EX	MEM	WB
a.	300ps	320ps	350ps	350ps	100ps
b.	200ps	170ps	210ps	210ps	150ps

4.26.4 [10] <4.9> If an overflow exception occurs once for every 100,000 instructions executed, what is the overall speed-up if we move overflow checking into the MEM stage? Assume that this change reduces EX latency by 30ns and that the IPC achieved by the pipelined processor is 1 when there are no exceptions.

4.26.5 [20] <4.9> Can we generate exception control signals in EX instead of in ID? Explain how this will work or why it will not work, using the "`bne $4,$5,Label`" instruction and these pipeline stage latencies as an example.

4.26.6 [10] <4.9> Assuming that each Mux has a latency of 40ps, determine how much the control unit has to generate the flush signals? Which signal is the most critical?

Exercise 4.27

This exercise examines how exception handling interacts with branch and load/store instructions. Problems in this exercise refer to the following branch instruction and the corresponding delay slot instruction:

	Branch and delay slot
a.	`beq $1,$0,Label` `sw $6,50($1)`
b.	`beq $5,$0,Label` `nor $5,$4,$3`

4.27.1 [20] <4.9> Assume that this branch is correctly predicted as taken, but then the instruction at "Label" is an undefined instruction. Describe what is done in each pipeline stage for each cycle, starting with the cycle in which the branch is decoded up to the cycle in which the first instruction of the exception handler is fetched.

4.27.2 [10] <4.9> Repeat Exercise 4.27.1, but this time assume that the instruction in the delay slot also causes a hardware error exception when it is in MEM stage.

4.27.3 [10] <4.9> What is the value in the EPC if the branch is taken but the delay slot causes an exception? What happens after the execution of the exception handler is completed?

The remaining three problems in this exercise also refer to the following store instruction:

	Store instruction
a.	`sw $6,50($1)`
b.	`sw $5,60($3)`

4.27.4 [10] <4.9> What happens if the branch is taken, the instruction at "Label" is an invalid instruction, the first instruction of the exception handler is the `sw` instruction given above, and this store accesses an invalid data address?

4.27.5 [10] <4.9> If load/store address computation can overflow, can we delay overflow exception detection into the MEM stage? Use the given store instruction to explain what happens.

4.27.6 [10] <4.9> For debugging, it is useful to be able to detect when a particular value is written to a particular memory address. We want to add two new registers, WADDR and WVAL. The processor should trigger an exception when the value

equal to WVAL is about to be written to address WADDR. How would you change the pipeline to implement this? How would this `sw` instruction be handled by your modified datapath?

Exercise 4.28

In this exercise we compare the performance of 1-issue and 2-issue processors, taking into account program transformations that can be made to optimize for 2-issue execution. Problems in this exercise refer to the following loop (written in C):

	C code
a.	`for(i=0;!=j;i++)` `  b[i]=a[i];`
b.	`for(i=0;a[i]!=a[i+1];i++)` `  a[i]=0;`

When writing MIPS code, assume that variables are kept in registers as follows, and that all registers except those indicated as Free are used to keep various variables, so they cannot be used for anything else.

	i	j	a	b	c	Free
a.	$1	$2	$3	$4	$5	$6,$7,$8
b.	$4	$5	$6	$7	$8	$1,$2,$3

4.28.1 [10] <4.10> Translate this C code into MIPS instructions. Your translation should be direct, without rearranging instructions to achieve better performance.

4.28.2 [10] <4.10> If the loop exits after executing only two iterations, draw a pipeline diagram for your MIPS code from Exercise 4.28.1 executed on a 2-issue processor shown in Figure 4.69. Assume the processor has perfect branch prediction and can fetch any 2 instructions (not just consecutive instructions) in the same cycle.

4.28.3 [10] <4.10> Rearrange your code from Exercise 4.28.1 to achieve better performance on a 2-issue statically scheduled processor from Figure 4.69.

4.28.4 [10] <4.10> Repeat Exercise 4.28.2, but this time use your MIPS code from Exercise 4.28.3.

