PARALLEL COMPUTER ORGANIZATION AND DESIGN

Michel Dubois, Murali Annavaram and Per Stenström

Exercise Solution Manual

June 2012

CHAPTER 1

Problem 1.1

a. Using Amdahl's speedup,

$$\frac{1}{1 - F_{fp} + \frac{F_{fp}}{10}} > \frac{1}{1 - F_{ls} + \frac{F_{ls}}{2}}$$

Therefore,

$$\frac{F_{fp}}{F_{ls}} > \frac{5}{9}$$

b.Can you still find out which improvement is better based on these numbers?

Yes. The reason is F_{fp}/F_{ls} is equal to the ratio of the execution time of floating point instructions ($ExTime_{fp}$) and the execution time of loads and stores ($ExTime_{ls}$). We can get the ratio of these execution time with given information. If the ratio is larger than 5/9, which is the value obtained in part a, then we can say the floating point upgrade is better than the loads/stores upgrade.

$$ExTime_{fp} = IC_{fp} \times CPI_{fp} \times T_c$$

$$ExTime_{ls} = IC_{ls} \times CPI_{ls} \times T_c$$

$$ExTime_{total} = IC_{total} \times CPI_{total} \times T_c$$

Therefore,

$$\frac{F_{fp}}{F_{ls}} = \frac{IC_{fp} \times CPI_{fp} \times T_c}{IC_{ls} \times CPI_{ls} \times T_c} = \frac{ExTime_{fp}}{ExTime_{ls}}$$

Can you still estimate the maximum speedup of each upgrade using Amdahl's law?

No, because the total execution time or average CPI of *all* instructions is not given and we cannot get the fraction of execution time spent in floating point and loads/stores instructions respectively.

c. Floating-point improvement:
$1.5 = 1 / (1-F_{fp} + F_{fp}/10)$
Therefore, $F_{fp} = 0.3707$

Loads and Stores improvement:
$1.5 = 1 / (1-F_{ls} + F_{ls}/2)$
Therefore, $F_{ls} = 0.67$

d. After upgrading to the floating point units, the execution time is

$$ExTime_{fp} = \left(0.7 + \frac{0.3}{10}\right) \times ExTime_{base} = 0.73 \times ExTime_{base}$$

and the new fraction of time spent in loads and stores after the floating point upgrade is 0.20/0.73 = 0.274(27.4%).

Therefore, the maximum speedup of the cache upgrade after the floating point unit upgrade is

$$Speedup = \frac{1}{1 - 0.274 + \frac{0.274}{2}} = 1.1587$$

Problem 1.2

We solve this problem assuming that N≥1024 and the number of available processors *P* in the machine goes from 1 to 1024.

a. $$Speedup = \frac{T_1}{T_p} = \frac{NT_c}{\frac{N}{P}T_c + NT_b} = \frac{PR}{R + P}$$

b. For a given *R*, the speedup increases monotonically as *P* increases. The maximum speedup is thus achieved when *P* is 1024 and the maximum speedup is

$$Speedup_{max} = \frac{1024R}{1024 + R} :$$

c. $P > \frac{R}{R - 1}$

d. The execution time stays constant at NxT_c for all *P*'s >1 and given *N*. As *P* increases from 1 to 1024, the workload size (N_1) increases so that:

$$NT_c = \frac{N_1}{P}T_c + N_1T_b$$

and:

$$N_1 = \frac{NPR}{R + P}$$

As *P* grows N_1 tends to an asymptote equal to *N*x*R*.

e. Reconsidering a-c above in the context of growing workload size.
In this part, for given *N*, the workload (N_1) grows according to d) above, so that the execution time remains constant. In this context T_1 is equal to N_1T_c and T_P remains fixed at NT_c

$$Speedup = \frac{N_1T_c}{NT_c} = \frac{PR}{R + P}$$

Surprisingly the speedup with a growing workload is the same as the speedup in part a with constant size workload, and therefore the maximum speedup and minimum *P* are also the same as in part a. The reason is that the serial part (the bus accesses) grows with *P*, contrary to Amdahl's or Gustafson's laws where the serial part remains a constant.

f. Let $O = T_o/T_b$.

$$Speedup = \frac{T_1}{T_p} = \frac{NT_c}{\frac{N}{P}T_c + PT_o + NT_b} = \frac{PR}{R + P + \frac{OP^2}{N}}$$

If P varies and R and O are fixed, the partial derivative with respect to P is:

$$\frac{\partial S}{\partial P} = \frac{R^2 - \frac{OR}{N}P^2}{\left(P + R + \frac{OP^2}{N}\right)^2}$$

To get the maximum value,

$$\frac{\partial S}{\partial P} = 0$$

$$R^2 - O\frac{P}{N}P^2 = 0 \quad \text{or } P = \sqrt{\frac{NR}{O}}$$

Because we have 1024 processors, when $1 \le \sqrt{\frac{NR}{O}} \le 1024$, the maximum speedup with given R and O will be:

$$Speedup_{max} = \frac{R\sqrt{\frac{NR}{O}}}{2R + \sqrt{\frac{NR}{O}}}$$

If $\sqrt{\frac{NR}{O}} > 1024$, the maximum speedup is obtained when P is 1024 with given R and O and its value is given by

$$Speedup_{max} = \frac{1024R}{R + 1024 + \frac{1024^2 \times O}{N}}$$,where N is fixed.

g. The single processor time is N_1Tc. Moreover, $NT_c = N_1T_c/P + PT_o + N_1T_b/P$, therefore

$$N_1 = \frac{(NR - OP)P}{(P + R)}$$

$$Speedup = \frac{T_1}{T_p} = \frac{N_1}{N} = \frac{NRP - OP^2}{N(R + P)}$$

The partial derivative with respect to P is:

$$\frac{\partial S}{\partial P} = \frac{NR^2 - 2POR - P^2O}{(R + P)^2} = 0 \qquad \frac{\partial S}{\partial P} = 0$$

$$P^2 + 2RP - \frac{NR^2}{O} = 0$$

$$P = -R \pm R\sqrt{\left(1 + \frac{N}{O}\right)}$$

Because P must be equal to or greater than 1, $P_{max} = -R + R\sqrt{\left(1 + \frac{N}{O}\right)}$.

If $1 \le P_{max} \le 1024$, the maximum speedup is $Speedup_{max} = \frac{NRP_{max} - OP_{max}{}^2}{N(R + P_{max})}$.

If $1024 < P_{max}$, the maximum speedup is obtained for P is 1024 $Speedup_{max} = \frac{1024NR - 1024^2O}{N(R + 1024)}$

Problem 1.3

a.

Table 1: Speedups of the three programs and average execution times (sec)

Machines	Program 1	Program 2	Program 3	Average Normalized Execution Time
Base machine	1	1	1	3.67
Base + FP units	1	5	1.67	2.334
Base + cache	1.43	1.11	1.25	2.903

Table 2: Average speedups

Machines	S1	S2	S3	S4
Base + FP units	1.57	2.55	1.67	2.03
Base + cache	1.26	1.26	1.25	1.26

The ratio of average execution time (S1): Base + FP units is better than Base + cache by 25%.

Arithmetic Mean (S2): Base + FP unit is better than Base + cache by a factor of 2.

Harmonic Mean (S3): Base + FP unit is better than Base + cache by 33%.

Geometric Mean (S4): Base + FP unit is better than Base + cache by 61%.

b. Even if we use normalized execution times of programs, the speedup for each program is equal to the speedup based on execution time in part a). Only the average normalized execution time is slightly different. Hence, average speedups are unchanged, except for S1 (see Table 3 and Table 4). In all cases, Base + FP units is still better than Base + cache.

Table 3: Speedups of the three programs and average normalized execution times

Machines	Program 1	Program 2	Program 3	Average Normalized Execution Time
Base machine	1	1	1	1
Base + FP units	1	5	1.67	0.6
Base + cache	1.43	1.11	1.25	0.8

Table 4: Average speedups

Machines	S1	S2	S3	S4
Base + FP units	1.67	2.55	1.67	2.03
Base + cache	1.25	1.26	1.25	1.26

Problem 1.4

a. Not enough data
The reason is that the total execution time (in cycles) is not given.

b. The clock cycle of M2 is 1.25 times the clock cycle of M1. But the improvement for all Multiplies is a factor 20.
Time $_{M2}$ = 1.25*(0.8*Time$_{M1}$ + 1/20*0.2*Time$_{M1}$)= 1.0125 Time$_{M1}$
Therefore,

$$\frac{Time_{M2}}{Time_{M1}} = 1.0125$$

M2 is 1.25% slower than M1. Thus the proposal is not an improvement.

c.This is caused by a simulation bug. According to Amdahl’s law, the maximum speedup cannot be than 1.

$$Speedup = \frac{Time_{M1}}{Time_{M3}} = \frac{1}{\left(0.8 + \frac{0.2}{s}\right) \times 1.25} \le 1$$

Problem 1.5

a. Average CPI$_{base}$ = 0.4*1 + 0.25*2+0.1*1 + 0.08*1 + 0.12*3+0.05*1 = 1.49

b. Branch instructions including taken and untaken are 20% of the Instruction Count of the base machine. Therefore, 10% of Instruction Count is removed from the Instruction Count of the base machine and

IC$_{new}$ = 0.9*IC$_{base}$

We also need to get the new fraction of execution of each instruction class in order to compute CPI$_{new}$. For every instruction inst the new fraction of instructions of type *inst* left in the code is:

f$_{inst\_new}$ = IC$_{inst}$ / IC$_{new}$

Because SLT is an Arithmetic/Logic instruction, only IC$_{ALU}$ (the instruction count of Arithmetic/Logic) is changed, and other IC$_{inst}$ are unchanged. Each IC$_{inst}$ can be obtained from Table 1.6.
For Arithmetic/Logic instructions,

IC$_{ALU\_new}$ = (0.4-0.1)*IC$_{base}$

For other instructions,

IC$_{inst}$ = f$_{inst\_base}$ * IC$_{base}$

For ALU instructions, the new frequency is:
f$_{ALU\_new}$ = IC$_{ALU\_new}$ / IC$_{new}$ = (0.4-0.1)*IC$_{base}$ / (0.9*IC$_{base}$) = 0.33
For instructions other than ALU instructions, the frequency is:
f$_{inst\_new}$ = IC$_{inst}$/IC$_{new}$ = f$_{inst\_base}$ * IC$_{base}$ / (0.9*IC$_{base}$) = f$_{inst\_base}$ /0.9

Hence, the new CPI is

CPI$_{new}$ = f$_{ALU\_new}$*1+f$_{Load\_new}$*2+f$_{Stores\_new}$*1+f$_{Br\_nt\_new}$*1+f$_{Br\_t\_new}$*3+f$_{misc\_new}$*1
= 0.33*1 + f$_{Load\_base}$*2+f$_{Stores\_base}$*1+f$_{Br\_nt\_base}$*1+f$_{Br\_t\_base}$*3+f$_{misc\_base}$*1

$$CPI_{New} = \frac{1}{0.9}((0.4 - 0.1) \times 1 + 0.25 \times 2 + 0.1 \times 1 + 0.08 \times 1 + 0.12 \times 3 + 0.05 \times 1) = 1.544$$

c. Yes, this is a good idea. Comparing the execution times of the base machine and of the new machine with BLT-type instructions, the new machine is better than the base machine. Even though the CPI and the cycle time of the new machine are raised, the number of instructions (IC) is reduced. Therefore, the execution time of the new machine is shorter than the base machine.

$$ExTime_{base} = IC_{base} \times CPI_{base} \times Tc_{base}$$

$$ExTime_{new} = IC_{new} \times CPI_{new} \times Tc_{new} = 0.9 \times IC_{base} \times \frac{1.544}{1.49} \times CPI_{base} \times 1.05 \times Tc_{base}$$

$$= 0.98 \times ExTime_{base}$$

Problem 1.6

Since the data is related to the new machine, we have to find the data for the base machine without improvements to obtain the speedup.

The fraction of time that the new machine with 16 cores runs a single core is 25%. During that time 30% is used for floating point operations, which are 4 times faster than on the base machine. The fraction of time on the new machine is 0.25x.3=.075.
The fraction of time with a single core and no floating point operation is 0.25x.7=.175.

The fraction of time the new machine runs 16 cores is 75%. During that time each core runs floating point operations 30% of the time. Thus the fraction of time on the new machine is 0.75x.3=.225
The fraction of time with 16 cores and no floating point operation is 0.75x.7=.525.

First consider the upgrade to a 16-way CMP, with no fp unit. Let T_{16_nofp} be the execution time on this new machine and T_{base} the execution time on the base machine. In the phases when the 16-core machine executes 16 threads in parallel, the base machine must executed them one at a time.
T_{base_nofp}= (.25+.75x16)xT_{16_nofp}=12.25xT_{16_nofp}
Now consider adding the floating point units.
T_{base_fp}=(.3x4+.7)xT_{base_nofp}= 1.2x12.25xT_{16_nofp}=14.7xT_{16_nofp}

Therefore the speedup is 14.7.

CHAPTER 2

Problem 2.1

a. Dynamic power only:
Let's say dynamic power of the base machine is $P_{dynamic\_base}$, and Delay of the base machine is D_{base}.
$P_{dynamic\_base}$ is proportional to CV^2f.

The circuit for the single-cycle CPU is divided into 5 stages and each stage has 1/5 of the delay of the base machine, which is the single-cycle processor clocked at f. Therefore, the frequency of the 5-stage pipeline is 5f. We can compare the single cycle processor at frequency f with the 5-stage pipeline processor clocked at 5f first. Because the 5-stage pipeline clocked at 5f can execute a given workload 5 times faster, its delay is 1/5 of D_{base}.

Regarding the dynamic power of the 5-stage pipeline, we do not need to raise the supply voltage with the frequency because the switching speed of circuits is unchanged. The overall circuit is (roughly) the same and therefore the total capacitance is also the same. Therefore, the dynamic power of the 5-stage pipeline clocked at 5f is:
$P_{dynamic\_5pipe\_5f} = C*V^2*5f = 5*P_{dynamic\_base}$

The 5-stage pipeline clocked at f has the same delay as the base machine. The supply voltage can be lowered by a factor 5. As its frequency is 1/5 of the 5-stage pipeline at clock 5f, its dynamic power is $1/125 * 5*P_{dynamic\_base} = 1/25* P_{dynamic\_base}$.

Table 5: Summary of metric ratios wrt Base for dynamic power

Processor	Power (P)	Delay(D)	Energy(E)	Energy-Delay(PD^2)	Energy-Delay2(PD^3)
5-stage pipeline clocked at 5f	5	1/5	1	1/5	1/25
5- stage pipeline clocked at f	1/25	1	1/25	1/25	1/25
5-way multiprocessor clocked at f (single cycle CPU)	5	1/5	1	1/5	1/25
5-way multiprocessor clocked at 5f (5-stage pipeline)	25	1/25	1	1/25	1/625

The 5-way multiprocessor in which each processor is the single cycle CPU clocked at f runs 5 times faster than the base machine due to the number of processors. Therefore, the delay is 1/5 of the base machine. Because there are 5 processors, the total capacitance is multiplied by 5 and power is multiplied by 5. The frequency is the same as in the base machine. Therefore, the dynamic power is $5*P_{dynamic\_base}$.

The 5-way multiprocessor in which each processor is the 5-stage pipeline clocked at 5f can run 5 times faster than one 5-stage pipeline processor clocked at 5f, whose delay is $(1/5)*D_{base}$. The delay of the 5-way multiprocessor using 5-stage pipelines clocked at 5f is $(1/25)*D_{base}$. Its dynamic power is 5 times the dynamic power of the 5-stage pipeline processor clocked at 5f because of the added capacitance. Hence, the dynamic power for this design is 25*$P_{dynamic\_base}$.

b. Consider static power only. Delays have not changed. They are the same as in part a).
Static power is proportional to V(supply voltage) and area as you can see the formula given in the course notes:

$$P_{subthreshold} = VI_{sub} \propto Ve^{-k\frac{V_T}{T}} \quad \text{(for a given circuit)}$$

Because the 5-stage pipeline clocked at 5f is a uniprocessor and its area is almost the same as the area of the base machine, its static power is also roughly the same as the base machine. However, the static power of the 5-stage pipeline clocked at f is reduced because its supply voltage is scaled down by a factor 5.

The static power of the multiprocessors is 5 times the static power of each of their processors, and so their static power is multiplied by 5 due to the area increase.

Table 6: Summary of metric ratios w.r.t. Base for static power

Processor	Power(P)	Delay(D)	Energy(E)	Energy-Delay(PD^2)	Energy-Delay^2(PD^3)
5-stage pipeline clocked at 5f	1	1/5	1/5	1/25	1/125
5- stage pipeline c locked at f	1/5	1	1/5	1/5	1/5
5-way multiprocessor clocked at f (single cycle CPU)	5	1/5	1	1/5	1/25
5-way multiprocessor clocked at 5f (5-stage pipeline)	5	1/25	1/5	1/125	1/3125

c. Considering thermal and packaging issues, the 5-stage pipeline clocked at f is the best architecture for both dynamic and static power.

For battery-powered embedded systems, the 5-stage pipeline clocked at f is the most efficient architecture based on the energy metric for both dynamic and static power.

For both workstations and high-performance systems, the 5-way multiprocessor in which each processor is a 5 stage pipeline clocked at 5f shows the best results in Energy-delay and energy-delay^2 because of the shorter delay. Hence, this is the best design for that class of machines.

When static power is a significant component of the overall power then using higher frequency (by pipelining) is a better option than running the program slowly at lower frequency. Since the longer the program takes to complete the longer static power is burned. Hence, it may be better to finish the program execution as fast as possible and then shut down (or power gate) idle components.

Problem 2.2

(a) Device characteristics if future supply voltage(Vdd) does not scale and stays constant

Device/Wire characterstics	Proportionality	Scaling
Transistor Gain(Beta)	W/(L.tox)	S
Current(Ids)	Beta(Vdd-Vth)^2	z*S
Resistance	Vdd/Ids	1/z*S
Gate capacitance	(W.L)/tox	1/S
Gate delay	RC	1/z*S^2
Clock frequency	1/RC	z*S^2
Area occupied by circuit	W.L	1/S^2
Wire resistance per unit length	1/(w.h)	S^2
Wire capacitance per unit length	h/s	1

(b) Z is a constant as Vdd stays same but Vth scale, which effectively increases the current proportional to S, rather than decrease by S with Vdd scaling. Resistance decreases and hence the gate delay decreases. Thus clock frequency improves. Obviously while some of the device characteristics such as delay get better the overall dynamic power consumption increases since Vdd does not scale while the frequency (gate delay) does not increase to compensate for the increased power consumption from unchanged Vdd.

(c) Device characterstics when supply voltage(Vdd) and threshold voltage remain constant

Device/Wire characterstics	Proportionality	Scaling
Transistor Gain(Beta)	W/(L.tox)	S
Current(Ids)	Beta(Vdd-Vth)^2	S
Resistance	Vdd/Ids	1/S
Gate capacitance	(W.L)/tox	1/S
Gate delay	RC	1/S^2
Clock frequency	1/RC	S^2
Area occupied by circuit	W.L	1/S^2
Wire resistance per unit length	1/(w.h)	S^2
Wire capacitance per unit length	h/s	1

Problem 2.3

Base machine has functional block that has delay of 64FO4 units.
(a) Functional unit is pipelined into 4 pipeline stages. So the voltage is reduced by ¼ and thus power is reduced by 1/16.

(b) Pipeline with additional latch overhead. With 8 pipeline stages each pipeline stage has a delay of 8FO4+2FO4 (overhead). Hence, the total delay of each stage is now 10FO4 stages, which implies frequency is decreased to F/1.25. To keep the frequency constant voltage must be scaled up by 25%.

Dynamic Power is proportional to $(1.25V/4)^2$ for the new pipeline. Hence the power decreases by about 10.2X instead of 16X in the ideal case.

With 16 stages the pipeline delay per stage is 4FO4+2FO4. Hence the frequency decreases by 1.5X. To keep the frequency constant the voltage must be scaled by 50%. Hence, the dynamic power is $(1.5V/4)^2$. Hence, power decreases by 7.1X only.

(c) The 64FO4 is replicated 4 times. So each circuit can run at ¼ th of original frequency. Supply voltage is reduced to ¼. So the total dynamic power is now $4 * (1/4)^3 = 1/16$.

(d) For every N instructions performed in parallel, M instructions are sequential. If N is 4 and M is 1, five instructions are executed every two cycles. If M is zero, we can execute 8 instructions every two cycles. Hence, if M=1, the system performance is 5/8=62.5% of the performance of a perfectly parallel system.
The power consumption of either a perfectly parallel or a partially parallel system is the same. But the delay of the system with M=1 increases by 60%. Hence the total energy consumption of the N=4, M=1 system is 1.6X the of the energy consumption of the perfectly parallel system.

Problem 2.4

The size of the cache is one word (assume one word = 32 bits), and the word is filled into the cache at the first cycle of the program execution. So let's assume from cycle 1, the word is exposed to the particle strike which follows Poisson distribution with arrival rate = 10^{-25} per cycle.

