

$$1.2 \quad a. \quad \text{Yield} = \left(1 + \frac{0.30 \times 3.89}{4.0}\right)^{-4} = 0.36$$

$$\text{Dies per wafer} = \frac{\pi \times (30/2)^2}{3.89} - \frac{\pi \times 30}{\sqrt{2 \times 3.89}} = 182 - 33.8 = 148$$

$$\text{Cost per die} = \frac{\$500}{148 \times 0.36} = \$9.38$$

$$b. \quad \text{Yield} = \left(1 + \frac{.7 \times 1.86}{4.0}\right)^{-4} = 0.32$$

$$\text{Dies per wafer} = \frac{\pi \times (30/2)^2}{1.86} - \frac{\pi \times 30}{\sqrt{2 \times 1.86}} = 380 - 48.9 = 331$$

$$\text{Cost per die} = \frac{\$500}{331 \times .32} = \$4.72$$

$$c. \quad \$9.38 \times .4 = \$3.75$$

$$d. \quad \text{Selling price} = (\$9.38 + \$3.75) \times 2 = \$26.26$$

$$\text{Profit} = \$26.26 - \$4.72 = \$21.54$$

$$e. \quad \text{Rate of sale} = 3 \times 500,000 = 1,500,000/\text{month}$$

$$\text{Profit} = 1,500,000 \times \$21.54 = \$32,310,000$$

$$\$1,000,000,000 / \$32,310,000 = 31 \text{ months}$$

$$1.5 \quad a. \quad \frac{14 \text{ KW}}{(79 \text{ W} + 2.3 \text{ W} + 7.0 \text{ W})} = 158$$

$$b. \quad \frac{14 \text{ KW}}{(79 \text{ W} + 2.3 \text{ W} + 2 \times 7.0 \text{ W})} = 146$$

$$c. \quad \text{MTTF} = \frac{1}{9 \times 10^6} + 8 \times \frac{1}{4500} + \frac{1}{3 \times 10^4} = \frac{8 \times 2000 + 300}{9 \times 10^6} = \frac{16301}{9 \times 10^6}$$

$$\frac{1}{\text{Failure rate}} = \frac{9 \times 10^6}{16301} = 522 \text{ hours}$$

$$1.7 \quad a. \quad 50\%$$

$$b. \quad \frac{\text{Power new}}{\text{Power old}} = \frac{(V \times 0.50)^2 \times (F \times 0.50)}{V^2 \times F} = 0.5^3 = 0.125$$

$$c. \quad = \frac{.70}{(1-x) + x/2}; x = 60\%$$

$$d. \quad \frac{\text{Power new}}{\text{Power old}} = \frac{(V \times 0.70)^2 \times (F \times 0.50)}{V^2 \times F} = 0.7^2 \times 0.5 = 0.245$$

1.9 a. $FIT = 10^9/MTTF$
 $MTTF = 10^9/FIT = 10^9/100 = 10,000,000$

b. $Availability = \frac{MTTF}{MTTF + MTTR} = \frac{10^7}{10^7 + 24} = \text{about } 100\%$

1.10 Using the simplifying assumption that all failures are independent, we sum the probability of failure rate of all of the computers:

$$\text{Failure rate} = 1000 \times 10^{-7} = 10^{-4} = \frac{10^5}{10^9} \quad FIT = 10^5, \text{ therefore } MTTF = \frac{10^9}{10^5} = 10^4$$

1.13 a. Pentium 4 570: $.4 \times 3,501 + .6 \times 11,210 = 8,126$
 Athlon 64 X2 4,800+: $.4 \times 3,423 + .6 \times 20,718 = 13,800$

b. $20,718/7,621 = 2.7$

c. $x \times 3,501 + (1x) \times 11,210 = x \times 3,000 + (1x) \times 15,220$
 $x = .89$

.89/.11 = 8x ratio of memory to processor computation

1.14 a. Amdahl's Law:

$$\frac{1}{.6 + .4/2} = 1.25x \text{ speedup}$$

b. Amdahl's Law:

$$\frac{1}{.01 + .99/2} = 1.98x \text{ speedup}$$

c. Amdahl's Law:

$$\frac{1}{.2 + .8 \times (.6 + .4/2)} = 1.19x \text{ speedup}$$

d. Amdahl's Law:

$$\frac{1}{.8 + .2 \times (.01 + .99/2)} = 1.11x \text{ speedup}$$

- A.1 a. Forwarding is performed only via the register file. Branch outcomes and targets are not known until the end of the execute stage. All instructions introduced to the pipeline prior to this point are flushed.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
LD	R1, 0(R2)	F	D	X	M	W													
DADDI	R1, R1, #1		F	s	s	D	X	M	W										
SD	0(R2), R1					F	s	s	D	X	M	W							
DADDI	R2, R2, #4								F	D	X	M	W						
DSUB	R4, R3, R2								F	s	s	D	X	M	W				
BNEZ	R4, Loop												F	s	s	D	X	M	W
LD	R1, 0(R2)																	F	D