4.28.5 [10] <4.10> What is the speed-up of going from a 1-issue processor to a 2-issue processor from Figure 4.69. Use your code from Exercise 4.28.1 for both

1-issue and 2-issue, and assume that 1,000,000 iterations of the loop are executed. As in Exercise 4.28.2, assume that the processor has perfect branch predictions, and that a 2-issue processor can fetch any 2 instructions in the same cycle.

4.28.6 [10] <4.10> Repeat Exercise 4.28.5, but this time assume that in the 2-issue processor one of the instructions to be executed in a cycle can be of any kind, and the other must be a non-memory instruction.

Exercise 4.29

In this exercise, we consider the execution of a loop in a statically scheduled superscalar processor. To simplify the exercise, assume that any combination of instruction types can execute in the same cycle, e.g., in a 3-issue superscalar, the three instructions can be three ALU operations, three branches, three load/store instruction, or any combination of these instructions. Note that this only removes a resource constraint, but data and control dependences must still be handled correctly. Problems in this exercise refer to the following loop:

	Loop

a.	`Loop: lw  $1,40($6)`
	`  add $5,$5,$1`
	`  sw  $1,20($5)`
	`  addi $6,$6,4`
	`  addi $5,$5,-4`
	`  beq $5,$0,Loop`

b.	`Loop: add $1,$2,$3`
	`  sw $0,0($1)`
	`  addi $2,$2,4`
	`  beq $2,$0,Loop`

4.29.1 [10] <4.10> If many (e.g., 1,000,000) iterations of this loop are executed, determine the fraction of all register reads that are useful in a 2-issue static superscalar processor?

4.29.2 [10] <4.10> If many (e.g., 1,000,000) iterations of this loop are executed, determine the fraction of all register reads that are useful in a 3-issue static superscalar processor? Compare this to your result for a 2-issue processor from Exercise 4.29.1.

4.29.3 [10] <4.10> If many (e.g., 1,000,000) iterations of this loop are executed, determine the fraction of cycles in which two or three register write ports are used in a 3-issue static superscalar processor?

4.29.4 [20] <4.10> Unroll this loop once and schedule it for a 2-issue static superscalar processor. Assume that the loop always executes an even number of iterations. You can use registers `$10` through `$20` when changing the code to eliminate dependences.

4.29.5 [20] <4.10> What is the speed-up of using your code from Exercise 4.29.4 instead of the original code with a 2-issue static superscalar processor. Assume that the loop has many (e.g., 1,000,000) iterations.

4.29.6 [10] <4.10> What is the speed-up of using your code from Exercise 4.29.4 instead of the original code with a pipelined (1-issue) processor. Assume that the loop has many (e.g., 1,000,000) iterations.

Exercise 4.30

In this exercise, we make several assumptions. First, we assume that an N-issue superscalar processor can execute any N instructions in the same cycle, regardless of their types. Second, we assume that every instruction is independently chosen, without regard for the instruction that precedes or follows it. Third, we assume that there are no stalls due to data dependences, that no delay slots are used, and that branches execute in the EX stage of the pipeline. Finally, we assume that instructions executed in the program are distributed as follows:

	ALU	Correctly predicted beq	Incorrectly predicted beq	lw	sw
a.	50%	18%	2%	20%	10%
b.	40%	10%	5%	35%	10%

4.30.1 [5] <4.10> What is the CPI achieved by a 2-issue static superscalar processor on this program?

4.30.2 [10] <4.10> In a 2-issue static superscalar whose predictor can only handle one branch per cycle, what speed-up is achieved by adding the ability to predict two branches per cycle? Assume a stall-on-branch policy for branches that the predictor can not handle.

4.30.3 [10] <4.10> In a 2-issue static superscalar processor that only has one register write port, what speed-up is achieved by adding a second register write port?

4.30.4 [5] <4.10> For a 2-issue static superscalar processor with a classic five-stage pipeline, what speed-up is achieved by making the branch prediction perfect?

4.30.5 [10] <4.10> Repeat Exercise 4.30.4, but for a 4-issue processor. What conclusion can you draw about the importance of good branch prediction when the issue width of the processor is increased?

4.30.6 <4.10> Repeat Exercise 4.30.5, but now assume that the 4-issue processor has 50 pipeline stages. Assume that each of the original five stages is broken into ten new stages, and that branches are executed in the first of ten new EX stages. What conclusion can you draw about the importance of good branch prediction when the pipeline depth of the processor is increased?