(a) Assume that a particle strike always flips one bit. The probability that there is only one bit flipped after 1 billion cycles can be obtained in three steps:

1. The probability that one bit is flipped during one cycle is obtained from Poisson probability mass function for all the odd hits:

$$p_{\text{fault,1cyc}} = \sum_{\forall \text{ odd } k} f_{\text{Poisson}}(k \mid \lambda) = \sum_{\forall \text{ odd } k} \frac{\lambda^k e^{-\lambda}}{k!} \cong \frac{10^{-25} e^{10^{-25}}}{1!} \cong 10^{-25}$$

2. $p_{\text{fault,1cyc}}$ gives the probability that a bit is flipped after one cycle period. Therefore, the probability that one bit remains flipped (faulty) after 1 billion cycles is the probability that a bit flips an odd number of times during 1 billion cycles, and can be obtained from the probability mass function of binomial distribution:

$$\begin{aligned} p_{\text{fault,1 billion cyc}} &= \sum_{\forall \text{ odd } k}^{10^9} f_{\text{Binomial}}(k \mid 10^9, p_{\text{fault,1cyc}}) \\ &= \binom{10^9}{k} (p_{\text{fault,1cyc}})^k (1 - p_{\text{fault,1cyc}})^{10^9 - k} \\ &\cong \binom{10^9}{1} (10^{-25})(1 - 10^{-25})^{10^9 - 1} \cong 10^{-16} \end{aligned}$$

3. Now we know the probability that any one bit becomes faulty after 1 billion cycles, we want to know the probability that only one bit becomes faulty after 1 billion cycles among one word of

32 bits. Again, this can be obtained from the p.m.f of binomial distribution:

$$p_{\text{single-fault,1 billion cyc}} = \binom{32}{1}\left(p_{\text{fault,1 billion cyc}}\right)^{1}\left(1 - p_{\text{fault,1 billion cyc}}\right)^{32-1}$$
$$= 3.2 \times 10^{-15}$$

The above probability gives the expected number of failure during 1 billion cycles, because there could be at most one failure happening from any one fault. If we assume that this single bit fault becomes a failure, the FIT rate is obtained by dividing the expected number of failures by the total time and then by scaling it properly as follows:

$$\text{FIT}\left[\frac{failures}{10^9\ hours}\right] = \frac{3.2 \times 10^{-15} \times 1\ [failures]}{10^9 \times 10^{-9}\ [seconds]} \times \frac{10^9 \times 3600\ [seconds]}{[10^9\ hours]} = 0.01152$$

Under the assumption that the failure rate does not change over time, we can calculate the MTTF from FIT:

$$\text{MTTF}\ [years] = \frac{10^9}{\text{FIT} \times 24 \times 365} = 9.9265 \times 10^6 = 9.9265\ million\ years$$

(b) In this case, a word’s reliability lifetime is renewed every one million cycles. The expected number of failures decides FIT rate during one million cycles:

$$p_{\text{fault,1 million cyc}} =\cong \binom{10^6}{1}(10^{-25})(1 - 10^{-25})^{10^6-1} \cong 10^{-19}$$

$$p_{\text{single-fault,1 million cyc}} = \binom{32}{1}(10^{-19})^1(1 - 10^{-19})^{32-1} = 3.2 \times 10^{-18}$$

Note here that $p_{\text{single-fault}}$ increased by approximately 10^3 times, when the exposure time to particles decreased by 10^3 times. This demonstrates that the single-fault model effectively follows the Poisson distribution whose rate is constant over time, and the intrinsic FIT rate is indeed constant.

$$\text{FIT}\left[\frac{failures}{10^9\ hours}\right] = \frac{3.2 \times 10^{-18} \times 1\ [failures]}{10^6 \times 10^{-9}\ [seconds]} \times \frac{10^9 \times 3600\ [seconds]}{[10^9\ hours]} = 0.01152$$

Additionally, in case where the program reads and uses the word right before the program termination by, say, printing the word out to the user, i.e. at its 1 billion cycles, the FIT rate becomes:

$$\text{FIT}\left[\frac{failures}{10^9\ hours}\right] = \frac{3.2 \times 10^{-18} \times 1\ [failures]}{10^9 \times 10^{-9}\ [seconds]} \times \frac{10^9 \times 3600\ [seconds]}{[10^9\ hours]}$$
$$= 1.152 \times 10^{-5}$$

This demonstrates the AVF approach. Among 1 billion cycles, only the last 1 million cycles are affecting the program execution that is architecturally correct. Therefore, AVF = 1 billion / 1 million = 0.001 and the intrinsic FIT multiplied by AVF = 0.01152 × 0.001 = 1.152×10^{-5}.

(c) Parity detects that there are odd number of faulty bits in a word:

$$p_{\text{fault,1 million cyc}} =\cong \binom{10^6}{1}(10^{-25})(1-10^{-25})^{10^6-1} \cong 10^{-19}$$

$$p_{\text{single-fault,1 million cyc}} = \sum_{\forall \text{ odd } n \in [1,32]} \binom{32}{n}(10^{-19})^n(1-10^{-19})^{32-n}$$
$$\cong 3.2 \times 10^{-18}$$

$$\text{DUE FIT} = \frac{3.2 \times 10^{-18} \times 1}{10^6 \times 10^{-9}} \times 10^9 \times 3600 = 0.01152$$

Again, if the program reads the word right before the program termination,

$$\text{DUE FIT} = \frac{3.2 \times 10^{-18} \times 1}{10^9 \times 10^{-9}} \times 10^9 \times 3600 = 1.152 \times 10^{-5}$$

This shows that single bit parity effectively turns the above FIT rate to DUE FIT rate.

However, in order to measure the SDC FIT rate, we need to measure the FIT for having two, four, eight, …, 32 faulty bits:

$$p_{\text{single-fault,1 million cyc}} = \sum_{\forall \text{ even } n \in [1,32]} \binom{32}{n}(10^{-19})^n(1-10^{-19})^{32-n}$$
$$\cong 4.96 \times 10^{-36}$$

$$\text{SDC FIT} = \frac{4.96 \times 10^{-36} \times 1}{10^6 \times 10^{-9}} \times 10^9 \times 3600 = 1.7856 \times 10^{-23}$$

Again, if the program reads the word right before the program termination,

$$\text{SDC FIT} = \frac{4.96 \times 10^{-36} \times 1}{10^9 \times 10^{-9}} \times 10^9 \times 3600 = 1.7856 \times 10^{-26}$$

Up to now, we can see that AVF factoring on the intrinsic rate effectively works. Also remember that, in the above, there is no such thing as FALSE DUE failure or masked undetected fault since there is only one word in the cache.

(d) SECDED corrects if there is only one faulty bit, detects if there are two faulty bits and is incapable of correcting or detecting faults if there are more than two faulty bits.

$$p_{\text{2-bit fault,1 million cyc}} = \binom{32}{2}(10^{-19})^2(1-10^{-19})^{32-2} \cong 4.96 \times 10^{-36}$$

$$\text{DUE FIT} = \frac{4.96 \times 10^{-36} \times 1}{10^6 \times 10^{-9}} \times 10^9 \times 3600 = 1.7856 \times 10^{-23}$$

Again, if the program reads the word right before the program termination,

$$\text{DUE FIT} = \frac{4.96 \times 10^{-36} \times 1}{10^9 \times 10^{-9}} \times 10^9 \times 3600 = 1.7856 \times 10^{-26}$$

$$p_{\text{undetectable fault,1 million cyc}} = \sum_{n=3}^{32} \binom{32}{n} (10^{-19})^n (1-10^{-19})^{32-n}$$
$$\cong 4.96 \times 10^{-54}$$

$$\text{DUE FIT} = \frac{4.96 \times 10^{-54} \times 1}{10^6 \times 10^{-9}} \times 10^9 \times 3600 = 1.7856 \times 10^{-38}$$

Again, if the program reads the word right before the program termination,

$$\text{DUE FIT} = \frac{4.96 \times 10^{-36} \times 1}{10^9 \times 10^{-9}} \times 10^9 \times 3600 = 1.7856 \times 10^{-41}$$

(e) In this kind of circumstances, there are too many permutations for us to manually calculate the probabilities of all possible faulty patterns happened after 1 million cycles.

So, in order to make the problem feasible to solve by hand, let's assume that there would not be any overlap in the flipped bits due to any two or more particle hits. Since there is 70% chance that a particle would flip one bit and 30% chance that a particle would flip two bits, probabilities to have one bit, two bits and three bits flipped, during one million cycles are:

$$p(\text{1bit fault}) = p(\text{1 hit flipping only one bit})$$
$$= \binom{32}{1} (0.7 \times 10^{-19})^1 (1 - 0.7 \times 10^{-19})^{32-1} = 3.2 \times 10^{-18}$$
$$\cong 2.24 \times 10^{-18}$$

$$p(\text{2bit fault}) = p(\text{2 hits flipping only one bit}) + (\text{1 hit flipping two bits})$$
$$= \binom{32}{2} (0.7 \times 10^{-19})^2 (1 - 0.7 \times 10^{-19})^{32-2}$$
$$+ \binom{10^6}{1} (0.3 \times 10^{-25})(1 - 0.3 \times 10^{-25})^{10^6-1} \cong 9.6 \times 10^{-19}$$

$$p(\text{3bit fault}) = \{p(\text{2 hits flipping only one bit}) + (\text{1 hit flipping two bits})\}$$
$$\times p(\text{1 hit flipping only one bit}) \cong 9.6 \times 10^{-19} \times 2.24 \times 10^{-18}$$
$$= 2.1504 \times 10^{-36}$$

Thus SDC and DUE when parity or SECDED exists are:

$$\text{SDC FIT}_{\text{parity}} = \frac{3.2 \times 10^{-18} \times 1}{10^6 \times 10^{-9}} \times 10^9 \times 3600 = 0.0081$$

$$\text{DUE FIT}_{\text{parity}} = \frac{9.6 \times 10^{-19} \times 1}{10^6 \times 10^{-9}} \times 10^9 \times 3600 = 0.0035$$

$$\text{SDC FIT}_{\text{SECDED}} = \frac{9.6 \times 10^{-19} \times 1}{10^6 \times 10^{-9}} \times 10^9 \times 3600 = 0.0035$$

$$\text{DUE FIT}_{\text{SECDED}} = \frac{2.1504 \times 10^{-36} \times 10^{-19} \times 1}{10^6 \times 10^{-9}} \times 10^9 \times 3600 = 7.7414 \times 10^{-21}$$

Again, if the program reads the word right before the program termination,

$$\text{SDC FIT}_{\text{parity}} = \frac{3.2 \times 10^{-18} \times 1}{10^6 \times 10^{-9}} \times 10^9 \times 3600 = 8.1 \times 10^{-6}$$

$$\text{DUE FIT}_{\text{parity}} = \frac{9.6 \times 10^{-19} \times 1}{10^6 \times 10^{-9}} \times 10^9 \times 3600 = 3.5 \times 10^{-6}$$

$$\text{SDC FIT}_{\text{SECDED}} = \frac{9.6 \times 10^{-19} \times 1}{10^6 \times 10^{-9}} \times 10^9 \times 3600 = 3.5 \times 10^{-6}$$

$$\text{DUE FIT}_{\text{SECDED}} = \frac{2.1504 \times 10^{-36} \times 10^{-19} \times 1}{10^6 \times 10^{-9}} \times 10^9 \times 3600 = 7.7414 \times 10^{-24}$$

Problem 2.5

The MTTF due to EM is given by Black's equation as:

$$MTTF_{EM} = A \times w \times J^{-n} \times e^{\frac{E_a}{kT}} \qquad \text{(Eq. 2.1)}$$

where A is constant, w is cross sectional area, J is current density, E_a is activation energy, k is Boltzmann's constant, T is the operating temperature and n is an empirical scaling factor here equal to 2.

(a) If the wire radius is doubled then MTTF increases as thicker wire has less resistance. If the wire radius is doubled then the cross-sectional area of the wire increases by 4 times (it is Π^*r^2). The current density decreases by four times. So J^{-2} increases by 16 times. Remaining parameters are not effected. So the total MTTF increases by 64 times.

(b) Doubling the wire radius causes doubling of temperature. The improvement in MTTF is not 64X as the term $e^{(Ea/kT)}$ decreases. The exact change in MTTF can be determined by knowing Ea and T values.

CHAPTER 3

Problem 3.1

Opcode is 1 byte
Memory address is 2 byte
Register address is 0.5 byte.

Table 7: Sample assembly code

Accumulator	Stack	Memory-to-memory	Register-based
LOAD B	PUSH A	ADD A,A,B	LOAD R1,A
ADDA A	PUSH B	SUB C,A,C	LOAD R2, B
MOVA A	ADD	ADD C,C,D	LOAD R3,C
LOAD C	POP A		LOAD R4,D
NEGATE	PUSH C		ADD R1,R1,R2
ADDA A	PUSH A		SUB R3,R2,R3
ADDA D	SUB		ADD R3,R3,R4
MOVA C	PUSH D		STORE A,R1
	ADD		STORE C,R3
	POP C		

Accumulator:
The code has 8 instructions and one of them only contains an opcode. 7 instructions have both opcode and 1 memory address. Therefore the code size is 7*(1+2)+1 = 22 bytes.
The data memory traffic is 7*(4+2)=42 bytes including data operands and memory addresses.
The instruction traffic is the sum of the code size and the size of addresses. Hence, it is 22+8*2=38 bytes.
Stack:
The number of instructions is 10, and 7 instructions have 1 memory address.
Therefore, the code size is 7*(1+2) + 3*1 = 24 bytes.
The data memory traffic is 7*(4+2) = 42 bytes.
The instruction traffic is 24+ 10*2= 44 bytes.
Memory-to-memory:
All instructions have three memory addresses. The code size is 3*(1+3*2) = 21 bytes. The data memory traffic is 3*(3*4+3*2) = 48 bytes. The instruction traffic is 21+3*2= 27 bytes.
Register-based:
There are 9 instructions. 6 instructions access memory and they include an opcode, 1 register field and 1 memory address field. 3 instructions have an opcode and 3 register fields. Thus the code size is
6*3.5+3*2.5 = 28.5 bytes.
The data traffic is 6*(4+2)= 36 bytes.
The instruction traffic is 33+9*2= 51bytes.

Table 8 summarizes, the results

Table 8:

	Accumulator	Stack	Memory-to-memory	Register-based
code size (bytes)	22	24	21	28.5
data traffic (bytes)	42	42	48	36
instruction traffic (bytes)	38	44	27	51

Considering code size and instruction traffic, the memory-to-memory architecture is the most efficient one
In terms of data traffic, the register-base architecture is the most efficient architecture.

Problem 3.2

a. 1) Little endian
32-bit word at address 1000H

Table 9: 32-bit word

MSB			LSB
ABH	32H	F7H	23H

Answer: -1422723293

16-bit half word at address 1000H

Table 10: 16-bit half word

MSB	LSB
F7H	23H

Answer: -2269

16 bit half word at address 1002H

Table 11: 16-bit half word

MSB	LSB
ABH	32H

Answer: -21710

2) Big endian
32-bit word at address 1000H

Table 12: 32-bit word

MSB			LSB
23H	F7H	32H	ABH

Answer: 603402923

16-bit half word at address 1000H

Table 13: 16-bit half word

MSB	LSB
23H	F7H

Answer: 9207

16 bit half word at address 1002H

Table 14: 16-bit half word

MSB	LSB
32H	ABH

Answer: 12971

b.

Table 15: ASCII codes

	G	O	(SPACE)	T	R	O	J	A	N	S	!
ASCII	47H	4FH	20H	54H	52H	4FH	4AH	41H	4EH	53H	21H

Both conventions yield the same memory byte content. This is because we only access bytes. The difference between little and big endian only manifests itself when items bigger than a byte are accessed.
1000H: 47H
1001H: 4FH
1002H: 20H
1003H: 54H

Problem 3.3

The value in R0 is always zero. S and i are stored in memory at locations 2000H and 3000H, respectively.

a.
```
                ADDI R1,R0, #2000 // base address of S
                ADDI R2,R0, #3000 // base address of i
                ADDI R3,R0, #1000 // base address of A[0]
      LOOP:     LW R4,0(R3) // A[i]
                LW R5,0(R1) // S
                LW R6,0(R2) //i
                ADD R5,R5,R4
                SW R5,0(R1)
                ADDI R3,R3,#4
                ADDI R6,#1 // i <- i+1
                SW R6,0(R2)
                SLTI R7,R6,#100
                BNEZ R7,LOOP
```
Estimated execution time:
Instructions before the start of the loop take 3 cycles.
Each iteration except for the last one takes (2+2+2+1+1+1+1+1+1+3)=15 cycles, for a total of 99 iterations. The last loop iteration takes 13 cycles. Therefore the estimated execution time is 3+99*15+13 = 1501 cycles.

b.
```
                ADD R1,R0,R0 // S =0;
                ADDI R2,R0,#100
                ADDI R3,R0,#1000
                ADDI R5,R0,#2000
      LOOP:     LW R4,0(R3) // R4<- mem[R3]
                ADD R1,R1,R4 // S=S+A[i]
                ADDI R3,R3,#4
                SUBI R2,R2,#1
                BNEZ R2,LOOP
                SW R1,0(R5)
```
Estimated execution time:
To initialize variables before the loop starts, 4 ALU instructions execute in 4 cycles.
For 99 iterations, each iteration takes (2+1+1+1+3) =8 cycles. The last iteration takes (2+1+1+1+1) = 6 cycles.
Finally, the store instruction takes 1 cycle.
Therefore, estimated execution time is 4 + 99*8 + 6+1 = 803 cycles.

Problem 3.4

a. In the machine with no forwarding at all instructions must wait for their inputs until they have been written back to register. It takes 3 clocks for the result of an instruction in EX to exit WB. Thus the latency of operation of all instructions writing in a register is 3 clocks and the compiler must insert NOOPs accordingly.
```
      SEARCH:   LW R5,0(R3)        /I1  Load item
                NOOP
                NOOP
                NOOP
                SUB R6,R5,R2       /I2  Compare with key
                NOOP
                NOOP
                NOOP
                BNEZ R6,NOMATCH    /I3  Check for match
                ADDI R1,R1,#1      /I4  Count matches
      NOMATCH:  ADDI R3,R3,#4      /I5  Next item
                NOOP
                NOOP
```

```
                    NOOP
                    BNE R4,R3,SEARCH   /I6  Continue until all items
```

b.Without forwarding but with a hazard detection unit in ID, an instruction which is dependent on a prior instruction should be stalled in ID until the result from the prior instruction is written back to register. The latency of operation of all instructions is 3 just as in part a of this problem.

b.1) First assume a match.

```
      SEARCH:     LW R5,0(R3)        (1)
                  SUB R6,R5,R2       (4)
                  BNEZ R6,NOMATCH    (4)
                  ADDI R1,R1,#1      (1)
      NOMATCH:    ADDI R3,R3,#4      (1)
                  BNE R4,R3,SEARCH   (4+2)
```

The numbers between parentheses are the number of cycles associated with each instruction. Each instruction spends one cycle in the ID stage plus additional stall cycles due to the latency of prior instructions it depends on. In the case of a branch, the number of instructions flushed when the branch is taken must be added. For example, BNE spends 4 cycles in ID due to data dependency and 2 additional cycles are lost due to flushing when the branch is taken in EX.
Summing up all delays, it takes (1+4+4+1+1+4+2)= 17 cycles to execute one iteration of the loop.

b.2) On no match, BNEZ (I3) is taken.

```
      SEARCH:     LW R5,0(R3)        (1)
                  SUB R6,R5,R2       (4)
                  BNEZ R6,NOMATCH    (4+2)
      NOMATCH:    ADDI R3,R3,#4      (1)
                  BNE R4,R3,SEARCH   (4+2)
```

After the BNEZ instruction is taken in ID, 2 instructions in IF and ID are flushed.
It takes 18 cycles to execute one iteration of the loop.

c. With register forwarding and a hazard detection unit, an instruction which is waiting for its operand from a previous instruction in ID can get the value when the previous instruction reaches WB. Therefore, the latency of operation of instructions writing to a register is 2 instead of 3.

c.1) On a match:

```
      SEARCH:     LW R5,0(R3)        (1)
                  SUB R6,R5,R2       (3)
                  BNEZ R6,NOMATCH    (3)
                  ADDI R1,R1,#1      (1)
      NOMATCH:    ADDI R3,R3,#4      (1)
                  BNE R4,R3,SEARCH   (3+2)
```

It takes 14 cycles to execute one iteration of the loop.

c.2) On no match:

```
      SEARCH:     LW R5,0(R3)        (1)
                  SUB R6,R5,R2       (3)
                  BNEZ R6,NOMATCH    (3+2)
      NOMATCH:    ADDI R3,R3,#4      (1)
                  BNE R4,R3,SEARCH   (3+2)
```

It takes 15 cycles to execute one iteration of the loop.

d. With full forwarding and a hazard detection unit, the latency of all instructions is 0 except for loads, and so instructions in the code have no stall in ID except for the SUB instruction.

d.1) On a match

```
      SEARCH:    LW R5,0(R3)        (1)
                 SUB R6,R5,R2       (2)
                 BNEZ R6,NOMATCH    (1)
                 ADDI R1,R1,#1      (1)
      NOMATCH:   ADDI R3,R3,#4      (1)
                 BNE R4,R3,SEARCH   (1+2)
```

It takes 9 cycles to execute one iteration of the loop.

d.2) On no match

```
      SEARCH:    LW R5,0(R3)        (1)
                 SUB R6,R5,R2       (2)
                 BNEZ R6,NOMATCH    (1+2)
      NOMATCH:   ADDI R3,R3,#4      (1)
                 BNE R4,R3,SEARCH   (1+2)
```

It takes 10 cycles to execute one iteration of the loop.

e. A basic block cannot have a branch or jump except at its bottom. Moreover, no instruction inside of it can be the target of a branch or a jump. Hence, there are 3 basic blocks in the given code (assuming no other instruction branches into the code).
Basic block 1: I1, I2, and I3
Basic block 2: I4
Basic block 3: I5 and I6

It is not possible to save cycles by local optimizations because the sizes of basic blocks are too small and we do not have enough instructions for local optimizations. In general it is not safe to move an instruction outside of its basic block. However, the program would still execute correctly if I5 was moved before BNEZ because I5 is always executed, whether or not the branch is successful. However global optimizations will be necessary to enable that move.

f. Yes. For example the LW and SUB for two consecutive execution of the loop can be moved up after register renaming, thus avoiding the stall on the SUB instruction. However to be able to do that global scheduling is needed because the LW and SUB must be moved up across a branch instruction.