Since the initial value of R3 is R2 + 396 and equal instance of the loop adds 4 to R2, the total number of iterations is 99. Notice that there are 8 cycles lost to RAW hazards including the branch instruction. Two cycles are lost after the branch because of the instruction flushing. It takes 16 cycles between loop instances; the total number of cycles is $98 \cdot 16 + 18 = 1584$. The last loop takes two addition cycles since this latency cannot be overlapped with additional loop instances.

- b. Now we are allowed normal bypassing and forwarding circuitry. Branch outcomes and targets are known now at the end of decode.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14
LD	R1, 0(R2)	F	D	X	M	W									
DADDI	R1, R1, #1		F	s	D	X	M	W							
SD	0(R2), R1				F	D	X	M	W						
DADDI	R2, R2, #4					F	D	X	M	W					
DSUB	R4, R3, R2						F	D	X	M	W				
BNEZ	R4, Loop							F	s	D	X	M	W		
LD	R1, 0(R2)									F	D	X	M	W	

Again we have 99 iterations. There are two RAW stalls and a flush after the branch since the branch is taken. The total number of cycles is $9 \cdot 98 + 12 = 894$. The last loop takes three addition cycles since this latency cannot be overlapped with additional loop instances.

- c.

		1	2	3	4	5	6	7	8	9	10	11
LD	R1, 0(R2)	F	D	X	M	W						
DADDI	R2, R2, #4		F	D	X	M	W					
DSUB	R4, R3, R2			F	D	X	M	W				
DADDI	R1, R1, #1				F	D	X	M	W			
BNEZ	R4, Loop					F	D	X	M	W		
SD	-4 (R2), R1						F	D	X	M	W	
LD	R1, 0(R2)							F	D	X	M	W

The branch now has a single-cycle delayed branch slot. It is assumed that this slot will always be executed regardless of whether or not the branch was taken. The instructions have been rearranged in order to remove all RAW hazard stalls. The store was safely moved to the delay slot with the minor exception that the R2 had changed. This was compensated by decrementing four from the register. Now, there are only 6 cycles between loops and the last cycle requires 4 extra cycles to complete: $98 \cdot 6 + 10 = 598$.

Pipeline Stages											
	← Cycle Number →										
I	1	2	3	4	5	6	7	8	9	10	11
I	1	IF	RF	ALU1	MEM	ALU2	WB				
N	2		IF	RF	ALU1	MEM	ALU2	WB			
S	3			IF	RF	ALU1	MEM	ALU2	WB		
T	4				IF	RF	ALU1	MEM	ALU2	WB	
#	5					IF	RF	ALU1	MEM	ALU2	WB
	6						IF	RF	ALU1	MEM	ALU2 WB

- The simultaneous occurrence of IF, ALU1, and ALU2 maximizes the number of needed adders. This situation happens when instructions 2 and 4 are reg-mem ALU ADD instructions and instruction 6 can be any instruction that changes the PC via an increment.
- The WB and RF stages both access the register file. A reg-reg ALU operation will require two values from the register file in the RF stage. If this operation occurs as instruction 6 in the pipeline diagram above and an operation that writes to the register occurs as instruction 2, then we will need a 2 read port, 1 write port register file to sustain this combination of instructions. The IF and MEM stages both access memory. When these stages occur together, we will need up to 2 memory read ports and 1 write port.
- In the following table, we only consider data forwarding between operations that use the ALU.

The number in parentheses refers to the instruction number in the pipeline state diagram at the beginning of this solution.

Source Instruction Pipeline Stage	Destination Instruction Pipeline Stage	Remarks
ALU2(1)	ALU2(2)	ALUop (including Load result forwarding) to ALUop
ALU2(1)	MEM(3)	ALUop to ALUop via MEM
ALU2(1)	ALU1(4)	ALUop to a memory or branch ALUop

- In the following table, we consider data forwarding between operations that do not use the ALU.

The number in parentheses refers to the instruction number in the pipeline state diagram at the beginning of this solution.

Source Instruction Pipeline Stage	Destination Instruction Pipeline Stage	Remarks
MEM(1)	MEM(2)	Load ALUop to Store ALUop
ALU2(1)	MEM(3)	Load ALUop to Store ALUop

- In the following table, we show the number of stalls it takes to resolve dependencies between various instructions. The number in parentheses refers to the instruction number in the pipeline state diagram at the beginning of this solution.