Exercise 4.31

Problems in this exercise refer to the following loop, which is given as x86 code and also as a MIPS translation of that code. You can assume that this loop executes many iterations before it exits. When determining performance, this means that you only need to determine what the performance would be in the "steady state", not for the first few and the last few iterations of the loop. Also, you can assume full forwarding support and perfect branch prediction without delay slots, so the only hazards you have to worry about are resource hazards and data hazards. Note that most x86 instructions in this problem have two operands each. The last (usually second) operand of the instruction indicates both the first source data value and the destination. If the operation needs a second source data value, it is indicated by the other operand of the instruction. For example, "sub (edx),eax" reads the memory location pointed by register edx, subtracts that value from register eax, and puts the result back in register eax.

	x86 Instructions	MIPS-like translation

a.	`Label: mov -4(esp), eax`
	`add (edx), eax`
	
	`mov eax, -4(esp)`
	`add 1, ecx`
	`add 4, edx`
	`cmp esi, ecx`
	`jl Label`
		`Label: lw $2,-4($sp)`
		`lw $3,0($4)`
		`add $2,$2,$3`
		`sw $2,-4($sp)`
		`addi $6,$6,1`
		`addi $4,$4,4`
		`slt $1,$6,$5`
		`bne $1,$0,Label`

b.	`Label: add eax, (edx)`
	
	`mov eax, edx`
	`add 1, eax`
	`jl Label`
		`Label: lw $2,0($4)`
		`add $2,$2,$5`
		`sw $2,0($4)`
		`add $4,$5,$0`
		`addi $5,$5,1`
		`slt $1,$5,$0`
		`bne $1,$0,Label`

4.31.1 [20] <4.11> What CPI would be achieved if the MIPS version of this loop is executed on a 1-issue processor with static scheduling and a five-stage pipeline?

4.31.2 [20] <4.11> What CPI would be achieved if the x86 version of this loop is executed on a 1-issue processor with static scheduling and a 7-stage pipeline? The stages of the pipeline are IF, ID, ARD, MRD, EXE, and WB. Stages IF and ID are similar to those in the five-stage MIPS pipeline. ARD computes the address of the memory location to be read, MRD performs the memory read, EXE executes

the operation, and WB writes the result to register or memory. The data memory has a read port (for instructions in the MRD stage) and a separate write port (for instructions in the WB stage).

4.31.3 [20] <4.11> What CPI would be achieved if the x86 version of this loop is executed on a processor that internally translates these instructions into MIPS-like micro-operations, then executes these micro-operations on a 1-issue five-stage pipeline with static scheduling. Note that the instruction count used in CPI computation for this processor is the x86 instruction count.

4.31.4 [20] <4.11> What CPI would be achieved if the MIPS version of this loop is executed on a 1-issue processor with dynamic scheduling? Assume that our processor is not doing register renaming, so you can only reorder instructions that have no data dependences.

4.31.5 [30] <4.10, 4.11> Assuming that there are many free registers available, rename the MIPS version of this loop to eliminate as many data dependences as possible between instructions in the same iteration of the loop. Now repeat Exercise 4.31.4, using your new renamed code.

4.31.6 [20] <4.10, 4.11> Repeat Exercise 4.31.4, but this time assume that the processor assigns a new name to the result of each instruction as that instruction is decoded, and then renames registers used by subsequent instructions to use correct register values.

Exercise 4.32

Problems in this exercise assume that branches represent the following fraction of all executed instructions, and the following branch predictor accuracy. Assume that the processor is never stalled by data and resource dependences, i.e., the processor always fetches and executes the maximum number of instructions per cycle if there are no control hazards. For control dependences, the processor uses branch prediction and continues fetching from the predicted path. If the branch has been mispredicted, when the branch outcome is resolved the instructions fetched after the mispredicted branch are discarded, and in the next cycle the processor starts fetching from the correct path.

	Branches as a % of all executed instructions	Branch prediction accuracy
a.	20	90%
b.	20	99.5%

4.32.1 [5] <4.11> How many instructions are expected to be executed between the time one branch misprediction is detected and the time the next branch misprediction is detected?