With a delayed branch instruction the pointer to the next item could be updated in the delay slot of I3, thus saving one cycle in the case of no match.

```
      SEARCH:    LW R5,0(R3)        /I1  Load item
                 SUB R6,R5,R2       /I2  Compare with key
                 DBNEZ R6,NOMATCH   /I3  Check for match
                 ADDI R3,R3,#4      /I5  Next item
                 ADDI R1,R1,#1      /I4  Count matches
      NOMATCH:   BNE R4,R3,SEARCH   /I6  Continue until all items
```

Problem 3.5.

a. The goal is to balance the functions in the pipeline stages so that the delays in all stages are as close as possible. One possible partition is shown in Figure 1.

b.
Instruction latency in non-pipelined implementation: PC delay(1ns) + ICache(6ns) + Itype Decode(3.5ns) + Src. Decode(2.5ns) + Reg. Read(4ns) + Mux(1ns) + ALU(6ns) + Reg. Write(4ns) = 28 ns.
In the pipelined implementation, there are 5stages and each stage takes 7.5ns (the bottleneck is the

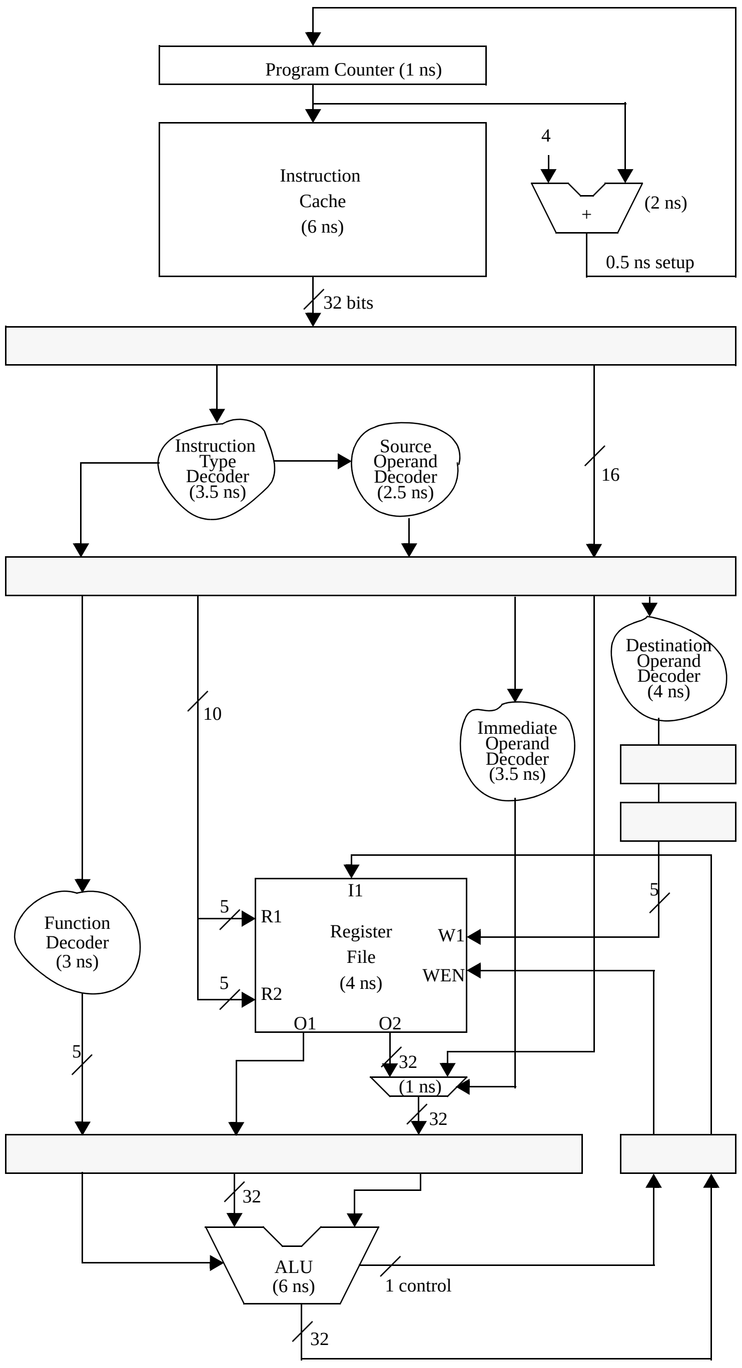


Figure 1. Pipelined implementation

I-fetch stage). Therefore the instruction latency is 5*7.5ns cycle time = 37.5ns.

c. Non-pipelined: machine cycle = instruction latency = 28ns.
Pipelined cycle time: 7.5ns

d. Potential speedup is not 5x, because of additional overhead from the pipeline registers and because the stages are not perfectly balanced. The non-pipelined solution finishes an instruction once every 28ns. The pipelined solution finishes an instruction once every 7.5ns. The speedup is 28/7.5 = 3.73.

e. The main bottleneck is the I-fetch stage (7.5ns), then the decode and ALU stages (6.5ns). Example of techniques are: faster instruction cache implementations, predecode bits in the instruction cache to reduce the decode time, faster ALU implementation, or superpipelining.

Problem 3.6.

a. In this case there are 4 stages after the decode stage (EX, ME1, ME2, and WB). With no forwarding at all, the latency of operation of all instructions with register result is 4. Thus the compiler must insert 4 NOOPs between the source and the sink of a dependency.

```
SEARCH:     LW R5,0(R3)        /I1  Load item
            NOOP
            NOOP
            NOOP
            NOOP
            SUB R6,R5,R2       /I2  Compare with key
            NOOP
            NOOP
            NOOP
            NOOP
            BNEZ R6,NOMATCH    /I3  Check for match
            ADDI R1,R1,#1      /I4  Count matches
NOMATCH:    ADDI R3,R3,#4      /I5  Next item
            NOOP
            NOOP
            NOOP
            NOOP
            BNE R4,R3,SEARCH   /I6  Continue until all items
```

b. No forwarding at all, but a hazard detection unit in ID.
The latency of operation of all instructions with register result is again 4. When a branch is taken 3 stages (IF1, IF2, and ID) must be flushed.

b.1) First assume a match. BNEZ is not taken--no flushing

```
SEARCH:     LW R5,0(R3)        (1)
            SUB R6,R5,R2       (5)
            BNEZ R6,NOMATCH    (5)
            ADDI R1,R1,#1      (1)
NOMATCH:    ADDI R3,R3,#4      (1)
            BNE R4,R3,SEARCH   (5+3)
```

It takes 21 cycles for one iteration of the loop.

b.2) On no match, BNEZ is taken--3 stages are flushed

```
SEARCH:     LW R5,0(R3)        (1)
            SUB R6,R5,R2       (5)
            BNEZ R6,NOMATCH    (5+3)
NOMATCH:    ADDI R3,R3,#4      (1)
            BNE R4,R3,SEARCH   (5+3)
```

It takes 23 cycles to execute one iteration of the loop.

c.Register forwarding and a hazard detection unit in ID.
With register forwarding, instructions stalled in ID can get their register value from a prior instruction when it reaches WB. Therefore, the latency of operation of an instruction producing a register

result is now 3. Taken branch penalty is unchanged.

c.1) On a match, BNEZ is not taken

```
      SEARCH:    LW R5,0(R3)       (1)
                 SUB R6,R5,R2      (4)
                 BNEZ R6,NOMATCH   (4)
                 ADDI R1,R1,#1     (1)
      NOMATCH:   ADDI R3,R3,#4     (1)
                 BNE R4,R3,SEARCH  (4+3)
```

It takes 18 cycles to execute one iteration of the loop

c.2) On no match, BNEZ is taken

```
      SEARCH:    LW R5,0(R3)       (1)
                 SUB R6,R5,R2      (4)
                 BNEZ R6,NOMATCH   (4+3)
      NOMATCH:   ADDI R3,R3,#4     (1)
                 BNE R4,R3,SEARCH  (4+3)
```

It takes 20 cycles to execute one iteration of the loop.

d. With full forwarding and a hazard detection unit in ID, the operation latency of reg-to-reg operations is 0 and the latency of loads is 2.
Because it takes two cycles to access memory, an instruction which is dependent on a load stalls for 2 cycles in ID.

d.1) On a match

```
      SEARCH:    LW R5,0(R3)       (1)
                 SUB R6,R5,R2      (3)
                 BNEZ R6,NOMATCH   (1)
                 ADDI R1,R1,#1     (1)
      NOMATCH:   ADDI R3,R3,#4     (1)
                 BNE R4,R3,SEARCH  (1+3)
```

It takes 11 cycles to execute one iteration of the loop.

d.2) On no match

```
      SEARCH:    LW R5,0(R3)       (1)
                 SUBI R6,R5,R2     (3)
                 BNEZ R6,NOMATCH   (1+3)
      NOMATCH:   ADDI R3,R3,#4     (1)
                 BNE R4,R3,SEARCH  (1+3)
```

One iteration of the loop takes 13 cycles.

Problem 3.7

a. To show that the design with branch predicted always taken is a good choice, we need to compare the penalty of a branch instruction in both cases (predicted always taken vs. predicted always untaken.)

In the design with branch predicted untaken, no cycle is lost due to a branch if the branch is untaken, but, if a branch is taken, 2 cycles are lost.

In the design with branch predicted always taken, 1 cycle is lost on every branch instruction because the instruction in IF is always flushed when the branch is in ID. Additionally, if a branch is untaken, instructions in IF and ID are flushed. Note that, in this case, the instruction in ID has already been flushed, so we should not count it twice (actually it is sufficient to flush the IF stage in this case). Therefore, if the branch is taken 1 cycle is lost and if the branch is untaken 2 cycles are lost.

Let f be the fraction of branches that should be taken so that the new scheme is better. Comparing the average branch penalties of the two designs, the fraction f must be such that

$$f+2(1-f) < 2f \text{ or } f > 2/3$$

Note that in actual programs this condition is often met since branches are more often taken than not.

b. Let's assume again that the fraction of taken branches is f. f is now considered a constant. Given f we want to compute X, the fraction of compiler predictions that are dynamically correct, so the compiler can improve on always taken/always untaken predictions. The possibilities are enumerated in Table 16.

Table 16:

Type of branch	Compiler prediction	Outcome	Fraction	Penalty (performance)	Penalty (energy)
Taken	Taken	success	fX	1	1
Taken	Untaken	misprediction	f(1-X)	2	3
Untaken	Taken	misprediction	(1-f)(1-X)	2	3
Untaken	Untaken	sucess	(1-f)X	0	0

Comparing with always taken: we must have (refer to a above)
$f+2(1-f) > fX + 2f(1-X) + 2(1-f)(1-X)$ or
$2-f > fX + 2(1-X)$ or
$2-f > 2-(2-f)X$ or
$X > f/(2-f)$
This should not be very hard for the compiler since the fraction of taken branches f is often more than 0.5. For example, if f=0.7, the success rate of the compiler prediction must be more than 0.7/1.3~0.54, just above a 50/50 guess!

Comparing with always untaken, we must have:
$2f > 2-(2-f)X$ or
$X > (2-2f)/(2-f)$
For example, if f = 0.7, the success rate of the compiler prediction must be more than 0.6/1.3~0.46. Thus even if the compiler prediction is worse than flipping a fair coin. This is because always predicting untaken is a very poor prediction in this case as it succeeds only 30% of the time.
If the compiler sets a hint bit even with very small fraction of the branch time, the compiler's prediction is better than always predicting untaken.

c. Assume that 1 unit of energy is spent in each pipeline stage in every cycle even if it is a NOOP. Then the total amount of energy spent for the execution of IC instructions with perfect branch handling is 5IC.

• Always predicted untaken
If the branch is untaken (probability1-f), then no additional energy is wasted. If the branch is taken (probability f), the instructions in IF and ID must be flushed. Flushed instructions must still go through the pipeline. Thus the energy penalty is 10f. We can also generalize this energy penalty as (Number of flushed instructions) * (pipeline depth). Let b be the fraction of branches. Then the total energy consumed is (5+10bf)IC. As compared to the pipeline with perfect branch handling, the rel-

ative energy consumption is (5+10bf)IC/(5IC) = 1+2bf.
• Always predicted taken
When a branch is taken five units of energy are wasted. When a branch is untaken, ten units of energy are wasted. So the total energy consumed is (5+5bf+10b(1-f))IC= (5+10b-5bf)IC and the relative energy consumption is (5+10b-5bf)/5 = 1+2b-bf
• With compiler hints and X=0.95
The total energy is: (5+0.95*5bf+0.05*10bf+0.05*10b(1-f))IC = (5+4.75bf+0.5b)IC and the relative energy consumption is (5+4.75bf+0.5b)/5 = 1+0.95bf+0.1b.

Problem 3.8

a. RAW hazards.
Because we have full forwarding, an integer instruction in ID can get an operand from EX/ME or ME/WB(int). However, if a load is in EX, a dependent integer instruction in ID must stall. Integer instructions get their operands from integer registers, and so they do not check FP pipeline registers. If an FP arithmetic instruction is in ID, we need to compare source registers to the destination of each pipeline registers in FP.
-- If an integer arithmetic/logic/store instruction or load is in ID, check ID/EX for a load
-- If an FP arithmetic instruction is in ID, check ID/EX (for FP loads), ID/FP, FP1/FP2, FP2/FP3, and FP3/FP4.
-- If an FP store is in ID, check ID/EX (for loads returning address register or FP value), FP1/FP2, FP2/FP3, and FP3/FP4.

b. For WAW hazards,
-- If an integer arithmetic/logic/store or integer load is in ID: No check is necessary, since these instructions update integer registers only and follow the integer pipeline path in process order.
-- If an FP load is in ID, check ID/FP, and FP1/FP2 and FP2/FP3 (no need to check FP3/FP4 because an FP load and an FP arithmetic instruction cannot reach the FP register file in the same cycle. This is done in ID to prevent structural hazards on the write port of the FP register file.)
-- If an FP arithmetic instruction is in ID, there is no need to check any pipeline stage because FP arithmetic instructions cannot bypass a previous instruction.
-- If an FP store is in ID, there is no need to check any pipeline stage because stores do not write value in registers.

Problem 3.9

a. For RAW hazards,
-- If an integer arithmetic/logic/store instruction or a load is in ID, check ID/EX1 (for integer instructions and loads), EX1/EX2 (for loads) and EX2/ME1 (for loads).
-- If an FP arithmetic instruction is in ID, check ID/EX1 (for FP loads), EX1/EX2 (for FP loads), EX2/ME1 (for FP loads), ID/FP, FP1/FP2, FP2/FP3 and FP3/FP4.
-- If an FP store instruction is in ID, check ID/EX1 (for loads returning address or FP value), EX/EX2 (for loads returning address or FP value), EX2/ME1(for loads returning address or FP value), ID/FP, FP1/FP2, FP2/FP3, and FP3/FP4.
b. for WAW hazards,
-- If an integer arithmetic/logic/store instruction or integer load is in ID, no check is necessary (as all integer instructions follow the same path)
-- If an FP load is in ID, check ID/FP1 and FP1/FP2. There is no need to check FP2/FP3 because logic in ID prevents two write to the same register file in the same clock.
-- If an FP arithmetic instruction is in ID, no check is necessary because FP arithmetic instructions take the longest path and all other preceding instructions are always completed before an FP arith-

metic instruction in ID.
-- If an FP store is in ID, no check is necessary because stores do not write to registers.

Problem 3.10

a. First, note that only one value (integer or FP) may be propagated in any one stage of the pipeline. For example we cannot have one instruction in EX and one instruction in FP1 in the same clock. This is because instructions are issued one at a time.

We consider values forwarded to EX first. Because integer instructions also traverse FP pipeline stages FP3, FP4 and FP5, values may be forwarded from their pipeline register to the EX stage.
EX/ME --> EX
ME/FP3 --> EX (this also include FP values forwarded by FP loads to FP stores)
FP3/FP4 --> EX
FP4/FP5 --> EX
FP5/WB - ->EX

Next we consider values forwarded to FP1.
ME/FP3 --> FP1
FP5/WB-->FP1
In this solution, we simplify by not providing a forwarding path from FP loads in FP4 and in FP5 to FP1. Rather we stall. One could also forward FP load values (ONLY) from FP3/FP4 and FP4/FP5 to avoid stalling on them.

b. Remaining RAW hazards
-- If an integer arithmetic/logic/store/load instruction is in ID, check ID/EX (for preceding loads)
-- If an FP arithmetic instruction is in ID, check ID/EX (for preceding loads), ID/FP1, FP1/FP2, FP2/FP3, FP3/FP4.
-- If an FP store instruction is in ID, check ID/EX (for loads returning address or FP value), ID/FP1, FP1/FP2, FP2/FP3, FP3/FP4 (for FP values).

c. Stores update machine state early (this is particularly bad because stores update memory). One simple solution is to stall stores in ID until it is determined that no previous instruction currently in the pipeline can trigger an exception.

Problem 3.11

a. In ID2 the two instructions have no dependency and are a correct pair (with possibly one of them a NOOP, which is fine). Each can execute as soon as they don't have any data hazard (RAW or WAW) with previous instructions currently in the pipeline. To respect process order, the lower instruction cannot start execution before the upper instruction. So the HDU associated with ID2 simply checks for hazards with instructions currently in the pipeline for both instructions in ID2 (in the scalar processor checks were made for a single instruction, now it is two, that's the only difference).

b. A branch is resolved in EX. Because branches are always predicted untaken, all the following instructions are in IF, ID1, ID2 and FP1. Thus, when a branch is taken, we need to flush the instructions in all these stages (and no more), a total of 7 instructions.

c. We show the code as it will be presented to ID2 by ID1. Instructions between parenthesis are not instructions in the code, rather they have been "inserted" by ID1. In the right most column, the number of cycles associated with each instruction pair in ID2 is shown (1 cycle in ID2 plus the stalls due to data dependencies based on the operation latencies in Figure 2). In the case of the taken branch we add 3 cycles because IF, ID1 and ID2 are flushed.

```
             Upper ID2              Lower ID2                 Cycles
LOOP         L.D F2,0(R1)           (NOOP)                     (1)
             (NOOP)                 ADD.D F4,F2,F4             (2)
             L.D F6, -8(R1)         (NOOP)                     (1)
             (NOOP)                 ADD.D F8,F6,F4             (4)
             S.D F8, 0(R1) (4)      (NOOP)                     (5)
             SUBI R1,R1,16(1)       (NOOP)                     (1)
             BNEZ R1, LOOP(3)       (NOOP)                     (1+3)
```

The execution time of one iteration of the loop is 18 cycles

For the machine in Exercise 3.8, the numbers between parenthesis are the number of cycles in ID and branch penalties due to flushing.

```
LOOP         L.D F2,0(R1)   (1)
             ADD.D F4,F2,F4 (2)
             L.D F6, -8(R1) (1)
             ADD.D F8,F6,F4 (4)
             S.D F8, 0(R1) (5)
             SUBI R1,R1,16 (1)
             BNEZ R1, LOOP (1+2)
```

Therefore, it takes 17 cycles for one iteration of the loop.

For the machine in Exercise 3.9, the latency of operation of a Load is now 3. The operation latency of a reg-to-reg instruction is now 1. The operation latency of FP arithmetic instruction is unchanged (4). A branch is resolved in EX1 and 3 stages must be flushed when it is taken.

```
LOOP         L.D F2,0(R1)   (1)
             ADD.D F4,F2,F4 (4)
             L.D F6, -8(R1) (1)
             ADD.D F8,F6,F4 (4)
             S.D F8, 0(R1)  (5)
             SUBI R1,R1,16  (1)
             BNEZ R1, LOOP  (2+3)
```

Therefore each iteration takes 21 cycles. However the clock rate can be raised by pipelining bottle-neck stages (such as IF, EX and ME).

Conclusion: In terms of cycles, no machine has a clear advantage. The superpipeline has an advantage if it can be clocked faster. The superscalar pipeline is penalized because the code does not use the two pipelines effectively.

Problem 3.12

a. The three penalties (data, branch, and memory) measured in cycles increase proportionally to the number of stages, assuming that the increase in stages is spread evenly among the 5 fundamental functions of the 5 stage pipeline --IF,ID,EX,ME, and WB. This may be seen as a gross approximation, but we are not looking for an exact value, just a trend.

b. The CPI of the pipeline with K stages is given by:

$$CPI(K) = 1 + (\alpha_d + 2\alpha_b + \alpha_m)\frac{K}{5} = 1 + A \times K$$

The average execution time of one instruction is equal to

$$CPI(K) \times T_K = (1 + A \times K)\left(\frac{T}{K} + t_l\right)$$

Note that the right side of this equality has two factors: the first one (CPI) increases with K because of hazards, whereas the second one (clock cycle) decreases with K. In practical cases, the second factor is dominant, thus a minimum must be reached as K increases

The instruction throughput is the inverse of the average instruction execution time, i.e.,

$$\frac{1}{CPI(K) \times T_K} = \frac{1}{(1 + A \times K)\left(\frac{T}{K} + t_l\right)}$$

c. The optimum instruction throughput is obtained for a value of K that minimizes the average execution time. To find this optimum pipeline depth, we take the derivative of the instruction execution time.

$$\frac{\partial}{\partial K}\left((1 + A \times K)\left(\frac{T}{K} + t_l\right)\right) = \frac{\partial}{\partial K}\left(\frac{T}{K} + t_l + AT + At_lK\right) = -\frac{T}{K^2} + t_l \times A$$

When $\frac{\partial}{\partial K}\left((1 + A \times K)\left(\frac{T}{K} + t_l\right)\right) = 0$, K is the optimum value to minimize execution time.

$$-\frac{T}{K^2} + t_l \times A = 0$$

Therefore,

$$K_{optimum} = \sqrt{\frac{T}{t_l \times A}} = \sqrt{\frac{5T}{t_l(\alpha_d + 2\alpha_b + \alpha_m)}}$$

d. Using the answer obtained in part (c), the optimum number of stages is:

$$K_{optimum} = \sqrt{\frac{5 \times 10 \times 10^{-9}}{100{\times}10^{-12}(0.2 + 2 \times 0.06 + 0.5)}} = 24.6$$

, where α_d =0.2,α_b = 0.06,α_m=0.2, T=10nsec and t_l=100ps.
The optimum pipeline depth K is 25.