Source Instruction Pipeline Stage	Destination Instruction Pipeline Stage	Stalls	Remarks
ALU2(1)	ALU1(2)	1	ALUop producing value for branch or memory op
ALU2(2)	ALU1(3)	2	ALUop producing value for branch or memory op
MEM(1)	ALU1(3)	1	Load op producing value for branch or memory op

- f. Assume Instruction 1 is a branch. It will take two cycles to determine if we are on the wrong path. If we are on the wrong path, Instructions 2 and 3 will have to be flushed.
- A.6 a. The traditional five stage pipeline of IF, ID, EX, MEM, WB can be modified to IF, ID, MEM, EX, WB if we only want to support register indirect addressing.
- b. The following illustrates the forwarding paths in the modified pipeline.

Source Instruction Pipeline Stage	Destination Instruction Pipeline Stage	Remarks
MEM	MEM	Load to Store (data value or index)
EX	MEM	ALU OP producing the register value for a Load or Store
EX	EX	Dependant ALU operation

- c. The following illustrates the new data hazards in the modified pipeline.

Examples:	Remarks
ADD R1, R2, R3	
LD R5, 0(R1)	Instruction has to stall for R1
ADD R1, R2, R3	
SW 0(R7), R1	Instruction has to stall for R1

- d. There are several ways that this modified pipeline can have a different instruction count for a given program over the original pipeline. The modified pipeline enables the compiler to merge loads and ALU operations, thereby reducing instruction count. However, these instructions cannot be merged if the memory operation requires an offset. If a memory operation requires an offset and we are not able to merge it with another instruction, the modified pipeline requires an additional instruction to compute the offset value and store it in a register.

Examples:				
Original Pipeline		Modified Pipeline		Remarks
LD	R9, 16(R1)	ADDUI	R2, R1, #16	Requires an additional instruction
		LD	R9, 0(R2)	
LD	R6, 8(R1)	ADDUI	R2, R1, #8	No gain nor loss (See Part (e))
ADD	R7, R6, R8	ADD	R7, R8, 0(R2)	
LD	R6, 0(R1)	ADD	R7, R8, 0(R1)	We are able to merge two instructions
ADD	R7, R6, R8			

- e. In the original pipeline, a load producing a value for the next instruction causes a stall. Since this is not the cause for the modified pipeline, the CPI decreases. The stall can also be “removed” by merging these two instructions in the modified pipeline. However, if an offset value has to be calculated the execute stage cannot deliver the value to the MEM stage fast enough, thereby causing a one cycle stall.

Example:		Remarks:	
Original Pipeline		Modify Pipeline	
LD	R6, 0(R1)		
ADD	R7, R6, R8	1 Cycle Stall	No Stalls
ADDUI	R6, R5, #8		
LD	R7, 0(R6)	No Stall	1 Cycle Stall

A.7 a. Delayed Branch—Penalties

	From Before	From Fall-Through	From Target
Branch Taken	0	1	0
Branch Not Taken	0	0	1

If the compiler takes an instruction from before the branch, it must be sure that there are no dependencies between that instruction and the predicate of the branch. If the compiler takes an instruction from fall-through, it must make sure that there are no dependencies between that instruction and instructions from the target of the branch. If it takes an instruction from target, the compiler must make sure that there are no dependencies between that instruction and the instruction from fall-through. Please note that these rules only apply to true dependencies. If an instruction from the correct path overwrites the value produced from the instruction in the branch delay slot, then essentially we are correcting the value stored in that particular register. However, we would still have to pay the penalty.

b. Cancel-If-Not-Taken—Penalties

	From Before	From Fall-Through	From Target
Branch Taken	0	1	0
Branch Not Taken	1	1	1

If the compiler takes an instruction from before the branch, it must be sure that there are no dependencies between that instruction and the predicate of the branch. However we have to be concerned that the instruction in the delay slot might be cancelled. Therefore, the compiler has to account for this possibility. Since there would be a penalty regardless of the branch being taken or not, the compiler shouldn't take an instruction from fall-through. Since the branch delay slot instruction is canceled if not taken, the compiler does not have to worry about dependencies when taking an instruction from target to fill the slot.

- c. If the compiler seeks to fill the branch delay slot from before the branch, then the delayed branch scheme is the best. This scheme will ensure the execution of the branch delay slot instruction regardless of the branch's outcome.

If the compiler seeks to fill the branch delay slot from fall-through, then the branch delayed scheme is the best. As pointed out in part b, there is a penalty using this scheme regardless of the branch's outcome.

If the compiler seeks to fill the branch delay slot from target, cancel-if-not-taken is better since the compiler would not have to worry about ensuring correctness when the branch is not taken.