The remaining problems in this exercise assume the following pipeline depth and that the branch outcome is determined in the following pipeline stage (counting from stage 1):

	Pipeline depth	Branch outcome known in stage
a.	12	10
b.	25	18

4.32.2 [5] <4.11> In a 4-issue processor with these pipeline parameters, how many branch instructions can be expected to be "in progress" (already fetched but not yet committed) at any given time?

4.32.3 [5] <4.11> How many instructions are fetched from the wrong path for each branch misprediction in a 4-issue processor?

4.32.4 [10] <4.11> What is the speed-up achieved by changing the processor from 4-issue to 8-issue? Assume that the 8-issue and the 4-issue processor differ only in the number of instructions per cycle, and are otherwise identical (pipeline depth, branch resolution stage, etc.).

4.32.5 [10] <4.11> What is the speed-up of executing branches 1 stage earlier in a 4-issue processor?

4.32.6 [10] <4.11> What is the speed-up of executing branches 1 stage earlier in a 8-issue processor? Discuss the difference between this result and the result from Exercise 4.32.5.

Exercise 4.33

This exercise explores how branch prediction affects performance of a deeply pipelined multiple-issue processor. Problems in this exercise refer to a processor with the following number of pipeline stages and instructions issued per cycle:

	Pipeline depth	Issue width
a.	10	4
b.	25	2

4.33.1 [10] <4.11> How many register read ports should the processor have to avoid any resource hazards due to register reads?

4.33.2 [10] <4.11> If there are no branch mispredictions and no data dependences, what is the expected performance improvement over a 1-issue processor with the classical five-stage pipeline? Assume that the clock cycle time decreases in proportion to the number of pipeline stages.

4.33.3 [10] <4.11> Repeat Exercise 4.33.2, but this time every executed instruction has a RAW data dependence to the instruction that executes right after it. You can assume that no stall cycles are needed, i.e., forwarding allows consecutive instructions to execute in back-to-back cycles.

For the remaining three problems in this exercise, unless the problem specifies otherwise, assume the following statistics about what percentage of instructions are branches, predictor accuracy, and performance loss due to branch mispredictions:

	Branches as a fraction of all executed instructions	Branches execute in stage	Predictor accuracy	Performance loss
a.	30%	7	95%	10%
b.	15%	8	97%	2%

4.33.4 [10] <4.11> If we have the given fraction of branch instructions and branch prediction accuracy, what percentage of all cycles are entirely spent fetching wrong-path instructions? Ignore the performance loss number.

4.33.5 [20] <4.11> If we want to limit stalls due to mispredicted branches to no more than the given percentage of the ideal (no stalls) execution time, what should be our branch prediction accuracy? Ignore the given predictor accuracy number.

4.33.6 [10] <4.11> What should the branch prediction accuracy be if we are willing to have a speed-up of 0.5 (one half) relative to the same processor with an ideal branch predictor?

Exercise 4.34

This exercise is designed to help you understand the discussion of the "Pipelining is easy" fallacy from Section 4.13. The first four problems in this exercise refer to the following MIPS instruction:

	Instruction	Interpretation
a.	`add Rd,Rs,Rt`	`Reg[Rd]=Reg[Rs]+Reg[Rt]`
b.	`lw Rt,Offs(Rs)`	`Reg[Rt]=Mem[Reg[Rs]+Offs]`

4.34.1 [10] <4.13> Describe a pipelined datapath needed to support only this instruction. Your datapath should be designed with the assumption that the only instructions that will ever be executed are instances of this instruction.

4.34.2 [10] <4.13> Describe the requirements of forwarding and hazard detection units for your datapath from Exercise 4.34.1.

4.34.3 [10] <4.13> What needs to be done to support undefined instruction exceptions in your datapath from Exercise 4.34.1. Note that the undefined instruction exception should be triggered whenever the processor encounters any other kind of instruction.