The optimum instruction throughput (in MIPS) is:

$$\frac{10^{-6}}{\left(1 + \frac{(0.2 + 0.12 + 0.5)}{5} \times 25\right)\left(\frac{10{\times}10^{-9}}{31} + 100{\times}10^{-12}\right)} = \frac{1}{(1 + 4.1)(0.323{\times}10^{-3} + 0.1{\times}10^{-3})} = 463MIPS$$

To compare this optimum pipeline with the5-stage pipeline, the instruction throughput of 5-stage pipeline is computed as:

$$\frac{10^{-6}}{\left(1 + \frac{(0.2 + 0.12 + 0.5)}{5} \times 5\right)\left(\frac{10{\times}10^{-9}}{5} + 100{\times}10^{-12}\right)} = \frac{1}{(1 + 0.82)(2{\times}10^{-3} + 0.1{\times}10^{-3})} = 262MIPS$$

The 25-stage pipeline is better than 5-stage pipeline by a factor of 1.77 (=463/262).
Thus, in this example, with these parameters, the optimum pipeline depth is much more than 5, but the potential speedup of superpipelining the 5-stage pipeline is quite small.

Problem 3.13

a. Tomasulo algorithm--no speculation.

Table 17: Tomasulo algorithm--no speculation

		Dispatch	Issue	Exec start	Exec complete	Cache	CDB	COMMENTS
I1	L.D F0,0(R1)	1	2	(3)	3	(4)	(5)	
I2	L.D F2,0(R2)	2	3	(4)	4	(5)	(6)	
I3	L.D F4,0(R3)	3	4	(5)	5	(6)	(7)	
I4	MUL.D F6,F2,F0	4	7	(8)	12	--	(13)	wait for F2
I5	ADD.D F8,F6,F4	5	14	(15)	19	--	(20)	wait for F6
I6	ADDI R1,R1,#8	6	7	(8)	8	--	(9)	
I7	ADDI R2,R2,#8	7	8	(9)	9	--	(10)	
I8	ADDI R3,R3,#8	8	9	(10)	10	--	(11)	
I9	S.S-A F8,-8(R2)	9	11	(12)	12	--	--	wait for R2
I10	S.S-D F8,-8(R2)	10	21	(22)	22	(23)	--	wait for F8
I11	BNE R4,R2,LOOP	11	12	(13)	13	--	(14)	
I12	L.D F0,0(R1)	15	16	(17)	17	(18)	(19)	wait for I11 in dispatch

The values between parentheses show resources reserved by an instruction at the time of dispatch. Store I10 can issue to cache at clock 23 because no memory instruction is pending in the L/S queue. Load I12 can issue to cache at clock 18 because the only preceding and pending memory instruction is the store I10 and the address of the store is known at the end of clock 12. Load I12 cannot dispatch until it knows the outcome of branch I11, at the end of clock 14.

b. Tomasulo algorithm with speculation.

Table 18: Tomasulo algorithm with speculation

		Dispatch	Issue	Exec start	Exec complete	Cache	CDB	Retire	COMMENTS
I1	L.D F0,0(R1)	1	2	(3)	3	(4)	(5)	6	
I2	L.D F2,0(R2)	2	3	(4)	4	(5)	(6)	7	
I3	L.D F4,0(R3)	3	4	(5)	5	(6)	(7)	8	
I4	MULT.D F6,F2,F0	4	7	(8)	12	--	(13)	14	wait for F2
I5	ADD.D F8,F6,F4	5	14	(15)	19	--	(20)	21	wait for F6
I6	ADDI R1,R1,#8	6	7	(8)	8	--	(9)	22	
I7	ADDI R2,R2,#8	7	8	(9)	9	--	(10)	23	
I8	ADDI R3,R3,#8	8	9	(10)	10	--	(11)	24	
I9	S.D-A F8,-8(R2)	9	11	(12)	12	--	--	--	wait for R2
I10	S.D-D F8,-8(R2)	10	21	(22)	22	24	--	25	wait for F8, then wait to reach top of ROB
I11	BNE R4,R2,LOOP	11	12	(13)	13	--	(14)	26	
I12	L.D F0,0(R1)	12	13	(14)	14	(15)	(16)	27	Address of store is known since cycle 13

Store I10 reaches the L/S queue at the end of clock 22. To proceed to cache it must verify that it is at the top of the ROB. The previous instruction in process order is I8 and it retires at clock 24. Therefore the store is at the top of the ROB during clock 24 and it may issue to cache in clock 24. Note that now load I12 can dispatch before it knows the outcome of branch I11.

c. Speculation with speculative scheduling

Table 19: Tomasulo algorithm with speculation and speculative scheduling

		Dispatch	Issue	Register Fetch	Exec start	Exec complete	Cache	CDB	Retire	Comments
I1	L.D F0,0(R1)	1	2	3	(4)	4	(5)	(6)	7	
I2	L.D F2,0(R2)	2	3	4	(5)	5	(6)	(7)	8	
I3	L.D F4,0(R3)	3	4	5	(6)	6	(7)	(8)	9	
I4	MULT.D F6,F2,F0	4	5	6	(7)	11	--	(12)	13	
I5	ADD.D F8,F6,F4	5	10	11	(12)	16	--	(17)	18	wait for F6
I6	ADDI R1,R1,#8	6	7	8	(9)	9	--	(10)	19	
I7	ADDI R2,R2,#8	7	8	9	(10)	10	--	(11)	20	
I8	ADDI R3,R3,#8	8	10	11	(12)	12	--	(13)	21	CDB conflict with I4
I9	S.D-A F8,-8(R2)	9	10	11	(12)	12	--	--	--	
I10	S.D-D F8,-8(R2)	10	15	16	17	17	21	--	22	wait for F8, then wait to reach top of ROB
I11	BNE R4,R2,LOOP	11	12	13	(14)	14	--	(15)	23	
I12	L.D F0,0(R1)	12	14	15	(16)	16	(17)	(18)	24	CDB conflict with I5

Because of speculative scheduling the MULT instruction I4 does not have to wait for F2 anymore because F2 is forwarded right on time through the common data bus in clock 7. By issuing I4 ahead of time, the forwarded F2 value is the input to the FP unit at the beginning of cycle 7. Instruction I8 cannot issue in clock 9, although it has no hazards with prior instructions because it cannot reserve the CDB for clock 12 (because it was reserved at clock 5 by I4). Store I10 waits for F8 to issue. F8 is put on the CDB in clock 17 and the store can execute in the AGU in clock 17 and is issued speculatively at clock 15. Then the store reaches the L/S queue where it waits to reach the top of the ROB, in clock 21. At the beginning of clock 21 the store is at the top of the ROB because the preceding instruction (I8) is in the retirement unit in clock 21.

d.

Each instruction in the data-flow graph is labeled as follows.

```
I1         L.D F0,0(R1)          /X[i] loaded in F0
I2         L.D F2,0(R2)          /Y[i] loaded in F2
I3         L.D F4,0(R3)          /Z[i] loaded in F4
I4         MUL.D F6,F2,F0        /Multiply X by Y
I5         ADD.D F8,F6,F4         /Add Z
I6         ADDI R1,R1,#8        /Update address registers
I7         ADDI R2,R2,#8
I8         ADDI R3,R3,#8
I9         S.D F8, -8(R2)       /store in Y[i]
I10        BNE R4,R2,LOOP       /(R4)-8 points to the last element of Y
```

The data-flow graph is given in Figure 2. The critical path in the data-flow graph is shown in bold (I1->I4->I5). The store I9 is off the critical path because there is no RAW memory dependencies on memory, either within one iteration or across iterations of the loop.

Table 20: Execution time comparisons

Execution time metric	Tomasulo w/o spec	Tomasulo with spec	Spec scheduling	Data-flow
Issue-to-issue	16-2=14	13-2=11	14-2=12	7
Execution-to-execution	17-3=14	14-3=11	15-3=12	7
Retirement	---	27-6=21	24-7=17	7

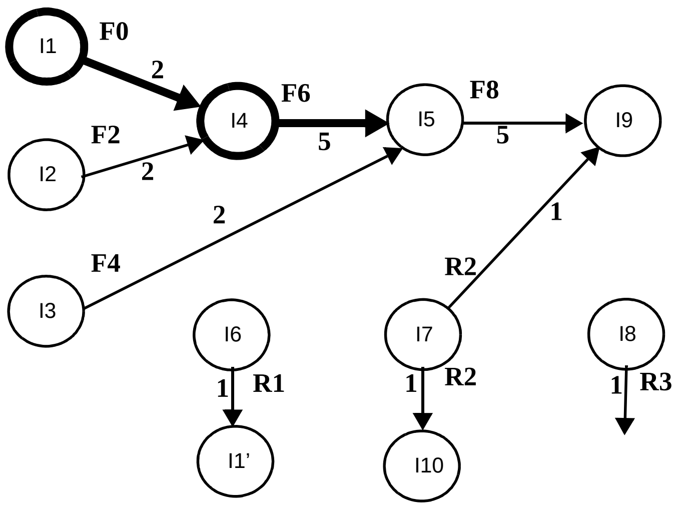


Figure 2Data-flow graph

It is a challenge to figure out the exact execution rate in a single iteration of a loop. The exercise asks to measure execution time from issue to issue of the first load. The results are shown in the first row of Table 20. Speculative execution is better than execution with no speculation. This is because of the delay needed to obtain the branch outcome. For this particular code speculative execution with speculative scheduling is slightly worse than without speculative scheduling. Looking at the schedule speculative scheduling for this particular code causes two conflicts on the CDB.
Table 20 also shows two other possible measures. The first one is between the starts of execution of the two loads. The numbers are unchanged. The second measure is the time between retirement of the two loads. This metric may be better because it only considers instructions that are non-speculative. In this case the store is included in the critical path. Under this metric, speculative scheduling improves on speculative execution without speculative scheduling.

Problem 3.14

Table 21: Tomasulo algorithm with speculation (two-way superscalar)

		Dispatch	Issue	Exec start	Exec complete	Cache	CDB	Retire	COMMENTS
I1	L.D F0,0(R1)	1(7)	2	(3)	3	(4)	(5)	6	
I2	L.D F2,0(R2)	1(6)	3	(4)	4	(5)	(6)	7	FU conflict with I1
I3	L.D F4,0(R3)	2(5)	4	(5)	5	(6)	(7)	8	FU conflict with I2
I4	MULT.D F6,F2,F0	2(4)	7	(8)	12	--	(13)	14	wait for F2
I5	ADD.D F8,F6,F4	3(3)	14	(15)	19	--	(20)	21	wait for F6
I6	ADDI R1,R1,#8	3(2)	6	(7)	7	--	(8)	22	CDB conflict with I2 & I3
I7	ADDI R2,R2,#8	4(1)	7	(8)	8	--	(9)	23	CDB conflict with I3 and FU conflict with I6
I8	ADDI R3,R3,#8	4(0)	8	(9)	9	--	(10)	24	CDB conflict with I3 and FU conflict with I6 & I7
I9	S.D-A F8,-8(R2)	6(1)	10	(11)	11	--	--	--	wait to get 1 ROB entry, then wait for R2
I10	S.D-D F8,-8(R2)	6(0)	21	(22)	22	(24)	--	25	wait to get 1 ROB entry, wait for F8, then to reach ROB top
I11	BNE R4,R2,LOOP	8(1)	10	(11)	11	--	(12)	26	wait to get 2 ROB entries, then wait for R2
I12	L.D F0,0(R1)	8(0)	11	(12)	12	(13)	(14)	27	wait to get 2 ROB entries, and CDB conflict with I11&I4
I13	L.D F2,0(R2)	21(1)	22	(23)	23	(24)	(25)	28	wait to get 2 ROB entries
I14	L.D F4,0(R3)	21(0)	23	(24)	24	(25)	(26)	29	wait to get 2 ROB entries, and con-flict with I13
I15	MULT.D F6,F2,F0	23(1)	26	(27)	31	--	(32)	33	wait to get 2 ROB entries, then wait for F2
I16	ADD.D F8,F6,F4	23(0)	33	(34)	38	--	(39)	40	wait to get 2 ROB entries, then wait for F6
I17	ADDI R1,R1,#8	25(1)	26	(27)	27	--	(28)	41	wait to get 2 ROB entries
I18	ADDI R2,R2,#8	25(0)	27	(28)	28	--	(29)	42	wait to get 2 ROB entries, then FU conflict with I17
I19	ADDI R3,R3,#8	26(0)	28	(29)	29	--	(30)	43	wait to get 1 ROB entry, then FU conflict with I18
I20	S.D-A F8,-8(R2)	26(0)	30	(31)	31	--	--	--	wait to get 1 ROB entry and then wait for R2
I21	S.D-D F8,-8(R2)	28(1)	40	(41)	41	(43)	--	44	wait to get 2 ROB entries, wait for F8, then wait to reach top of ROB

		Dispatch	Issue	Exec start	Exec complete	Cache	CDB	Retire	COMMENTS
I22	BNE R4,R2,LOOP	28(0)	31	(32)	32	--	(33)	45	wait to get 2 ROB entries, wait for R2, and CDB conflict with I15
I23	L.D F0,0(R1)	33(1)	34	(35)	35	(36)	(37)	46	wait to get 2 ROB entries
I24	L.D F2,0(R2)	33(0)	35	(36)	36	(37)	(38)	47	wait to get 2 ROB entries and conflict with I23

Execution Time per Iteration (issue-to-issue; second iteration) = 34 - 11 = 23 cycles. Comparing this to the execution time in part b (single dispatch) of Exercise 3.13 (11 cycles), we find that dual dispatch degrades the performance significantly. The main bottleneck is the number of ROB entries available, compounded by the fact that the machine waits until two instructions are structural hazard free to dispatch. This machine definitely needs more ROB entries.
Also, since there are a lot of data dependencies between instructions, the machine stalls instructions in issue queues while waiting for their operands and this reduces the benefits of dual dispatch. The last bottleneck is structural hazards on functional units (ALU's, cache ports, FP units), and on the single CDB.

Problem 3.15

a. Conservative Disambiguation:

Table 22: Tomasulo algorithm with speculation (two-way superscalar)-Conservative disambiguation

		Dispatch	Issue	Exec start	Exec complete	Cache	CDB	Retire	Comment
I1	L.D F2,0(R1)	1(7)	2	(3)	3	(4)	(5)	6	
I2	ADDI R1,R1,#8	1(6)	2	(3)	3	--	(4)	7	
I3	ADDI R2,R2,#8	2(5)	4	(5)	5	--	6	8	CDB conflict with I1
I4	S.D-A F2,-8(R2)	2(5)	7	(8)	8	--	--	--	wait for R2
I5	S.D-D F2,-8(R2)	3(4)	6	(7)	7	9	--	10	wait for F2 and wait for address
I6	BNEQ R1,R3,LOOP	3(3)	5	(6)	6	--	(7)	11	wait for R1
I7	L.D F2,0(R1)	4(2)	5	(6)	6	10	11	12	wait for address of previous store and CDB conflict with I9
I8	ADDI R1,R1,#8	4(1)	7	(8)	8	--	(9)	13	FU conflict with I6 and CDB conflict with I7
I9	ADDI R2,R2,#8	5(0)	8	(9)	9	--	(10)	14	CDB conflict with I7and FU conflict with I8
I10	S.D-A F2,-8(R2)	5(0)	11	(12)	12	--	--	--	wait for R2
I11	S.D-D F2,-8(R2)	7(1)	12	(13)	13	(14)	--	15	wait to get 2 ROB entries, wait for F2
I12	BNEQ R1,R3,LOOP	7(0)	10	(11)	11	--	(12)	16	wait for R1
I13	L.D F2,0(R1)	10(1)	13	(14)	14	(15)	(16)	17	wait to get 2 ROB entries and AGU conflict with I10 & I11
I14	ADDI R1,R1,#8	10(0)	11	(12)	12	--	(13)	18	wait to get 2 ROB entries

Table 22: Tomasulo algorithm with speculation (two-way superscalar)-Conservative disambiguation

		Dispatch	Issue	Exec start	Exec complete	Cache	CDB	Retire	Comment
I15	ADDI R2,R2,#8	11(0)	12	(13)	13	--	(14)	19	
I16	S.D-A F2,-8(R2)	11(0)	15	(16)	16	--	--	--	wait for R2
I17	S.D-D F2,-8(R2)	13(1)	17	(18)	18	(19)	--	20	wait to get 2 ROB entries, wait for F2
I18	BNEQ R1,R3,LOOP	13(0)	15	(16)	16	--	(17)	21	wait to get 2 ROB entries and CDB conflict with I13
I19	L.D F2,0(R1)	15(1)	16	(17)	17	(18)	(19)	22	wait to get 2 ROB entries

At the time of issue, load I7 reserved the cache for clock 7 and the CDB for clock 8. However, the load must wait in the L/S queue until the address of the previous store is known, at the end of clock 8. However if the load issues to cache in clock 9, it will conflict with I9 at clock 10. Therefore I7 must wait one more clock to issue to cache, at clock 10. Meanwhile I8 cannot issue at clock 5 because I6 has already reserved the integer unit for clock 6. It also cannot issue at clock 6 because load I7 has reserved the CDB for clock 8 (a false conflict and a CDB cycle that is lost). Similarly I9 fails to issue at first in clock 6 because of the CDB conflict with I7. Then it cannot issue at clock 7 because of the conflict with I8 on the integer unit.

b. Speculative Disambiguation:

Table 23: Tomasulo algorithm with speculation (two-way superscalar)-Speculative disambiguation

		Dispatch	Issue	Exec start	Exec complete	Cache	CDB	Retire	Comment
I1	L.D F2,0(R1)	1(7)	2	3	3	4	5	6	
I2	ADDI R1,R1,#8	1(6)	2	3	3	--	4	7	
I3	ADDI R2,R2,#8	2(5)	4	5	5	--	6	8	CDB conflict with I1
I4	S.D-A F2,-8(R2)	2(5)	7	8	8	--	--	--	wait for R2
I5	S.D-D F2,-8(R2)	3(4)	6	7	7	9	--	10	wait for F2 and wait for address
I6	BNEQ R1,R3,LOOP	3(3)	5	6	6	--	7	11	wait for R1
I7	L.D F2,0(R1)	4(2)	5	6	6	7	8	12	
I8	ADDI R1,R1,#8	4(1)	7	8	8	--	9	13	FU conflict with I6 and CDB conflict with I7
I9	ADDI R2,R2,#8	5(0)	8	9	9	--	10	14	CDB conflict with I7 and FU conflict with I8
I10	S.D-A F2,-8(R2)	5(0)	11	12	12	--	--	--	wait for R2
I11	S.D-D F2,-8(R2)	7(1)	9	10	10	14	--	15	wait to get 2 ROB entries, wait for F2, then wait to reach top of ROB
I12	BNEQ R1,R3,LOOP	7(0)	10	11	11	--	12	16	wait to get 2 ROB entries then wait for R1
I13	L.D F2,0(R1)	10(1)	12	13	13	14	15	17	wait to get 2 ROB entries and AGU conflict with I10
I14	ADDI R1,R1,#8	10(0)	11	12	12	--	13	18	wait to get 2 ROB entries
I15	ADDI R2,R2,#8	11(0)	12	13	13	--	14	19	
I16	S.D-A F2,-8(R2)	11(0)	15	16	16	--	--	--	wait for R2

Table 23: Tomasulo algorithm with speculation (two-way superscalar)-Speculative disambiguation

		Dispatch	Issue	Exec start	Exec complete	Cache	CDB	Retire	Comment
I17	S.D-D F2,-8(R2)	13(1)	16	17	17	19	--	20	wait to get 2 ROB entries, then wait for F2
I18	BNEQ R1,R3,LOOP	13(0)	14	15	15	--	16	21	wait to get 2 ROB entries
I19	L.D F2,0(R1)	15(1)	17	18	18	19	20	22	wait to get 2 ROB entries and AGU conflict with I17

Looking again at load I7, the load is (successively) issued to cache speculatively at clock 7. This avoids load delays, and eliminates the wasted cycle on the CDB.

Problem 3.16

a. With one-bit state machine, 0 is not taken (NT) and 1 is taken (T).
The values are: 8,9,10,11,7,20,29,30,31

b1:	T,U,T,U,U,T,U,T,U
b2:	T,T,U,T,T,U,T,U,T

Table 24: 1-bit predictor

Iteration	1	2	3	4	5	6	7	8	9
B1(Prediction/Actual Branch Direction)	0/1	1/0	0/1	1/0	**0/0**	0/1	1/0	0/1	1/0
B2(Prediction/Actual Branch Direction)	0/1	**1/1**	1/0	0/1	**1/1**	1/0	0/1	1/0	0/1

If a prediction and the actual branch direction are the same in a given entry of Table 24, it means that the branch prediction is correct, and these cases are in bold in the table above.
Also the branch direction in the iteration is the prediction in the next iteration.
Therefore, the prediction accuracy for b1 is 1/9 = 11%
The prediction accuracy for b2 is 2/9 = 22%.
The overall prediction accuracy for both branches is 3/18 = 1/6 = 16.67%

b. 2-level branch prediction scheme.
If the previous branch, either b1 or b2, is taken, g is set to 1. If the previous branch is not taken, then g is 0 in the history register. If a prediction and an actual branch direction are same, they are shown in bold in Table 25.
Thus, the prediction accuracy for b1 is 4/9 = 44.4%.
The prediction accuracy for b2 is also 6/9=66.7%.
The overall accuracy is 55.5%.

c.
The prediction success rate for b2 when g=0 is 4/5 = 80%.
For b2, g is equal to 0 when the previous branch, b1, is not taken. That means the value is odd. In the given 9 values, there is no odd number which is also a multiple of 5. Therefore, once the branch

predictor is warmed-up, b2 branch with g=0 is highly predictable.

Table 25: Two-level branch prediction scheme

Iteration	1	2	3	4	5	6	7	8	9
B1 (g/Prediction/ Actual Branch Direction)	0/0/1	**1/0/0**	1/0/1	0/1/0	1/1/0	1/0/1	**0/0/0**	**1/1/1**	**0/0/0**
B2 (g/Prediction/ Actual Branch Direction)	1/0/1	0/0/1	1/1/0	**0/1/1**	**0/1/1**	**1/0/0**	**0/1/1**	**1/0/0**	**0/1/1**

The values are: 8,9,10,11,7,20,29,30,31

b1:	T,U,T,U,U,T,U,T,U
b2:	T,T,U,T,T,U,T,U,T

Problem 3.17

First, let's think about the possible aliasing for private table access. Branch1 and Branch2 are separated by 1 instruction and their PC values differ by 8. Branch2 and Bbranch3 are separated by 2 instructions and their PC values differ by 12. Branch1 and Branch3 are separated by 4 instructions and their PC values differ by 20. Since 10 bits of the branch PC are used, there is no possibility of aliasing of branch addresses for the private tables for this piece of code.

1. GAg
The history pattern is global and the predictors are shared. In the following table, the first column indicates the loop iteration and the 2nd column indicates branch. The pattern column shows the global history and the value of the predictor. The action column shows the prediction, the actual branch action, and the updated predictor value.

Table 26:

Iteration	Branch	Pattern	Action
1	BNEZ R2, LAB1	0/0	NT/NT/0 (Success)
1	BEQZ R0, LAB2	0/0	NT/T/1
1	BNEZ R1, LOOP	1/0	NT/T/1
2	BNEZ R2, LAB1	1/1	T/T/1 (Success)
2	BNEZ R1, LOOP	1/1	T/T/1 (Success)
3	BNEZ R2, LAB1	1/1	T/NT/0
3	BEQZ R0, LAB2	0/1	T/T/1 (Success)
3	BNEZ R1, LOOP	1/0	NT/T/1
4	BNEZ R2, LAB1	1/1	T/T/1 (Success)
4	BNEZ R1, LOOP	1/1	T/T/1 (Success)
5	BNEZ R2, LAB1	1/1	T/NT/0
5	BEQZ R0, LAB2	0/1	T/T/1 (Success)

Table 26:

Iteration	Branch	Pattern	Action
5	BNEZ R1, LOOP	1/0	NT/T/1
6	BNEZ R2, LAB1	1/1	T/T/1 (Success)
6	BNEZ R1, LOOP	1/1	T/T/1 (Success)
...	...	...	...
999	BNEZ R2, LAB1	1/1	T/NT/0
999	BEQZ R0, LAB2	0/1	T/T/1 (Success)
999	BNEZ R1, LOOP	1/0	NT/T/1
1000	BNEZ R2, LAB1	1/1	T/T/1 (Success)
1000	BNEZ R1, LOOP	1/1	T/NT/0

The misprediction pattern repeats itself from the 3rd iteration on.
We have 2 mispredictions in iterations 1 and 2.
From the 3rd iteration to the 998th iteration, for every two iterations, we have 2 mispredictions. In the last 1000th iteration, we have one misprediction.

Therefore, the total misprediction count is (1000/2) *2 +1 = 1001 among 5*1000/2 =2500 branches.
Misprediction rate =1001/2500 = 40%

2. GAp
The history is global, but the predictors are private.

Table 27:

Iteration	Branch	Pattern	Action
1	BNEZ R2, LAB1	0/0	NT/NT/0 (Success)
1	BEQZ R0, LAB2	0/0	NT/T/1
1	BNEZ R1, LOOP	1/0	NT/T/1
2	BNEZ R2, LAB1	1/0	NT/T/1
2	BNEZ R1, LOOP	1/1	T/T/1 (Success)
3	BNEZ R2, LAB1	1/1	T/NT/0
3	BEQZ R0, LAB2	0/1	T/T/1 (Success)
3	BNEZ R1, LOOP	1/1	T/T/1 (Success)
4	BNEZ R2, LAB1	1/0	NT/T/1
4	BNEZ R1, LOOP	1/1	T/T/1 (Success)
5	BNEZ R2, LAB1	1/1	T/NT/0
5	BEQZ R0, LAB2	0/1	T/T/1 (Success)

Table 27:

Iteration	Branch	Pattern	Action
5	BNEZ R1, LOOP	1/1	T/T/1 (Success)
6	BNEZ R2, LAB1	1/0	NT/T/1
6	BNEZ R1, LOOP	1/1	T/T/1 (Success)
...	...		
999	BNEZ R2, LAB1	1/1	T/NT/0
999	BEQZ R0, LAB2	0/1	T/T/1 (Success)
999	BNEZ R1, LOOP	1/1	T/T/1 (Success)
1000	BNEZ R2, LAB1	1/0	NT/T/1
1000	BNEZ R1, LOOP	1/1	T/NT/0

The misprediction pattern repeats itself from the 3rd iteration on.
We have 3 mispredictions in iteration 1 and 2.
From the 3rd iteration to the 998th iteration, for every two iterations we have two mispredictions. In the 999th and 1000th iterations, we have three mispredictions.

Therefore, the total misprediction count is 1+ (1000/2) *2 +1 = 1002 among 5*1000/2 =2500 branches.

The misprediction rate is 1002/2500 = 40%

3. PAg
The history pattern is private but the predictors are shared.

Table 28:

Iteration	Branch	Pattern	Action
1	BNEZ R2, LAB1	0/0	NT/NT/0 (Success)
1	BEQZ R0, LAB2	0/0	NT/T/1
1	BNEZ R1, LOOP	0/1	T/T/1 (Success)
2	BNEZ R2, LAB1	0/1	T/T/1 (Success)
2	BNEZ R1, LOOP	1/0	NT/T/1
3	BNEZ R2, LAB1	1/1	T/NT/0
3	BEQZ R0, LAB2	1/0	NT/T/1
3	BNEZ R1, LOOP	1/1	T/T/1 (Success)
4	BNEZ R2, LAB1	0/1	T/T/1(Success)
4	BNEZ R1, LOOP	1/1	T/T/1 (Success)
5	BNEZ R2, LAB1	1/1	T/NT/0

Table 28:

Iteration	Branch	Pattern	Action
5	BEQZ R0, LAB2	1/0	NT/T/1
5	BNEZ R1, LOOP	1/1	T/T/1 (Success)
6	BNEZ R2, LAB1	0/1	T/T/1 (Success)
6	BNEZ R1, LOOP	1/1	T/T/1 (Success)
...	...		
999	BNEZ R2, LAB1	1/1	T/NT/0
999	BEQZ R0, LAB2	1/0	NT/T/1
999	BNEZ R1, LOOP	1/1	T/T/1 (Success)
1000	BNEZ R2, LAB1	0/1	T/T/1(Success)
1000	BNEZ R1, LOOP	1/1	T/NT/0

The misprediction pattern repeats itself from the 3rd iteration on.
We have two mispredictions in iterations 1 and 2.
From the 3rd iteration to the 999th iteration, for every two iterations we have two mispredictions. In the 999th and 1000th iterations, we have three mispredictions.

Therefore, the total misprediction count is (1000/2) *2 +1 = 1001 among 5*1000/2 =2500 branches.
Misprediction rate: 1001/2500 = 40%

4. PAp
The history pattern and the predictors are both private.

Table 29:

Iteration	Branch	Pattern	Action
1	BNEZ R2, LAB1	0/0	NT/NT/0 (Success)
1	BEQZ R0, LAB2	0/0	NT/T/1
1	BNEZ R1, LOOP	0/0	NT/T/1
2	BNEZ R2, LAB1	0/0	NT/T/1
2	BNEZ R1, LOOP	1/0	NT/T/1
3	BNEZ R2, LAB1	1/0	NT/NT/0 (Success)
3	BEQZ R0, LAB2	1/0	NT/T/1
3	BNEZ R1, LOOP	1/1	T/T/1 (Success)
4	BNEZ R2, LAB1	0/1	T/T/1 (Success)
4	BNEZ R1, LOOP	1/1	T/T/1 (Success)
5	BNEZ R2, LAB1	1/0	NT/NT/0 (Success)

Table 29:

Iteration	Branch	Pattern	Action
5	BEQZ R0, LAB2	1/1	T/T/1 (Success)
5	BNEZ R1, LOOP	1/1	T/T/1 (Success)
6	BNEZ R2, LAB1	0/1	T/T/1 (Success)
6	BNEZ R1, LOOP	1/1	T/T/1 (Success)
7	BNEZ R2, LAB1	1/0	NT/NT/0 (Success)
7	BEQZ R0, LAB2	1/1	T/T/1 (Success)
7	BNEZ R1, LOOP	1/1	T/T/1 (Success)
8	BNEZ R2, LAB1	0/1	T/T/1 (Success)
8	BNEZ R1, LOOP	1/1	T/T/1 (Success)
...	...	...	...
999	BNEZ R2, LAB1	1/0	NT/NT/0 (Success)
999	BEQZ R0, LAB2	1/1	T/T/1 (Success)
999	BNEZ R1, LOOP	1/1	T/T/1 (Success)
1000	BNEZ R2, LAB1	0/1	T/T/1 (Success)
1000	BNEZ R1, LOOP	1/1	T/NT/0

We only have five mispredictions in iterations 1 to 4, and one misprediction in the last iteration.

Thus, the total misprediction count is six among 2500 branches.
The misprediction rate is 6/2500 = 0.24%

Problem 3.18

a.Simple 1-bit predictor (no history).

The sequences of bits are as follows for the input string of bits, the string of predictions bits (0 for b=0 and 1 for b=1), and the string of correct predictions (these sequences are repeated:

b0b1b2..	prediction bits	Correct predictions(0:wrong;1:correct)
...000... :	...000...	...111...
...001... :	...100...	...010...
...010... :	...001...	...100...
...011... :	...101...	...001...
...100... :	...010... .	...001...
...101... :	...110...	...100...
...110... :	...011...	...010...
...111... :	...111...	...111...

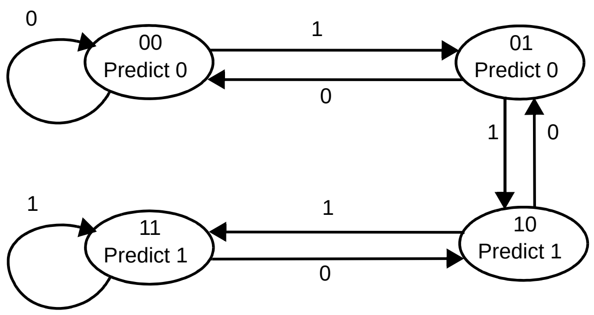
The prediction bits are the previous bit in the input bit sequence. Their pattern is periodic as well.

The misprediction rate for each input bit sequence is therefore:

b_0,b_1,b_2=>	000	001	010	011	100	101	110	111
misprediction rate(%) =>	0	66.6	66.6	66.6	66.6	66.6	66.6	0

Assuming that all sequences of bits b0b1b2 are equally likely, the average misprediction rate is (2/3 * 6) / 8 = 0.5 = 50%

b. Simple 2-bit predictor (saturating counter; no history):



The sequences of bits are as follows:

b0b1b2	prediction counters	predictions	Correct predictions(0:wrong)
...000... :	...[00][00][00]...	...000...	...111...
...001... :	...[01][00][00]...	...000...	...110...
...010... :	...[00][00][01]...	...000...	...101...
...011... :	...[11][10][11]...	...111...	...011...
...100... :	...[00][01][00]...	...000...	...011...
...101... :	...[11][11][10]...	...111...	...101...
...110... :	...[10][11][11]...	...111...	...110...
...111... :	...[11][11][11]...	...111...	...111...

The prediction counters simply count the number of 1's (+1) and 0's (-1) in the input bit sequence. Their value sequence is periodic.

The misprediction rate for each input bit sequence is therefore:

b_0,b_1,b_2=>	000	001	010	011	100	101	110	111
misprediction rate(%) =>	0	33.3	33.3	33.3	33.3	33.3	33.3	0

Assuming that all patterns are equally likely, the average misprediction rate is:
(1/3 * 6) / 8 = 0.25 = 25%

c. 2-bit predictor with 2 bits of global history:

b0b1b2	global history bits	prediction ctrs	prediction	Correct
...000... :	...[00][00][00]...	...[00][00][00]...	...000...	...111...
...001... :	...[01][10][00]...	...[00][00][11]...	...001...	...111...

...010... :	...[10][00][01]...	...[00][11][00]...	...010...	...111...
...011... :	...[11][10][01]...	...[00][11][11]...	...011...	...111...
...100... :	...[00][01][10]...	...[11][00][00]...	...100...	...111...
...101... :	...[01][11][10]...	...[11][00][11]...	...101...	...111...
...110... :	...[10][01][11]...	...[11][11][00]...	...110...	...111...
...111... :	...[11][11][11]...	...[11][11][11]...	...111..	...111...

History bits are the previous two bits in the input bit sequence, and their sequence is periodic. The prediction ctrs are the 2-bit counters associated with each history bit pair in the global history bit column. Their values are also periodic.

The misprediction rate for each input bit sequence is therefore:

b_0,b_1,b_2=>	000	001	010	011	100	101	110	111
misprediction rate(%) =>	0	0	0	0	0	0	0	0

The prediction of this predictor is perfect for all possible sequences of input bits.
The reason that this predictor is so successful is because the input sequence is periodic and the period is three bits. With two bits of history the next bit can always be reliably predicted because the value of that bit is always the same, thus the 2-bit counter is consistently either incremented or decremented. This simple GAp predictor is very successful at that.

d. Design a predictor which will predict the branch testing b with 100% accuracy.

The answer to this question is trivial since the GAp predictor above already achieves this.

Problem 3.19

a. Recall that all instructions in a basic block must be executed whenever any one of them is executed. Thus, a basic block can only have a branch at the bottom of the block and can only be the target of a branch at the top of the block.
We scan the code from beginning to end. At the beginning, three instructions (Lines 1~3) are always executed whenever one of them is executed. However, the instruction at line 4 is the target of a jump, Therefore, the instruction at line 4 must be the start of a new basic block. Because the instruction at line 4 is also a branch, the next instruction at line 5 must start a new basic blocks. Basic block 3 consists of instructions at line 5 and 6 because they are both executed whenever one of them is executed. The instruction at line 7 is a basic block by itself for the same reason as for the instruction at line 4. And so on...To identify basic block the compiler scans the code from top to bottom and every time an instruction must be executed whenever the previous instruction is executed, the instruction becomes a part of the current basic block. Otherwise, the instruction starts a new basic block.

b.

Table 30: . Branch behavior

Branch Instruction Number	Branch Instruction Execution Number									
	1	2	3	4	5	6	7	8	9	10
4	N	N	T							
7	N	N	N	N	T	N	N	N	N	T
11	N	T	T	N	T	T	N	N		
16	T	T	T	T	T	T	T	T		
17	T	T								
18	T									

Table 31: . Branch execution statistics

Branch Instruction Number	Times Executed	Times Taken	Times Not Taken	% Taken	% Not Taken
4	3	1	2	33.3	66.7
7	10	2	8	20	80
11	8	4	4	50	50
16	8	8	0	100	0
17	2	2	0	100	0
18	1	1	0	100	0

Average branch penalty for unconditional and conditional branches:
Unconditional branch penalty is 1 cycle. Conditional branch penalty is 2 cycles when the branch is taken (branch always predicted untaken). With the information obtained in the table above, we can compute the average branch penalty. When branches are always predicted untaken, the branch penalty for each branch in the code is:
C. Branch 4: 2*0.333=0.67
C. Branch 7: 2*0.2 = 0.4
C. Branch 11 = 2*0.5 =1
U. Branch 16: 1
U. Branch 17: 1
U. Branch 18: 1
The average penalty for branches is (0.67+0.4+1+1+1+1)/6= 0.85 cycles
Number of cycles needed to execute the trace the trace?
In the case where conditional branches are always predicted untaken, the branch penalty is 2 cycles only when the branch is taken. The total number of cycles is computed as follows:
Instruction Count (83) + Data Dependency Stalls (8 cycles from BB5) + Unconditional branch Stalls (11= 8 cycles from BB7 +2 cycles form BB8 + 1 cycle from BB9) + Conditional branch Stalls (14 = 2 cycles from BB2 + 4 cycles from BB4 + 8 cycles from BB5)+ Drain Cycles(4) = 120 cycles

Number of cycles lost to control dependency stalls?
Unconditional branch stall + Conditional branch stalls = 11+ 14 = 25 cycles

c.Static Branch Prediction
Branch instruction 4: BEQN   R5, R0, end
Branch instruction 7: BEQN   R4, R3, cont
Branch instruction 11: BEQZN R8, skip

Because we need to flush the instructions in IF and ID only when the branch is taken by predicting not taken, predicting not taken is better if the fractions of branch taken and not taken are equal. Therefore, BEQZN is selected for branch instruction 11.

Cycle count:
Instruction Count (83) + Data Dependency Stalls (8 cycles from BB5) + Unconditional Branch Stalls (11= 8 cycles from BB7 +2 cycles form BB8 + 1 cycle from BB9) + Conditional Branch Stalls (14 = 2 cycles from BB2 + 4 cycles from BB4 + 8 cycles from BB5)+ Drain Cycles(4) = 120 cycles

IPC: 83/120 = 0.69

Problem 3.20

a. Local Scheduling.
The best code after local scheduling is:

```
      LOOP        L.D F2,0(R1)
                  L.D F6, -8(R1)
                  SUBI R1,R1,#16
                  ADD.D F4,F2,F4
                  ADD.D F8,F6,F4
                  S.D F8, 16(R1)
                  BNEZ R1, LOOP
```

Dynamically, this code will be scheduled as follows in ID2.
Assume branches are always predicted not-taken and are resolved in the EX stage. Therefore, the branch penalty is 3 cycles. This branch penalty is included between parentheses after the BNEZ instruction in the following schedule.

```
                  Upper ID2               Lower ID2               Cycles
      LOOP        L.D F2,0(R1)                (NOOP)              (1)
                  L.D F6, -8(R1)              (NOOP)              (1)
                  SUBI R1,R1,#16          ADD.D F4,F2,F4          (1)
                      (NOOP)              ADD.D F8,F6,F4          (5)
                  S.D F8, 16(R1)              (NOOP)              (5)
                  BNEZ R1, LOOP               (NOOP)              (1+3)
```

After local scheduling, it takes 17 cycles for one iteration of the loop. The execution time of the original loop is 18 cycles based on the answer to problem 8 of homework 2. Thus, the speedup over the original loop on the superscalar architecture is 18/17 = 1.06

b. Loop Unrolling
We unroll the loop twice, rename and schedule the code to maximize the effectiveness of the super-

scalar machine.

Table 32: Loop unrolling

Unroll twice	Rename FP registers	Schedule
L.D F2,0(R1)	L.D F2,0(R1)	L.D F2,0(R1)
ADD.D F4,F2,F4	ADD.D F4,F2,F4	L.D F6,-8(R1)
L.D F6,-8(R1)	L.D F6,-8(R1)	L.D F10,-16(R1)
ADD.D F8,F6,F4	ADD.D F8,F6,F4	ADD.D F4,F2,F4
S.D F8,0(R1)	S.D F8,0(R1)	L.D F12,-24(R1)
L.D F2,-16(R1)	L.D F10,-16(R1)	ADD.D F8,F6,F4
ADD.D F4,F2,F4	ADD.D F4,F10,F4	SUBI R1,R1,#32
L.D F6,-24(R1)	L.D F12,-24(R1)	ADD.D F4,F10,F4
ADD.D F8,F6,F4	ADD.D F14,F12,F4	S.D F8,32(R1)
S.D F8,-16(R1)	S.D F14,-16(R1)	ADD.D F14,F12,F4
SUBI R1,R1,#32	SUBI R1,R1,#32	S.D F14,16(R1)
BNEZ R1, LOOP	BNEZ R1,LOOP	BNEZ R1,LOOP

We do not rename F4 because of the loop-carried dependency on F4.
The code will be scheduled as follows in ID2.

```
           Upper ID2            Lower ID2           Cycles
Loop:      L.D F2,0(R1))        (NOOP)              (1)
           L.D F6,-8(R1)        (NOOP)              (1)
           L.D F10,-16(R1)      ADD.D F4,F2,F4      (1)
           L.D F12,-24(R1)      ADD.D F8,F6,F4      (5)
           SUBI R1,R1,#32       ADD.D F4,F10,F4     (1)
           S.D F8,32(R1)        ADD.D F14,F12,F4    (5)
           S.D F14,16(R1)       (NOOP)              (5)
           BNEZ R1,LOOP         (NOOP)              (1+3)
```

In line 6 of the schedule S.D F8, 32(R1) wait on F8 for 3 cycles in ID2 until ADD.D F8,F6,F4 finishes its execution while ADD.D F16, F14,F4 must wait for two additional cycles for the value of F4. Thus ID2 waits for 5 cycles total in ID2.
After unrolling the loop twice, it takes 23 cycles for two iterations of the loop. The effective length of one iteration is 23/2 = 11.5 cycles. Thus, the speedup over the original loop is 18/11.5= 1.56.