The remaining two problems in this exercise also refer to this MIPS instruction:

	Instruction	Interpretation
a.	`beq Rs, Rt, Label`	`if Reg[Rs] == Reg[Rt]  PC=PC+Offs`
b.	`and Rd, Rs, Rt`	`Reg[Rd]=Reg[Rs]&Reg[Rt]`

4.34.4 [10] <4.13> Describe how to extend your datapath from Exercise 4.34.1 so it can also support this instruction. Your extended datapath should be designed to only support instances these two instructions.

4.34.5 [10] <4.13> Repeat Exercise 4.34.2 for your extended datapath from Exercise 4.34.4.

4.34.6 [10] <4.13> Repeat Exercise 4.34.2 for your extended datapath from Exercise 4.34.4.

Exercise 4.35

This exercise is intended to help you better understand the relationship between ISA design and pipelining. Problems in this exercise assume that we have a multiple-issue pipelined processor with the following number of pipeline stages, instructions issued per cycle, stage in which branch outcomes are resolved, and branch predictor accuracy:

	Pipeline depth	Issue width	Branches execute in stage	Branch predictor accuracy	Branches as a % of instructions
a.	10	4	7	80%	20%
b.	25	2	17	92%	25%

4.35.1 [5] <4.8, 4.13> Control hazards can be eliminated by adding branch delay slots. How many delay slots must follow each branch if we want to eliminate all control hazards in this processor?

4.35.2 [10] <4.8, 4.13> What is the speed-up that would be achieved by using four branch delay slots to reduce control hazards in this processor? Assume that there are no data dependences between instructions and that all four delay slots can be filled with useful instructions without increasing the number of executed instructions. To make your computations easier, you can also assume that the mispredicted branch instruction is always the last instruction to be fetched in a cycle, i.e., no instructions that are in the same pipeline stage as the branch are fetched from the wrong path.

4.35.3 [10] <4.8, 4.13> Repeat Exercise 4.35.2, but now assume that 10% of executed branches have all four delay slots filled with useful instruction, 20% have only three useful instructions in delay slots (the fourth delay slot is a `nop`), 30% have only two useful instructions in delay slots, and 40% have no useful instructions in their delay slots.

The remaining four problems in this exercise refer to the following C loop:

a.
```
for(i=0;i!=j;i++){
  b[i]=a[i];
}
```

b.
```
for(i=0;a[i]!=a[i+1];i++){
  c++;
}
```

4.35.4 [10] <4.8, 4.13> Translate this C loop into MIPS instructions, assuming that our ISA requires one delay slot for every branch. Try to fill delay slots with non-`nop` instructions when possible. You can assume that variables `a`, `b`, `c`, `i`, and `j` are kept in registers `$1`, `$2`, `$3`, `$4`, and `$5`.

4.35.5 [10] <4.7, 4.13> Repeat Exercise 4.35.4 for a processor that has two delay slots for every branch.

4.35.6 [10] <4.10, 4.13> How many iterations of your loop from Exercise 4.35.4 can be "in flight" within this processor's pipeline? We say that an iteration is "in flight" when at least one of its instructions has been fetched and has not yet been committed.

Exercise 4.36

This exercise is intended to help you better understand the last pitfall from Section 4.13—failure to consider pipelining in instruction set design. The first four problems in this exercise refer to the following new MIPS instruction:

	Instruction	Interpretation
a.	`lwinc Rt,Offset(Rs)`	`Reg[Rt]=Mem[Reg[Rs]+Offset]` `Reg[Rs]=Reg[Rs]+4`
b.	`addr Rt,Offset(Rs)`	`Reg[Rt]=Mem[Reg[Rs]+Offset]+Reg[Rt]`

4.36.1 [10] <4.11, 4.13> Translate this instruction into MIPS micro-operations.

4.36.2 [10] <4.11, 4.13> How would you change the five-stage MIPS pipeline to add support for micro-op translation needed to support this new instruction?

4.36.3 [20] <4.13> If we want to add this instruction to the MIPS ISA, discuss the changes to the pipeline (which stages, which structures in which stage) that are needed to directly (without micro-ops) support this instruction.

4.36.4 [10] <4.13> How often do you expect this instruction can be used. Do you think that we would be justified if we added this instruction to the MIPS ISA?

The remaining two problems in this exercise are about adding a new `addm` instruction to the ISA. In a processor to which `addm` has been added, these problems assume the following breakdown of clock cycles according to which instruction is completed in that cycle (or which stall is preventing an instruction from completing):

	add	beq	lw	sw	addm	Control Stalls	Data Stalls
a.	35%	20%	20%	10%	5%	5%	5%
b.	25%	10%	25%	10%	10%	10%	10%

4.36.5 [10] <4.13> Given this breakdown of execution cycles in the processor with direct support for the `addm` instruction, what speed-up is achieved by replacing this instruction with a 3-instruction sequence (`lw`, `add`, and then `sw`)? Assume that the `addm` instruction is somehow (magically) supported with a classical five-stage pipeline without creating resource hazards.

4.36.6 [10] <4.13> Repeat Exercise 4.36.5, but now assume that `addm` was supported by adding a pipeline stage. When `addm` is translated, this extra stage can be removed and, as a result, half of the existing data stalls are eliminated. Note that the data stall elimination applies only to stalls that existed before `addm` translation, not to stalls added by the `addm` translation itself.

Exercise 4.37

This exercise explores some of the tradeoffs involved in pipelining, such as clock cycle time and utilization of hardware resources. The first three problems in this exercise refer to the following MIPS code. The code is written with an assumption that the processor does not use delay slots.

a.
```
        lw  $1,40($6)
        beq $1,$0,Label ; Assume $1 == $0
        sw  $6,50($1)
Label:  add $2,$3,$1
        sw  $2,50($1)
```

b.
```
        lw $5,-16($5)
        sw $5,-16($5)
        lw $5,-20($5)
        beq $5,$0,Label ; Assume $5!=$0
        add $5,$5,$5
```

4.37.1 [5] <4.3, 4.14> Which parts of the basic single-cycle datapath are used by all of these instructions? Which parts are the least utilized?

4.37.2 [10] <4.6, 4.14> What is the utilization for the read and for the write port of the data memory unit?

4.37.3 [10] <4.6, 4.14> Assume that we already have a single-cycle design. How many bits in total do we need for pipeline registers to implement the pipelined design?

The remaining three problems in this exercise assume that components of the datapath have the following latencies:

	I-Mem	Add	Mux	ALU	Regs	D-Mem	Sign-extend	Shift-left-2
a.	400ps	100ps	30ps	120ps	200ps	350ps	20ps	0ps
b.	500ps	150ps	100ps	180ps	220ps	1000ps	90ps	20ps

4.37.4 [10] <4.3, 4.5, 4.14> Given these latencies for individual elements of the datapath, compare clock cycle times of the single-cycle and the five-stage pipelined datapath.

4.37.5 [10] <4.3, 4.5, 4.14> Repeat Exercise 4.37.4, but now assume that we only want to support ADD instructions.

4.37.6 [20] <4.3, 4.5, 4.14> If it costs $1 to reduce the latency of a single component of the datapath by 1ps, what would it cost to reduce the clock cycle time by 20% in the single-cycle and in the pipelined design?

Exercise 4.38

This exercise explores energy efficiency and its relationship with performance. Problems in this exercise assume the following energy consumption for activity in Instruction Memory, Registers, and Data Memory. You can assume that the other components of the datapath spend a negligible amount of energy.

	I-Mem	1 Register Read	Register Write	D-Mem Read	D-Mem Write
a.	100pJ	60pJ	70pJ	120pJ	100pJ
b.	200pJ	90pJ	80pJ	300pJ	280pJ

4.38.1 [10] <4.3, 4.6, 4.14> How much energy is spent to execute an `add` instruction in a single-cycle design and in the five-stage pipelined design?

4.38.2 [10] <4.6, 4.14> What is the worst-case MIPS instruction in terms of energy consumption, and what is the energy spent to execute it?

4.38.3 [10] <4.6, 4.14> If energy reduction is paramount, how would you change the pipelined design? What is the percentage reduction in the energy spent by a `lw` instruction after this change?

The remaining three problems in this exercise assume that components in the datapath have the following latencies. You can assume that the other components of the datapath have negligible latencies.