Can the compiler garner more speedup by unrolling more? Why?
Yes, however the gains are marginal and hard to get. Unrolling is subject to Amdahl's law. Because of the dependencies between the ADD.D instructions the minimum execution time is 6 clocks and the maximum speedup is 3. We need 1 clock between the first L.D and the first ADD.D, 4 clocks between the last ADD.D and the last S.D and 3 clocks for the stage flush caused by the branch. Let N be the number of unrolls then the execution time of the unrolled loop is 6N+(2+5+4) = 6N+11 and the speedup is 18N/(6N+11). There is also a limit on the number of registers. Each additional unroll requires 3 double-precision registers and we only have 16 such registers. So we can unroll 5 times, for a maximum speedup of 2.2.

c. Software pipelining

Table 33: Software pipelining

	Original Iteration 1	Original Iteration 2	Original Iteration 3	Original Iteration 4
Prologue	L.D F2,0(R1)			
	ADD.D F4,F2,F4			
	L.D F6,-8(R1)	L.D F2,-16(R1)		
	ADD.D F8,F6,F4	ADD.D F4,F2,F4		
Pipelined iteration 1	**S.D F8, 0(R1)**	**L.D F6,-24(R1)**	**L.D F2,-32(R1)**	
		ADD.D F8,F6,F4	**ADD.D F4,F2,F4**	
Pipelined iteration 2		**S.D F8, -16(R1)**	**L.D F6,-40(R1)**	**L.D F2,-48(R1)**
			ADD.D F8,F6,F4	**ADD.D F4,F2,F4**
Epilogue			S.D F8, -32(R1)	L.D F6,-56(R1)
				ADD.D F8,F6,F4
				S.D F8, -48(R1)

Pipelined Code:

```
Prologue:  L.D F2,0(R1)
           ADD.D F4,F2,F4
           L.D F6,-8(R1)
           ADD.D F8,F6,F4
           L.D F2,-16(R1)
           ADD.D F4,F2,F4
           ADD.D F8,F6,F4
Loop:      L.D F6,-24(R1)            / iteration i+1
           L.D F2, -32(R1)           / iteration i+2
           S.D F8, 0(R1)             / iteration i+0
           ADD.D F8,F6,F4            / iteration i+1
           SUBI R1,R1,#16
           ADD.D F4,F2,F4            / iteration i+2
           BNEZ R1,LOOP
Epilogue:  S.D F8, 0(R1)
           L.D F6, -24(R1)
           ADD.D F8,F6,F8
           S.D F8,-16(R1)
```

The instructions in the loop body are scheduled as follows (after applying local scheduling optimizations on the pipelined code):

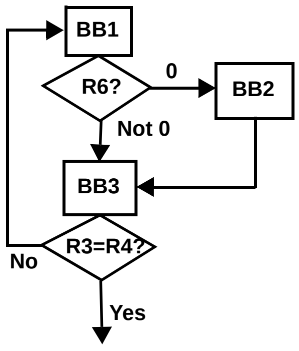
```
           Upper ID2                 Lower ID2              Cycles
Loop:      L.D F6,-24(R1)              (NOOP)               (1)
           L.D F2,-32(R1)              (NOOP)               (1)
           S.D F8, 0(R1)             ADD.D F8,F6,F4         (1)
           SUBI R1,R1,#16            ADD.D F4,F2,F4         (1)
           BNEZ R1,LOOP                (NOOP)               (1+3)
```

Although S.D stalls in the first iteration of the loop due to data dependency on F8, for following iterations the value on F8 is ready when SD is in ID2 because of the branch penalty.
It takes 1+1+1+1+1+3=8 cycles to execute one iteration of the software pipeline loop.
Thus, the speedup over the original loop is 18/8= 2.25.

Although the speedup of software pipelining is similar to that of unrolling the loop 5 times, this speedup is obtain more easily with software pipelining than with loop unrolling. The software pipeline does not consume additional architectural registers and the code of the loop (ignoring prologue and epilogue) has the same size as the original loop.

Problem 3.21

a.
BB (Basic Block) 1: I1,I2,I3
BB 2: I4
BB 3: I5,I6
The flowchart with basic blocks is shown below.



b. The schedule for Tomasulo algorithm with speculation is shown in Table 34.

Table 34: Tomasulo algorithm with speculation

		Dispatch	Issue	Exec start	Exec complete	Cache	CDB	Retire	Comments
I1	`LW R5,0(R3)`	1	2	(3)	3	(4)	(5)	6	
I2	`SUBI R6,R5,R2`	2	6	(7)	7	--	(8)	9	wait on R5;BNEZ at top of ROB at clock 9
I3	`BNEZ R6,NOMATCH`	3	9	(10)	(10)	--	(11)	12	wait on R6;at 12 BNEZ is detected mis-predicted; flush in 12
I4	`ADDI R3,R3,#4`	~~4~~	~~5~~	~~(6)~~	~~(6)~~	~~--~~	~~(7)~~		
I5	`BNE R4,R3,SEARCH`	~~5~~	~~8~~	~~(9)~~	~~9~~	~~--~~	~~10~~		wait on R3
I6	`LW R5,0(R3)`	~~6~~	~~9~~	~~(10)~~	~~(10)~~	~~(11)~~	~~(12)~~		wait on R3, CDB conflict w/I3
I7	`SUBI R6,R5,R2`	~~7~~							wait on R5
I8	`BNEZ R6,NOMATCH`	~~8~~				~~--~~			wait on R6
I9	`ADDI R3,R3,#4`	~~9~~	~~11~~	~~(12)~~	~~12~~	~~--~~			CDB conflict with I6
I10	`BNE R4,R3,SEARCH`	~~10~~							wait on R3
I11	`LW R5,0(R3)`	~~11~~							wait on R3
I12	`SUBI R6,R5,R2`	~~12~~							wait on R5
I13	`ADDI R1,R1,#1`	13	14	(15)	15	--	(16)	17	restart after flush
I14	`ADDI R3,R3,#4`	14	15	(16)	16		(17)	18	
I15	`BNE R4,R3,SEARCH`	15	18	(19)	19		(20)	21	wait on R3
I16	`LW R5,0(R3)`	16	18	(19)	19	20	(21)	22	wait on R3

Table 34: Tomasulo algorithm with speculation

		Dispatch	Issue	Exec start	Exec complete	Cache	CDB	Retire	Comments
I17	`SUBI R6,R5,R2`	17	22	(23)	23		(24)	25	wait on R5
I18	`BNEZ R6,NOMATCH`	18	25	(26)	26		(27)	28	wait on R6
I19	`ADDI R3,R3,#4`	19	20	(21)	21		(22)	29	
I20	`BNE R4,R3,SEARCH`	20	23	(24)	24		(25)	30	wait on R3

In Table 34, the cancelled work is crossed. Some entries in the table are in italics. This is work that was not done but reservation for resources were made.
In the first iteration, I4 is the next instruction after BNEZ in static code order but it is not dispatched because the branch is predicted taken. BNEZ instruction (I3) reaches the top of the ROB at clock 9. However, at that time its outcome is not yet known. So branch I3 must wait at the top of the ROB until it finishes its execution, which writes its result to the ROB during clock 11. In clock 12 the branch retires but at the same time the ROB and the backend are flushed. This action destroys all the work that was done between clock 4 and clock 12, even if the work done on behalf of the second iteration was correct. Starting at clock 13, the work following the BNEZ instruction (I3) is restarted, starting at instruction I13 (cycle 13) and the partial execution of the second iteration must be repeated, on top of the work needed to complete the first iteration.

As we can see, the work started on behalf of the second iteration was correct, but because the short branch (a mere skip instruction) inside of the first iteration was mispredicted the work in the second iteration must be rolled back even if it was correct. This is a weakness of ROB-based speculative architectures: when something wrong happens all following work must be scrapped, even if some of it was correct.

c.
```
SEARCH:     LW R5,0(R3)        /I1  Load item
            SUB R6,R5,R2       /I2  Compare with key
            ADDI R7,R1,#1      /I3  Assume a match
            CMOVZ R1,R7,R6     /I4  If match, then R1 is increased
            ADDI R3,R3,#4      /I5  Next item
            BNE R4,R3,SEARCH   /I6  Continue until all items
```

The BNEZ instruction is now replaced with a CMOVZ instruction and R6 is the predicate register. Register R7 holds the match count increased by 1 in case there is a match. Then, R7 is moved to R1 if R6 is zero.
The control dependency is transformed into a data dependency on R6. Thus no branch prediction is needed and the hammock in the loop body will never cause following iterations which were correctly executed speculatively to roll back.

d. If a conditional instruction is dispatched while the predicate register is pending, the conditional instruction must be annulled if the predicate turns out to be false. This will be a serious problem for instructions dependent on the conditional instruction. For example, in the program used in this exercise, I3 depends on the CMOVZ of the previous iteration. If the predicate fails, no value will be written in the destination register (in ROB) by the CMOVZ. Thus I3 may stay in the backend forever.
One way to solve this problem is to have the predicated instruction (CMOVZ) read the latest value of its destination register at dispatch (besides its two input registers) and carry that value with it. If the predicate is true, the CMOVZ instruction writes the result of the move into the destination. If the predicate is false, the CMOVZ instruction writes the previous value of the destination register

instead. This way all dependent instructions (in this case I3) will get the right value and will become ready and issue. Of course this solution makes the pipeline more complex and one can wonder whether it is justified. May be you have another, better solution???

Problem 3.22

a.

Table 35: Latency of operation under various forwarding assumptions

Source	Destination	No Forwarding	Register Forwarding	Full Forwarding
L.W	INT	3	2	1
L.W	S.W (on the memory operand and the address register)	3	2	1
INT	L.W or S.W (on the address register)	2	1	0
INT	Branch (on register)	2	1	1
L.W	Branch (on register)	3	2	2

If there is no forwarding at all, the dependent instruction can be in ID stage in the cycle right after the parent instruction has left the WB stage.
With register forwarding, a dependent instruction can be in the decode stage in the same cycle when its parent instruction is in the WB stage.
With full forwarding, a dependent instruction can enter execution after the cycle when its parent instruction has completed execution.
The one exception is when the dependent instruction is a conditional branch, because branches are executed in the ID stage and therefore cannot take advantage of the forwarding paths, they can only take advantage of register forwarding. Thus the latency TO a branch is the same for full forwarding as it is for register forwarding.

With full forwarding, the forwarding paths are indicated below:
from ME1/WB1 to EX1, EX2, EX3, and EX4
from ME2/WB2 to EX1, EX2, EX3, and EX4
from EX3/WB3 to EX1, EX2, EX3, and EX4
from EX4/WB4 to EX1, EX2, EX3, and EX4

b. Because the result of CMOVZ instruction is used in the next iteration, there is a data dependency on R1 register even though we unroll the loop three times. Therefore, R1 is not renamed.
In the scheduled code below, the last CMOVZ instruction is executed whether the branch is taken or not because the branch is delayed by one instruction.

Table 36: unrolling the loop three times

Unroll 3 times	Rename	Schedule

`LW R5,0(R3)`
`SUB R6,R5,R2`
`ADDI R7,R1,#1`
`CMOVZ R1,R7,R6`
`LW R5,0(R3)`
`SUB R6,R5,R2`
`ADDI R7,R1,#1`
`CMOVZ R1,R7,R6`
`LW R5,0(R3)`
`SUB R6,R5,R2`
`ADDI R7,R1,#1`
`CMOVZ R1,R7,R6`
`ADDI R3,R3,#12`
`BNE R4,R3,SEARCH`
	`LW R5,0(R3)`
	`SUB R6,R5,R2`
	`ADDI R7,R1,#1`
	`CMOVZ R1,R7,R6`
	`LW R8,4(R3)`
	`SUB R9,R8,R2`
	`ADDI R10,R1,#1`
	`CMOVZ R1,R10,R9`
	`LW R11,8(R3)`
	`SUB R12,R11,R2`
	`ADDI R13,R1,#1`
	`CMOVZ R1,R13,R12`
	`ADDI R3,R3,#12`
	`BNE R4,R3,SEARCH`
		`LW R5,0(R3)`
		`LW R8,4(R3)`
		`LW R11,8(R3)`
		`ADDI R7,R1,#1`
		`SUB R6,R5,R2`
		`SUB R9,R8,R2`
		`SUB R12,R11,R2`
		`CMOVZ R1,R7,R6`
		`ADDI R10,R1,#1`
		`ADDI R3,R3,#12`
		`CMOVZ R1,R10,R9`
		`ADDI R13,R1,#1`
		`BNE R4,R3,SEARCH`
		`CMOVZ R1,R13,R12`

No forwarding at all:

Table 37: VLIW program after unrolling the loop 3 times w/o forwarding

	LD/ST 1	LD/ST 2	INT	INT/BRANCH
Search:	LW R5,0(R3)	LW R8,4(R3)	NOOP	NOOP
2	LW R11,8(R3)	NOOP	NOOP	NOOP
3	NOOP	NOOP	ADDI R7,R1,#1	NOOP
4	NOOP	NOOP	NOOP	NOOP
5	NOOP	NOOP	SUB R6,R5,R2	SUB R9,R8,R2
6	NOOP	NOOP	SUB R12,R11,R2	NOOP
7	NOOP	NOOP	NOOP	NOOP
8	NOOP	NOOP	CMOVZ R1,R7,R6	NOOP
9	NOOP	NOOP	NOOP	NOOP
10	NOOP	NOOP	NOOP	NOOP
11	NOOP	NOOP	ADDI R10,R1,#1	ADDI R3,R3,#12
12	NOOP	NOOP	NOOP	NOOP
13	NOOP	NOOP	NOOP	NOOP
14	NOOP	NOOP	CMOVZ R1,R10,R9	NOOP
15	NOOP	NOOP	NOOP	NOOP
16	NOOP	NOOP	NOOP	NOOP
17	NOOP	NOOP	ADDI R13,R1,#1	NOOP
18	NOOP	NOOP	NOOP	NOOP

Table 37: VLIW program after unrolling the loop 3 times w/o forwarding

	LD/ST 1	LD/ST 2	INT	INT/BRANCH
19	NOOP	NOOP	NOOP	BNE R4,R3,SEARCH
20	NOOP	NOOP	CMOVZ R1,R13,R12	NOOP

Register Forwarding only:

Table 38: VLIW program after unrolling the loop 3 times w/ Register Forwarding

	LD/ST 1	LD/ST 2	INT	INT/BRANCH
Search:	LW R5,0(R3)	LW R8,4(R3)	NOOP	NOOP
2	LW R11,8(R3)	NOOP	NOOP	NOOP
3	NOOP	NOOP	ADDI R7,R1,#1	NOOP
4	NOOP	NOOP	SUB R6,R5,R2	SUB R9,R8,R2
5	NOOP	NOOP	SUB R12,R11,R2	NOOP
6	NOOP	NOOP	CMOVZ R1,R7,R6	NOOP
7	NOOP	NOOP	NOOP	NOOP
8	NOOP	NOOP	ADDI R10,R1,#1	NOOP
9	NOOP	NOOP	NOOP	NOOP
10	NOOP	NOOP	CMOVZ R1,R10,R9	NOOP
11	NOOP	NOOP	NOOP	ADDI R3,R3,#12
12	NOOP	NOOP	ADDI R13,R1,#1	NOOP
13	NOOP	NOOP	NOOP	BNE R4,R3,SEARCH
14	NOOP	NOOP	CMOVZ R1,R13,R12	NOOP

Full forwarding:

Table 39: VLIW program after unrolling the loop 3 times w/ Full Forwarding

	LD/ST 1	LD/ST 2	INT	INT/BRANCH
Search:	LW R5,0(R3)	LW R8,4(R3)	NOOP	NOOP
2	LW R11,8(R3)	NOOP	ADDI R7,R1,#1	NOOP
3	NOOP	NOOP	SUB R6,R5,R2	SUB R9,R8,R2
4	NOOP	NOOP	SUB R12,R11,R2	CMOVZ R1,R7,R6
5	NOOP	NOOP	ADDI R10,R1,#1	ADDI R3,R3,#12
6	NOOP	NOOP	CMOVZ R1,R10,R9	NOOP
7	NOOP	NOOP	ADDI R13,R1,#1	BNE R4,R3,SEARCH

Table 39: VLIW program after unrolling the loop 3 times w/ Full Forwarding

	LD/ST 1	LD/ST 2	INT	INT/BRANCH
8	`NOOP`	`NOOP`	`CMOVZ R1,R13,R12`	`NOOP`

c. The major limitation of loop unrolling is the number of available architectural registers. We have 32 general purpose registers. Five registers are reserved for various reasons and cannot be used for renaming: R0, R1, R2, R3, and R4. Thus 27 registers remain for renaming. Each loop unroll consumes 3 rename registers. Thus the loop can be unrolled 9 times. The problem is: is it useful?
Looking at the 3 pieces of code, we observe that the dependency chain between the CMOVZ and the increment of R1, limits the possible code compression. The minimum execution time of the loop unrolled N times is:
latency_of_Load_to_SUB + latency_of_SUB_to_CMOVZ
+(N-1) x (latency_CMOVZ_to_ADDI + latency_of_ADDI_to_CMOVZ)
Because unrolling the loop increases the code size, it is not advisable to unroll the loop more than 3 times.

Problem 3.23

In this problem, instruction scheduling is speculative.
a. Conservative policy without store-to-load forwarding

Table 40: Tomasulo algorithm with speculation and speculative scheduling

		Dispatch	Issue	Register Fetch	Exec start	Exec complete	Cache	CDB	Retire	Comments
I1	L.D F0,0(R1)	1	2	3	(4)	4	(5)	(6)	7	
I2	L.D F2,-8(R1)	2	3	4	(5)	5	(6)	(7)	8	
I3	ADD.D F0,F2,F0	3	5	6	(7)	11	--	(12)	13	wait for F2
I4	S.D-A F0,-8(R1)	4	5	6	(7)	7	--	--	--	
I5	S.D-C F0,-8(R1)	5	10	11	(12)	12	(13)	--	14	wait for F0
I6	SUBI R1,R1,#8	6	7	8	(9)	9	--	(10)	15	
I7	BNEZ R1,R2,	7	8	9	(10)	10	--	(11)	16	
I8	L.D F2,-8(R1)	8	9	10	(11)	11	(12)	(13)	17	
I9	ADD.D F0,F2,F0	9	11	12	(13)	18	--	(19)	20	wait for F2
I10	S.D-A F0,-8(R1)	10	11	12	(13)	13	--	--	--	
I11	S.D-C F0,-8(R1	11	17	18	(19)	19	(20)	--	21	wait for F0
I12	SUBI R1,R1,#8	12	13	14	(15)	15	--	(16)	22	
I13	BNEZ R1,R2,	13	14	15	(16)	16	--	(17)	23	

The addresses of loads I1 and I2 are both in the L/S queue after clock 5. Store I5 reaches the L/S queue at the end of clock 12. By that time all previous memory access addresses are known and load I2 with the same address has retired. Moreover store I5 reaches the top of the ROB at clock 13 when I3 retires. Therefore all conditions are met for store I5 to access the cache at clock 13.
The address of load I8 is known at the end of clock 11. It is different from the address of store I5 known by the end of clock 7. Thus load i8 can access the cache at clock 12, ahead of store I5.
Store I11 reaches the L/S queue at the end of clock 19. By that time all previous memory accesses have retired and store I11 is at the top of the ROB in clock 20, when I9 retires. Thus store I11 can access the cache in clock 20.

b. Store-to-load forwarding cannot improve the execution of the first two iterations of the loop in this problem. The only opportunity for forwarding would be between store I5 and load I8 (a store followed by a load). However these two memory instructions have different addresses.

c. In the optimistic policy, a load can issue to cache as soon as its address is known, even if the address of prior stores are still unknown. In the schedule of Table 40 all addresses of prior stores are always known when a load reaches the L/S queue. Therefore an optimistic policy for memory disambiguation would not improve the schedule.

Problem 3.24

```
I1              LW R5,0(R1)             /load A
I2              LW R6,0(R2)             /load B
I3              LW R7,0(R3)             /load C
I4              SLT R8,R5,R7            /A<C?
I5              SLT R9,R5,R6            /A<B?
I6              OR R9,R8,R9             /A>=C&&A>=B?
I7     (~R9)    ADD R10,R6,R7           /Yes. Compute B+C in R10.
I8     (~R9)    SW R10,0(R1)            /execute if clause
I9     (~R9)    ADDI R17,R0,#1          /set R17 to skip I17 and I18
I10    (R9)     LW R11,0(R4)            /No. Load D
111    (R9)     ADD R12,R11,R7          /C+D in R12
I12    (R9)     SLT R13,R11,R6          /B>D?
I13    (R9)     SLT R14,R5,R12          /A<C+D?
I14    (R9)     SLT R15,R12,R5          /C+D<A?
I15    (R9)     OR R16,R14,R15          /C+D!=A?
I16    (R9)     AND R17,R13,R16         /B>D&&A!=C+D??
I17    (~R17)   SUB R18,R5,R7           /NO. Execute then clause. A-C in R18
I18    (~R17)   SW R18,0(R2)
I19    exit
```

I9 is inserted to make sure that I17 and I18 are not executed when R9 is equal to 0.