	I-Mem	Control	Register Read or Write	ALU	D-Mem Read or Write
a.	400ps	300ps	200ps	120ps	350ps
b.	500ps	400ps	220ps	180ps	1000ps

4.38.4 [10] <4.6, 4.14> What is the performance impact of your changes from Exercise 4.38.3?

4.38.5 [10] <4.6, 4.14> We can eliminate the MemRead control signal and have the data memory be read in every cycle, i.e., we can permanently have MemRead=1. Explain why the processor still functions correctly after this change. What is the effect of this change on clock frequency and energy consumption?

4.38.6 [10] <4.6, 4.14> If an idle unit spends 10% of the power it would spend if it were active, what is the energy spent by the instruction memory in each cycle? What percentage of the overall energy spent by the instruction memory does this idle energy represent?

Exercise 4.39

Problems in this exercise assume that, during an execution of the program, processor cycles are spent in the following way. A cycle is "spent" on an instruction if the processor completes that type of instruction in that cycle; a cycle is "spent" on a stall if the processor could not complete an instruction in that cycle because of a stall.

	add	beq	lw	sw	Control Stalls	Data Stalls
a.	35%	20%	20%	10%	10%	5%
b.	25%	10%	25%	10%	20%	10%

Problems in this exercise also assume that individual pipeline stages have the following latency and energy consumption. The stage expends this energy in order

to do its work within the given latency. Note that no energy is spent in the MEM stage during a cycle in which there is no memory access. Similarly, no energy is spent in the WB stage in a cycle in which there is no register write. In several of the following problems, we make assumptions about how energy consumption changes if a stage performs its work slower or faster than this.

	IF	ID	EX	MEM	WB
a.	300ps/120pJ	400ps/60pJ	350ps/75pJ	500ps/130pJ	100ps/20pJ
b.	200ps/150pJ	150ps/60pJ	120ps/50pJ	190ps/150pJ	140ps/20pJ

4.39.1 [10] <4.14> What is the performance (in instructions per second)?

4.39.2 [10] <4.14> What is the power dissipated in watts (joules per second)?

4.39.3 [10] <4.6, 4.14> Which pipeline stages can you slow down and by how much, without affecting the clock cycle time?

4.39.4 [20] <4.6, 4.14> It is often possible to sacrifice some speed in a circuit in order to reduce its energy consumption. Assume that we can reduce energy consumption by a factor of X (new energy is 1/X times the old energy) if we increase the latency by a factor of X (new latency is X times the old latency). Now we can adjust latencies of pipeline stages to minimize energy consumption without sacrificing any performance. Repeat Exercise 4.39.2 for this adjusted processor.

4.39.5 [10] <4.6, 4.14> Repeat Exercise 4.39.4, but this time the goal is to minimize energy spent per instruction while increasing the clock cycle time by no more than 10%.

4.39.6 [10] <4.6, 4.14> Repeat Exercise 4.39.5, but now assume that energy consumption is reduced by a factor of X^2 when latency is made X times longer. What are the power savings compared to what you computed for Exercise 4.39.2?

Answers to Check Yourself

§4.1, page 303: 3 of 5: Control, Datapath, Memory. Input and Output are missing.
§4.2, page 307: false. Edge-triggered state elements make simultaneous reading and writing both possible and unambiguous.
§4.3, page 315: I. A. II. C.
§4.4, page 330: Yes, Branch and ALUOp0 are identical. In addition, MemtoReg and RegDst are inverses of one another. You don't need an inverter; simply use the other signal and flip the order of the inputs to the multiplexor!
§4.5, page 343: 1. Stall on the LW result. 2. Bypass the first ADD result written into `$t1`. 3. No stall or bypass required.
§4.6, page 358: Statements 2 and 4 are correct; the rest are incorrect.
§4.8, page 383: 1. Predict not taken. 2. Predict taken. 3. Dynamic prediction.

§4.9, page 391: The first instruction, since it is logically executed before the others.
§4.10, page 403: 1. Both. 2. Both. 3. Software. 4. Hardware. 5. Hardware. 6. Hardware. 7. Both. 8. Hardware. 9. Both.
§4.11, page 404: First two are false and last two are true.
§4.12, page 6.7-3: Statements 1 and 3 are both true.
§4.12, page 6.7-7: Only statement 3 is completely accurate.