Problem 3.25

a.
BB1: I1~ I5
BB2: I6 ~ I7
BB3: I8 ~ I10
BB4: I11 ~ I13
BB5: I14 ~ I15
BB6: I16 ~ I17

b. LOCAL SCHEDULING
There are 7 execution traces depending on the conditions:
Trace 1: BB1 -> BB2 -> BB3: (A>=C&&A>=B)
Trace 2: BB1 -> BB2 -> BB4 -> BB5 -> BB6: (A>=C&&A<B) and (B>D&&A=C+D)
Trace 3: BB1 -> BB2 -> BB4 -> BB5: (A>=C&&A<B) and (B>D&&A!=C+D)
Trace 4: BB1 -> BB2 -> BB4 -> BB6: (A>=C&&A<B) and (B<=D)
Trace 5: BB1 -> BB4 -> BB5 -> BB6: (A<C) and (B>D&&A=C+D)
Trace 6: BB1 -> BB4 -> BB5: (A<C) and (B>D&&A!=C+D)
Trace 7: BB1 -> BB4 -> BB6: (A<C) and (B<=D)

To compute the speedup, we first need to get the execution time without local scheduling. The original code on the VLIW machine is given below. In this VLIW program, no instruction of the original code can be scheduled before a prior instruction in process order. The VLIW program is shown in Table 41.

Table 41: VLIW program with no scheduling

BB	INSTR	LABEL	LD/ST 1	LD/ST 2	INT	INT/BRANCH
1	I1		LW R5,0(R1)	LW R7,0(R3)	NOOP	NOOP
1	I2		NOOP	NOOP	NOOP	NOOP
1	I3		LW R6,0(R2)	NOOP	SLT R8,R5,R7	NOOP
1	I4		NOOP	NOOP	NOOP	NOOP
1	I5		NOOP	NOOP	NOOP	BNEZ R8,else
1	I6		NOOP	NOOP	NOOP	NOOP
2	I7		NOOP	NOOP	SLT R9,R5,R6	NOOP
2	I8		NOOP	NOOP	NOOP	NOOP
2	I9		NOOP	NOOP	NOOP	BNEZ R9,else
2	I10		NOOP	NOOP	NOOP	NOOP
3	I11		NOOP	NOOP	ADD R10,R6,R7	NOOP
3	I12		SW R10,0(R1)	NOOP	NOOP	NOOP
3	I13		NOOP	NOOP	NOOP	J exit
3	I14		NOOP	NOOP	NOOP	NOOP
4	I15	else	LW R11,0(R4)	NOOP	NOOP	NOOP
4	I16		NOOP	NOOP	NOOP	NOOP
4	I17		NOOP	NOOP	SLT R12,R11,R6	NOOP
4	I18		NOOP	NOOP	NOOP	NOOP
4	I19		NOOP	NOOP	NOOP	BEQZ R12,else1
4	I20		NOOP	NOOP	NOOP	NOOP
5	I21		NOOP	NOOP	ADD R13,R7,R11	NOOP
5	I22		NOOP	NOOP	NOOP	NOOP
5	I23		NOOP	NOOP	NOOP	BNE R5,R13,exit
5	I24		NOOP	NOOP	NOOP	NOOP
6	I25	else1	NOOP	NOOP	SUB R14,R5,R7	NOOP
6	I26		SW R14,0(R2)	NOOP	NOOP	NOOP
	I27	exit				

The execution times of basic blocks are:

BB1: 6 clocks (I1-I6)
BB2: 4 clocks (I7-I10)
BB3: 4 clocks (I11-I14)
BB4: 6 clocks (I15-I20)
BB5: 4 clocks (I21-I24)
BB6: 2 clocks (I25-I26)
Thus the execution times of each trace is:
Trace 1: BB1 -> BB2 -> BB3: (A>=C&&A>=B): 6+4+4 = 14
Trace 2: BB1 -> BB2 -> BB4 -> BB5 -> BB6: (A>=C&&A<B) and (B>D&&A=C+D): 6+4+6+4+2 = 22
Trace 3: BB1 -> BB2 -> BB4 -> BB5: (A>=C&&A<B) and (B>D&&A!=C+D): 6+4+6+4 = 20
Trace 4: BB1 -> BB2 -> BB4 -> BB6: (A>=C&&A<B) and (B<=D): 6+4+6+2 = 18
Trace 5: BB1 -> BB4 -> BB5 -> BB6: (A<C) and (B>D&&A=C+D): 6+6+4+2 = 18
Trace 6: BB1 -> BB4 -> BB5: (A<C) and (B>D&&A!=C+D): 6+6+4 = 16
Trace 7: BB1 -> BB4 -> BB6: (A<C) and (B<=D): 6+6+2 = 14

Table 42 shows the VLIW program after local scheduling. Here instructions may be moved within basic blocks but not across branches or jumps.

Table 42: VLIW program with local scheduling

BB	INSTR	LABEL	LD/ST 1	LD/ST 2	INT	INT/BRANCH
1	I1		LW R5,0(R1)	LW R7,0(R3)	NOOP	NOOP
1	I2		NOOP	NOOP	NOOP	NOOP
1	I3		LW R6,0(R2)	NOOP	SLT R8,R5,R7	NOOP
1	I4		NOOP	NOOP	NOOP	NOOP
1	I5		NOOP	NOOP	NOOP	BNEZ R8,else
1	I6		NOOP	NOOP	NOOP	NOOP
2	I7		NOOP	NOOP	SLT R9,R5,R6	NOOP
2	I8		NOOP	NOOP	NOOP	NOOP
2	I9		NOOP	NOOP	NOOP	BNEZ R9,else
2	I10		NOOP	NOOP	NOOP	NOOP
3	I11		NOOP	NOOP	ADD R10,R6,R7	J exit
3	I12		SW R10,0(R1)	NOOP	NOOP	NOOP
4	I13	else	LW R11,0(R4)	NOOP	NOOP	NOOP
4	I14		NOOP	NOOP	NOOP	NOOP
4	I15		NOOP	NOOP	SLT R12,R11,R6	NOOP
4	I16		NOOP	NOOP	NOOP	NOOP
4	I17		NOOP	NOOP	NOOP	BEQZ R12,else1
4	I18		NOOP	NOOP	NOOP	NOOP
5	I19		NOOP	NOOP	ADD R13,R7,R11	NOOP

Table 42: VLIW program with local scheduling

BB	INSTR	LABEL	LD/ST 1	LD/ST 2	INT	INT/BRANCH
5	I20		NOOP	NOOP	NOOP	NOOP
5	I21		NOOP	NOOP	NOOP	BNE R5,R13,exit
5	I22		NOOP	NOOP	NOOP	NOOP
6	I23	else1	NOOP	NOOP	SUB R14,R5,R7	NOOP
6	I24		SW R14,0(R2)	NOOP	NOOP	NOOP
	I25	exit				

After applying local scheduling with delayed branch, the execution time of BB3 could be reduced by two clocks. Other BBs have the same execution time. So only Trace 1 benefits from local scheduling and its execution time becomes 14-2 = 12.

Because of data dependencies, we cannot take advantage of local scheduling, and there is no proper instruction that can be placed after the branch in basic blocks 1 and 3. The speedup for Trace 1 is 14/12=1.17. No speedup is achieved for the other traces.

c.
The student decided not to move stores up across branches. Why is this a good thing?
Since store instructions modify the value in memory, they cannot be executed speculatively. Hence, store instructions cannot be moved up across a branch. The code with a check instruction is given below (w/o branch/jump delay slots):

```
I1             LW R5,0(R1)               /load A
I2             LW R7,0(R3)               /load C
I3             SLT R8,R5,R7
I4             LW R6,0(R2)               /Load B
I11            LW.s R11,0(R4)            /Load D
I12            SLT R12,R11,R6
I14            ADD R13,R7,R11
I6             SLT R9,R5,R6
I8             ADD R10,R6,R7
I16            SUB R14,R5,R7
I5             BNEZ R8,then   /test A<C
I7             BNEZ R9,then              /test A<B
I9             SW R10,0(R1)              /execute if clause
I10            J exit
       then    check.s R11,repair        /inserted where I11 was
I13            BEQZ R12,then1            /test D<B
I15            BNE R5,R13,exit           /test A==C+D
I17    then1   SW R14,0(R2)              /execute then clause
I18    exit
```

The LW instruction is hoisted across a branch and a jump. Thus a check.s instruction is inserted at the original location of the LW to guard against control hazards (unwanted exceptions). Although it would seem that the LW is elevated across a SW, the flowchart reveals that the LW is never executed when the SW is executed (and vice-versa). Thus there is no data hazard. The scheduled code with the delayed branch (1 instruction) is shown in Table 43.

Table 43: VLIW program with local scheduling

BB	INST	LABEL	LD/ST 1	LD/ST 2	INT	INT/BRANCH
1	I1		LW R5,0(R1)	LW R7,0(R3)	NOOP	NOOP
1	I2		LW R6,0(R2)	LW.s R11,0(R4)	NOOP	NOOP
1	I3		NOOP	NOOP	SLT R8,R5,R7	SUB R14,R5,R7
1	I4		NOOP	NOOP	SLT R12,R11,R6	ADD R13,R7,R11
1	I5		NOOP	NOOP	SLT R9,R5,R6	BNEZ R8,then
1	I6		NOOP	NOOP	ADD R10,R6,R7	NOOP
2	I7		NOOP	NOOP	NOOP	BNEZ R9,then
2	I8		NOOP	NOOP	NOOP	NOOP
3	I9		SW R10,0(R1)	NOOP	NOOP	J exit
3	I10		NOOP	NOOP	NOOP	NOOP
4	I11	then	NOOP	NOOP	**check.s R11,repair**	BEQZ R12,then1
4	I12		NOOP	NOOP	NOOP	NOOP
5	I13		NOOP	NOOP	NOOP	BNE R5,R13,exit
5	I14		NOOP	NOOP	NOOP	NOOP
6	I15	then1	SW R14,0(R2)	NOOP	NOOP	NOOP
	I16	exit				

The execution times of basic blocks are:
BB1: 6 clocks (I1-I6)
BB2: 2 clocks (I7-I8)
BB3: 2 clocks (I9-I10)
BB4: 2 clocks (I11-I12)
BB5: 2 clocks (I13-I14)
BB6: 1 clocks (I15)
Thus the execution times of each trace are:
Trace 1: BB1 -> BB2 -> BB3: (A>=C&&A>=B): 6+2+2 = 10
Trace 2: BB1 -> BB2 -> BB4 -> BB5 -> BB6: (A>=C&&A<B) and (B>D&&A=C+D): 6+2+2+2+1 = 13
Trace 3: BB1 -> BB2 -> BB4 -> BB5: (A>=C&&A<B) and (B>D&&A!=C+D): 6+2+2+2 = 12
Trace 4: BB1 -> BB2 -> BB4 -> BB6: (A>=C&&A<B) and (B<=D): 6+2+2+1 = 11
Trace 5: BB1 -> BB4 -> BB5 -> BB6: (A<C) and (B>D&&A=C+D): 6+2+2+1 = 11
Trace 6: BB1 -> BB4 -> BB5: (A<C) and (B>D&&A!=C+D): 6+2+2 = 10
Trace 7: BB1 -> BB4 -> BB6: (A<C) and (B<=D): 6+2+1 = 9

When is the new code with speculative loads better?

The code with the speculative load performs always better than the original code without load speculation. The speedups are as follows:

Trace 1: 14/10 = 1.4
Trace 2: 22/13 = 1.69
Trace 3: 20/12 =1.67
Trace 4: 18/11 = 1.64
Trace 5: 18/11 = 1.64
Trace 6: 16/10 = 1.6
Trace 7: 14/9 = 1.56

Problem 3.26

a. The code for processing each strip of 64 components is given below:

```
      LOOP:  L.V    V1,0(R2),R6          /load X; R6 contains the stride of 1.
             L.V    V2,0(R3),R6          /load Y; 0(R3) is the base address of Y
             MUL.V  V3,V2,V1             /Multiply two vector registers
             ADD.V  V4,V4,V3             /Partial sums accumulate in V4
             ADDI   R2,R2,#64            /This assumes that memory
             ADDI   R3,R3,#64            /addresses point to vector elements
             ADDI   R4,R4,#1
             BNE    R4,R5,LOOP
```

Partial vector sums accumulate in V4. At the end we simply have to add the components of V4. To simplify we have assumed that memory addresses point to vector elements. If vector components have 8 bytes (double precision) and memory is byte-addressable, then the increments on the address registers should be 512.
Given that vector operations are chained, one iteration of the loop takes:
Tite = latency(L.V) + latency(MUL.V)+ latency(ADD.V) +(V.L)-1 = 30+10+5+64-1 = 108 cycles.
Since the loop has to iterate 16 times (16=1024/64), the total number of cycles taken by a dot-product is 108*16 =1728 cycles (ignoring the scalar phase at the end).

b.
For the multiplication of two matrices, a component of the result matrix is obtained by a dot-product of two vectors. Therefore, we need 1024*1024= 1048576 dot-products to multiply two 1024*1024 matrices. The matrix multiply takes 1728*1048576 = 1.812*10^9 cycles.

c. Unrolling the vector loop twice (after register renaming):

```
      LOOP:  L.V    V1,0(R2),R6          /load X
             L.V    V2,0(R3),R6          /load Y
             MUL.V  V3,V2,V1             /Multiply two vector registers
             ADD.V  V4,V4,V3             /Partial-sum
             ADDI   R2,R2, #64
             ADDI   R3,R3, #64
             L.V    V5, 0(R2), R6        /load X
             L.V    V6, 0(R3), R6        /load Y
             MUL.V  V7,V5,V6             /Multiply two vector registers
             ADD.V  V8,V8,V7             / partial-sum
             ADDI   R2,R2,#64
             ADDI   R3,R3,#64
             ADDI   R4,R4,#2
             BNE    R4,R5,LOOP
```

Load and multiplication on each strip of 64 components of the vector are chained and run in parallel. Partial sums are written in two different vector registers, V4 and V8. The scalar processor accumulates the components of the partial sum vectors at the end. Ignoring the scalar phases, the one iteration of the unrolled loop takes 108 cycles. Hence we can compute 128 elements of the vector in each iteration, and the number of iterations for the vector with size of 1024 is halved and it is 8(=1024/128).Therefore, the total number of cycle to execute the dot-product is 108 * 8 = 864 cycles.

For the multiplication of the two 1024*1024 matrices, we need 1024*1024 dot-product. Hence, the total number of cycle for this computation is 864*1024*1024 = $0.914*10^9$ cycles.

Problem 3.27

a. A bank number where a component Xi of a vector is stored is computed by (Stride * i) mod 32. The number of component accesses between two consecutive accesses to the same bank is called the gap denoted K. If the stride is 1, the same bank is accessed after 32 consecutive component accesses of a vector and K is 32.

The value of K for each stride S from 1 to 32 is shown in Table 44.

Table 44: Number of vector components fetched between conflicts(32 banks)

S	K	S	K	S	K	S	K
1	32	9	32	17	32	25	32
2	16	10	16	18	16	26	16
3	32	11	32	19	32	27	32
4	8	12	8	20	16	28	8
5	32	13	32	21	32	29	32
6	16	14	16	22	16	30	16
7	32	15	32	23	32	31	32
8	4*	16	2*	24	4*	32	1*

The results in Table 44 can be obtained informally by considering each stride, from 1 to 32. Since the access time of a bank is 8 clocks, conflicts occur when the gap is less than 8, i.e., 1,2 or 4. The strides causing conflicts are marked by an asterisk in Table 44 (S=8,16,24, and 32).

To dig further, let's start with the first access i=0 to bank 0. Whenever an access i accesses bank 0 it must be that ixS is a multiple of 32 so that ixS mod32=0 and, of course, ixS must be a multiple of S. Thus ixS must be divisible by both S and 32 and the first such instance is for ixS = LCM(S,32) (LCM is the least common multiple).
Thus the gap K is equal to LCM(S,32)/S, which is equal to 32/GCD(S,32). This formula extends the computation of K for all S. Because the bank access time is 8, a conflict occurs if K is less than 8.

Conclusion: A bank conflict occurs if $\frac{32}{GCD(S, 32)} < 8$.

Hence, all strides except multiples of 8, such as 8,16,24, and 32, avoid conflicts.

b. The same formula obtained in part a applies here too and K is $\frac{31}{GCD(S, 31)}$.

Because 31 is a prime number and the GCDs of all strides except for multiples of 31 are 1, the same bank is accessed every 31 accesses (K=31), except if the stride is 31, in which case the same bank is accessed on every access. Hence, all strides except multiples of 31 avoid bank conflicts.

c. In a memory system with 31 banks the number of strides causing bank conflicts is much less than in a memory system with 32 banks because 31 is a prime number. However, computing an address modulo 31 is much more complex than computing an address modulo 32.

Problem 3.28

a. In this problem we need to sum up the components of a vector by pipelining it through an adder with 8 stages. The basic idea is as follows. (This process is shown in Table 45 for a vector of 16 components).The first vector component (v0) enters the add pipeline in cycle 1. In cycle 9, the pipeline stages contain the sums of the first 8 components with 0. Then, in cycle 9, the first vector component v0 is recirculated and starts its addition with v8. Partial sums are recirculated so that, in cycle 16, the pipeline stages contain partial sums of 2 components: v0+v8, v1+v9, v2+v10, v3+v11, v4+v12, v5+v13, v6+v14, and v7+v15. This is when the feedback register becomes useful.

The special circuit is a feedback register that stores the last result of the ADD pipeline and it can be clocked at f, f/2, or f/4. Up to clock 16, the feedback register is clocked at frequency f. At clock 16, when no more operands are transmitted from the MULTIPLY pipeline, 8 partial sums of the vector are in different stages of the ADD pipeline.

From then on the feedback register provides an operand to the ADD pipeline, but some values are don't care (marked by -- in the table). The feedback register is still clocked at frequency f for 8 clocks. After these 8 clocks the pipeline contains 4 partial sums of different components (denoted S4-1,2,3,and 4). At this point we divide the clock frequency of the feedback register by 2 for the next 8 cycles. At this time we have 2 partial sums in different stages (denoted S8-1,2) and we divide the clock frequency by 2 once more. After 8 cycles we have a single sum of the 16 components in the first stage and it takes 8 more cycles to output the total sum.

The total number of clocks is 48 (16+4x8).

Table 45: Contents of ADD pipeline stages

cycle	ADD1	ADD2	ADD3	ADD4	ADD5	ADD6	ADD7	ADD8	Feedback Reg. content	Feedback Reg. Clock
8	V7+0	V6+0	V5+0	V4+0	V3+0	V2+0	V1+0	V0+0	--	clock
9	V8+V0	V7	V6	V5	V4	V3	V2	V1	V0	clock
...										clock
16	V15+V7	V14+V6	V13+V5	V12+V4	V11+V3	V10+V2	V9+V1	V8+V0	V7	clock
17	--	V15+V7	V14+V6	V13+V5	V12+V4	V11+V3	V10+V2	V9+V1	V8+V0	clock
18	S4-1	--	V15+V7	V14+V6	V13+V5	V12+V4	V11+V3	V10+V2	V9+V1	clock
19	--	S4-1	--	V15+V7	V14+V6	V13+V5	V12+V4	V11+V3	V10+V2	clock
20	S4-2	--	S4-1	--	V15+V7	V14+V6	V13+V5	V12+V4	V11+V3	clock
21	--	S4-2	--	S4-1	--	V15+V7	V14+V6	V13+V5	V12+V4	clock
22	S4-3	--	S4-2	--	S4-1	--	V15+V7	V14+V6	V13+V5	clock
23	--	S4-3	--	S4-2	--	S4-1	--	V15+V7	V14+V6	clock
24	S4-4	--	S4-3	--	S4-2	--	S4-1	--	V15+V7	clock

Table 45: Contents of ADD pipeline stages

cycle	ADD1	ADD2	ADD3	ADD4	ADD5	ADD6	ADD7	ADD8	Feedback Reg. content	Feedback Reg. Clock
25	--	S4-4	--	S4-3	--	S4-2	--	S4-1	V15+V7	
26	--	--	S4-4	--	S4-3	--	S4-2	--	S4-1	clock
27	--	--	--	S4-4	--	S4-3	--	S4-2	S4-1	
28	S8-1	--	--	--	S4-4	--	S4-3	--	S4-2	clock
29	--	S8-1	--	--	--	S4-4	--	S4-3	S4-2	--
30	--	--	S8-1	--	--	--	S4-4	--	S4-3	clock
31	--	--	--	S8-1	--	--	--	S4-4	S4-3	
32	S8-2	--	--	--	S8-1	--	--	--	S4-4	clock
33	--	S8-2	--	--	--	S8-1	--	--	S4-4	
34	--	--	S8-2	--	--	--	S8-1	--	S4-4	
35	--	--	--	S8-2	--	--	--	S8-1	S4-4	
36	--	--	--	--	S8-2	--	--	--	S8-1	clock
37	--	--	--	--	--	S8-2	--	--	S8-1	
38	--	--	--	--	--	--	S8-2	--	S8-1	
39	--	--	--	--	--	--	--	S8-2	S8-1	
40	S16	--	--	--	--	--	--	--	--	clock

b.Let 64K be the number of components to add. The two input vectors are streamed from memory into the MULTIPLY pipeline. After 10 clocks the first result enters the ADD pipeline.
64K cycles later (a multiple of 8), partial sums of the 64K components are in each stage of the add pipeline.
As no more operands are transmitted from the MULTIPLY pipeline, 8 partial sums of the vector are in different stages of the ADD pipeline. Then the accumulation continues using the feedback register and the same algorithm as in Table 45. It takes 32 cycles to finish off the accumulation, and this time is independent of the vector length and only depends on the number of stages in the ADD pipeline.

c.
Memory_latency + Multiply_latency + (V.L) + 32= 100+10+65536+32 = 65678 cycles
d. To simplify we assume that N is a multiple of 8 (if it is not the vector is padded with 0's), the number of stages. The total number of cycles to execute the dot-product is 100+10+N+32= N+142.
To compare performance, we compute the execution time per element of a vector. The execution time per element of the vector of size 64K is 65678/65536= 1.0022 cycles/ element.
We need that $(N_{1/2}+142)/N_{1/2}=2x65678/65536$ or $N_{1/2}+142 = 2x65678/65536\ N_{1/2}$ or $N_{1/2} = 141.4$
Thus because of the long latencies (especially of memory) the machine does not perform well for short vectors.

e. On static pipeline it takes 23 cycles per element of a vector. Thus, it takes 23N cycles for vectors of size N.

The vector machine of Figure 3 in the problem statement takes 142+N cycles. The condition is:
$23N>142+N$ or $N>142/22$ or $N>6.45$

Therefore, the vector machine is much better than the scalar machine, even for very short vectors.

CHAPTER 4

Problem 4.1

This problem injects cache misses, structural hazards and nonblocking (lockup-free) caches in the behavior of OoO processors. A simple program that accumulates values from memory is used in this problem:

```
LOOP:   LW R4,0(R3)
        ADDI R3,R3,#4
        ADD R1,R1,R4
        BNE R3,R5,LOOP
```

BNE is always predicted taken.

a.

Table 46: Tomasulo algorithm with speculation (NO PREFETCH)

		Dispatch	Issue	Exec start	Exec complete	Cache	CDB	Retire	COMMENTS
I1	LW R4,0(R3)	1(1)	2	(3)	3	13*(4)	14(5)	15	Primary miss
I2	ADDI R3,R3,#4	2(2)	3	(4)	4	--	(5)	16	
I3	ADD R1,R1,R4	3(3)	15	(16)	16	--	(17)	18	wait on R4
I4	BNE R3,R5,LOOP	4(4)	6	(7)	7	--	(8)	19	wait on R3
I5	LW R4,0(R3)	5(5)	6	(7)	7	14*(8)	15(9)	20	Secondary miss
I6	ADDI R3,R3,#4	6(6)	8	(9)	9	--	(11)	22	CDB conflict with I5
I7	ADD R1,R1,R4	7(7)	18	(19)	19	--	(20)	23	wait on R1
I8	BNE R3,R5,LOOP	8(8)	12	(13)	13	--	(14)	24	wait on R3
I9	LW R4,0(R3)	15(8)	16	(17)	17	27*(18)	28(19)	29	ROB full;primary miss
I10	ADDI R3,R3,#4	16(8)	19	(20)	20	--	(21)	30	CDB conflicts with reservation by I9,I7
I11	ADD R1,R1,R4	18(8)	29	(30)	30	--	(31)	32	ROB full; wait on R4
I12	BNE R3,R5,LOOP	19(8)	22	(23)	23	--	(24)	33	wait on R3
I13	LW R4,0(R3)	20(8)	21	(22)	22	28*(23)	30(24)	34	Secondary miss; CDB conflict with I16
I14	ADDI R3,R3,#4	22(8)	23	(24)	24	--	(25)	35	ROB full
I15	ADD R1,R1,R4	23(8)	32	(33)	33	--	(34)	36	wait on R1
I16	BNE R3,R5,LOOP	24(8)	26	(27)	27	--	(29)	37	wait on R3

In the cache column the asterisk indicates that the cache port is busy during that cycle. Other instructions cannot access the cache in that cycle. In the Cache and CDB columns we show the reservations made by loads at their time of issue. Unfortunately these reserved CDB cycles are wasted. The load I13 misses and additionally must wait until the CDB is free. The CDB was reserved for clock 24, but, because of the miss, this reservation is missed and wasted and the cache must wait two additional cycles to put the value on the bus because of bus conflicts with reservations from other instructions.

b. To avoid the misses, non-binding prefetch instructions are inserted in the code. The new instruction is PW d(R), where d is the displacement and R is a base address. This instruction loads a block in cache but does not return any value on the CDB and does not need to retire (All exceptions caused by prefetches are dropped.)

The new code is:

```
LOOP:  LW R4,0(R3)
       PW 8(R3)
       ADDI R3,R3,#4
       ADD R1,R1,R4
       BNE R3,R5,LOOP
```

When the prefetch hits in cache or if it is a secondary miss, it is dropped. Fill Table 47 for the first 4 iterations. Does the prefetch improves performance?

Table 47: Tomasulo algorithm with speculation (and PREFETCH)

		Dispatch	Issue	Exec start	Exec complete	Cache	CDB	Retire	COMMENTS
I1	LW R4,0(R3)	1(1)	2	(3)	3	13*(4)	14(5)(	15	Primary miss
I2	PW 8(R3)	2(1)	3	(4)	4	14*(5)	--	--	Primary prefetch miss
I3	ADDI R3,R3,#4	3(2)	4	(5)	5	--	(6)	16	
I4	ADD R1,R1,R4	4(3)	15	(16)	16	--	(17)	18	wait on R4
I5	BNE R3,R5,LOOP	5(4)	7	(8)	8	--	(9)	19	wait on R3
I6	LW R4,0(R3)	6(5)	7	(8)	8	15*(9)	16(10)	20	Secondary miss; cache port conflict with I2
I7	PW 8(R3)	7(5)	8	(9)	9	(10)	--	--	Dropped prefetch
I8	ADDI R3,R3,#4	8(6)	9	(10)	10	--	(11)	21	
I9	ADD R1,R1,R4	9(7)	18	(19)	19	--	(20)	22	wait on R1
I10	BNE R3,R5,LOOP	10(8)	12	(13)	13	--	(14)	23	wait on R3
I11	LW R4,0(R3)	15(8)	16	(17)	17	(18)	(19)	24	Cache hit
I12	PW 8(R3)	16(7)	17	(18)	18	28*(19)	--	--	Primary prefetch miss
I13	ADDI R3,R3,#4	17(8)	19	(20)	20	--	(21)	25	conflicts w/I9
I14	ADD R1,R1,R4	18(8)	21	(22)	22	--	(23)	26	wait on R1
I15	BNE R3,R5,LOOP	19(8)	22	(23)	23	--	(24)	27	wait on R3
I16	LW R4,0(R3)	20(8)	22	(23)	23	(24)	(25)	28	wait on R3;cache hit
I17	PW 8(R3)	21(7)	23	(24)	24	(25)	--	--	Conflicts with I16; Dropped prefetch
I18	ADDI R3,R3,#4	22(7)	23	(24)	24	--	(25)	29	
I19	ADD R1,R1,R4	23(7)	26	(27)	27	--	(28)	30	wait on R4
I20	BNE R3,R5,LOOP	24(7)	27	(28)	28		(29)	31	wait on R3; conflicts with I19

Due to prefetching, all loads after the first one hit. This improves the schedule slightly, even if every load is shadowed by a prefetch, which doubles the instruction overhead of loads. One dominant factor in this schedule is the constraint on dispatch due to the small ROB. Note that if we unrolled the loop twice the number of prefetches would be halved and prefetches would never be dropped.

Problem 4.2

In this problem we design and size structures supporting virtual memory. The following parameters apply to a system employing a 42-bit virtual address and 256MBytes of physical memory. Word size is 64 bits (8 bytes). Addresses point to bytes and are aligned on byte boundaries. We use the following notation for an i-bit address: Ai-1...A2A1A0 where Ai-1 is the most significant bit of the address and A0 is the least significant bit of the address. The virtual address is denoted by V41-V0 and the physical address is denoted by P27-P0.

a. PAGE TABLES. Consider the following parameters.
Page size: 4KBytes
Page table: Three-level page table. The virtual page number is split in 3 fields: 8 bits, 11 bits and 11 bits.
Entries in tables are 32 bits (4 bytes).
1. Which bits of the virtual address are used to index the first level table (top level in the hierarchy):
Answer: bits V_41_ to V_34_
2. Which bits of the virtual address are used to index the page tables at the bottom of the hierarchy:
Answer: bits V_22_ to V_12_
3. What is the size of each page table at every level (in bytes)?
level1: 2^8*4byte = 1KB
level2: 2^{11}*4byte = 8KB
level3: 2^{11}*4byte = 8KB
4. What is the total amount of virtual memory covered by entries of page table at each level:
level1: 16GB
level2: 8MB
level3: 4KB

b. TLB
TLB size: 256 entries
TLB organization: 2-way set-associative
1. Which bits of the virtual address are used to index the TLB:
Answer: bits V_18_ to V_12_
2. Which bits of the virtual address are used as tags in the TLB?
Answer: bits V_41_ to V_19_
3. TLB tag size: 23 bits

c. Second-Level Cache
Cache size: 5 MB
Line size: 64 bytes
Set size: 10 lines per set
The second-level cache is accessed with physical addresses. The width of each data ram column is 64 bits (1 word).
1. Which bits of the physical address are used to index the tag RAM of the directory?
Answer: bits P_18_ to P_6_
2. Which bits of the physical address are used to index the cache data RAM?
Answer: bits P_18_ to P_3_
3. Tag size: 9 bits

c. First-Level Cache
The first level cache is a virtually indexed, physically tagged 3-way cache.
First-level Cache size: 96KB
Block size: 16bytes
Set size: 3 blocks per set
The width of each data ram column is 64 bits.
1. Which bits of the virtual address are used to index the tag RAM of the directory?
Answer: bits V_14_ to V_4_
2. Which bits of the virtual address are used to index the data RAM of the directory?
Answer: bits V_14_ to V_3_
3. Which bits of the physical address are matched against the tags in the Tag RAMs?
Answer: bits P_27_ to P_12_

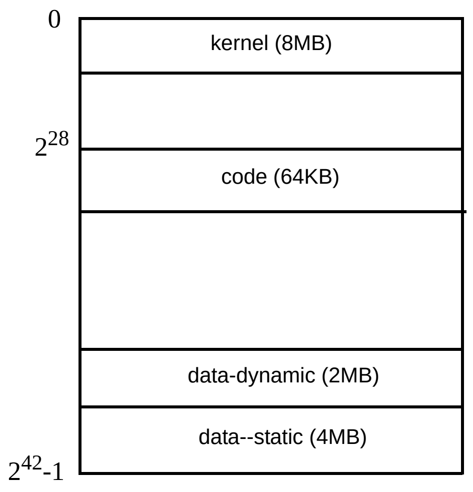
4. In this first-level cache, there are consistency problems due to synonyms. One solution is to search all the sets that could contain the block on every miss. How many sets should be searched?
Number of sets: ___8_____
5. Another solution to solve the synonym problem is page coloring. What are the bits defining the color of the page?
Color bits: bits V_14_ to V_12_

Problem 4.3

This problem is about the structure of page tables to support large virtual address spaces. Assume a 42-bit virtual address space per process in a 64-bit machine and 512MByte of main memory. The page size is 4KBytes. Page table entries are 4 byte in every table. Various hierarchical page table organizations are envisioned: 1, 2, and 3 levels. The virtual space to map is populated as shown in Figure 2. Kernel space addresses are not translated because physical addresses are identical to virtual addresses. However virtual addresses in all other segments must be translated.
Please answer the following questions:

1. What would be the size of a single-level page table?
The total number of pages is 2^{30} and each entry has 4 bytes. So the page table size is 4GB. This would take more than the entire memory in most PCs today.

2. Assume now a 2-level page table. We split the 30 bits of virtual page number into two fields of 15-bits each? How many page tables would we have? What would be their total size?
The first level (root) page table must be allocated. It is accessed with 15 bits and therefore has 32K entries of 4 bytes each for a total of 128KB. Each entry of the first-level page table covers 2^{15+12} = 2^{27}=128MB of virtual memory. We don't need a second-level table for the kernel since it is not mapped. But we need a second-level page table for the code segment and the two data segments. The code segment lies in the virtual memory area covered by the second entry of the first-level page table. The two data segments lie in the virtual memory area covered by the last entry of the first-level page table. Therefore we need a total of two second-level page tables which occupy 128KB each. Thus the number of page tables is 3 and the total physical memory occupied by the page tables is 3x128KB = 384KB, which is a huge reduction as compared to the single-level page table.



3. Repeat 2. for a 3-level page table splitting the 30 bits of virtual page number into 3 fields of 10 bits each.
The root page table has 1K entries of 4 bytes each, for a total of 4KB. The kernel is unmapped. each

entry of the first-level page table covers 2^{20+12}=2^{32}=4GB of virtual space. The code segment falls within this space. So we need one second-level page table to cover the code (size 4KB). The two data segments lie in the virtual memory area covered by the last entry of the root page table. So we need another second level page table for the data (size 4KB). Each entry of the second-level page tables covers 2^{10+12} = 2^{22} = 4MB of virtual memory. This is enough to cover the code segment (1 third-level table or 4KB). However the data segments requires two third-level page tables as they occupy 6MB of virtual memory. The size of these two tables is 8KB. Thus the grand total is 1+2+3 = 6 tables or a total physical memory occupation of 24KB.

We see that, in terms of space (physical memory resources), having more levels in the page table hierarchy pays huge dividends. However the more levels in the hierarchy, the more time it takes to walk through the tables to translate a virtual address. This is a classical space-time trade-off. However the trade-off is very favorable in this context because a TLB can effectively bypass the whole table hierarchy.

Problem 4.4

Pseudo-LRU is an approximate LRU algorithm which requires much less overhead to manage. Thus pseudo-LRU may be less effective and cause more misses than LRU but the update of the bits required to identify the victim block is much less complex.

LRU is easily manageable for a 2-way cache. One single bit per set is sufficient to point to the LRU line. However, for larger set sizes, the complexity of LRU grows exponentially. For a 4-way cache, exact LRU needs four two bits fields, which must be updated on each access. Each two bit field is associated with a line and indicates the relative recency of access to each line in the set. This is a total of 8 bits. For a 16-way cache, exact LRU will require 16 sets of 4 bits each or 64bits total. In general, for an N-way cache, exact LRU needs N$\log_2$N bits.

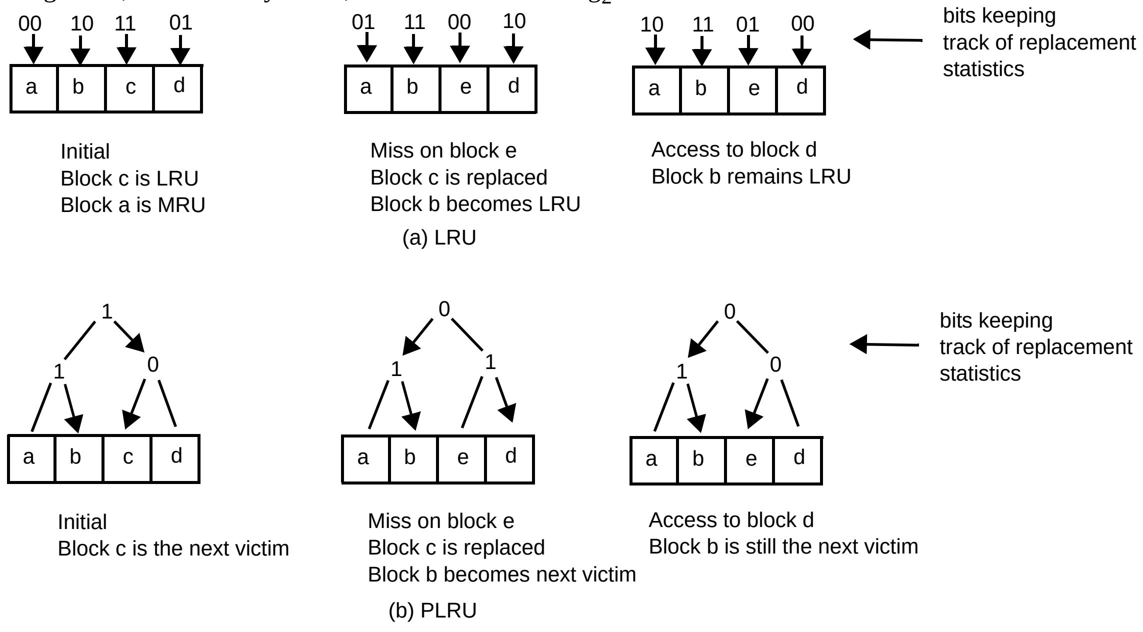


Figure 3. LRU vs. PLRU

For this reason pseudo-LRU is often preferred. Pseudo-LRU only works with power-of-two set sizes. In pseudo-LRU, the lines in each set are recursively partitioned in two subsets, and one bit points to the LRU subsets at each level of the recursive partition.

Let’s say that we have N lines in a set, where N=2^n. These N lines are split into two subsets of size N/2 each. One bit is enough to track the LRU subset. Then each subset is again split into two sub-subsets of size N/4 each, which requires two bits to track the LRU sub-subsets in the subsets, for a total of 3 bits. Next we split the sub-subsets again into sub-sub-subsets of size N/8 each, which requires four bits, for a total of 7 bits. We do this recursively until we cannot split anymore because we have reached individual lines. The total number of bits is $\log_2$N-1, much less than for LRU. For example, for a 16-way cache, the number of bits is 15, as compared to 64 for LRU. Therefore the FSM updating the replacement policy bits is much simpler.

When the set size is 2, LRU and pseudo-LRU are identical. Only 1 bit is needed. When it is more than 2 lines, the bit at each level points to the LRU subset at the next level. So we keep track of the LRU subset at each level.

The difference between LRU bits and pseudo-LRU (PLRU) bits management is illustrated in Figure 3 for a set size of 4 cache lines (to simplify, the cache size is four lines).

The four lines are divided into two subsets: S0={line0,line1} and S1={line2,line3}. Each subset is subdivided into two lines. In the initial state, S0 contains blocks a and b and S1 contains blocks c and d. At the root of the tree bit value 0 points to S0 and bit value 1 points to S1. This correspondence is indicated by the arrows. Within S0, one bit points to line0 (value 0) or line1 (value 1). A bit always points away from the MRU subset and towards the LRU subset. So in the initial state, the MRU subset is S0 and the LRU subset is S1. Within S1, the LRU line is line2 (containing block c) and the MRU line is line 3. This is consistent with the initial state of LRU. After a miss on block e the victim is pointed to by the arrows, i.e., block c contained in line2. At this point the LRU subset becomes S0 and line1 is the new candidate for replacement. Then block d is accessed. S1 remains the MRU subset and line1 contains the next victim, i.e., block b. This is still consistent with LRU.

PLRU approximates LRU, and in this example it points to the same victim as LRU. However, PLRU will eventually diverge from LRU.

Problem 4.5

a.
DIRECT-MAPPED:

Table 48: Direct-Mapped

Cycle #	1						2						3						4					
Block Address	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5
Line0	0	0	0	0	4	4	0	0	0	0	4	4	0	0	0	0	4	4	0	0	0	0	4	4
Line1	-	1	1	1	1	5	5	1	1	1	1	5	5	1	1	1	1	5	5	1	1	1	1	5
Line2	-	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Line3	-	-	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
Misses	*	*	*	*	*	*	*	*			*	*	*	*			*	*	*	*			*	*

In the first cycle, all blocks (0,1,2,3,4,5) miss in the cache (cold misses). From then on the blocks mapping to lines 0 and 1 keep missing (4 misses) while blocks 2 and 3 remain in lines 2 and 3 (2 hits) in every cycle. So, after 6 misses in the first cycle, all subsequent cycles experience 4 misses.

Total number of misses = 6 + (4 * 9) = **42 misses**

FA with LRU REPLACEMENT:

Table 49: FA with LRU

Cycle #	**1**						**2**						**3**						**4**					
Block Address	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5
Line0	0	0	0	0	4	4	4	4	2	2	2	2	0	0	0	0	4	4	4	4	2	2	2	2
Line1	-	1	1	1	1	5	5	5	5	3	3	3	3	1	1	1	1	5	5	5	5	3	3	3
Line2	-	-	2	2	2	2	0	0	0	0	4	4	4	4	2	2	2	2	0	0	0	0	4	4
Line3	-	-	-	3	3	3	3	1	1	1	1	5	5	5	5	3	3	3	3	1	1	1	1	5
Misses	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*

In the first iteration, all blocks (0,1,2,3,4,5) miss in the cache (cold misses). From then on the cache misses at every reference because the cache cannot hold block copies across cycles. LRU copies are replaced and then accessed again. Thus we have 6 misses per cycle.

Total number of misses = 6 * 10 = **60 misses**

FA with FIFO REPLACEMENT:

Table 50: FA with FIFO

Cycle #	**1**						**2**						**3**						**4**					
Block Address	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5
Line0	0	0	0	0	4	4	4	4	2	2	2	2	0	0	0	0	4	4	4	4	2	2	2	2
Line1	-	1	1	1	1	5	5	5	5	3	3	3	3	1	1	1	1	5	5	5	5	3	3	3
Line2	-	-	2	2	2	2	0	0	0	0	4	4	4	4	2	2	2	2	0	0	0	0	4	4
Line3	-	-	-	3	3	3	3	1	1	1	1	5	5	5	5	3	3	3	3	1	1	1	1	5
Misses	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*

In the first iteration, all blocks (0,1,2,3,4,5) miss in the cache (cold misses). From then on, the cache misses at every reference. Because the reference string is cyclic, the FIFO cache behaves like the LRU cache. Again we have 6 misses per cycle.

Total number of misses = 6 * 10 = **60 misses**
FA with LIFO REPLACEMENT:

Table 51: FA with LIFO

Cycle #	**1**						**2**						**3**						**4**					
Block Address	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5

Table 51: FA with LIFO

Cycle #	1						2						3						4					
Line0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Line1	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Line2	-	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Line3	-	-	-	3	4	5	5	5	5	3	4	5	5	5	5	3	4	5	5	5	5	3	4	5
Misses	*	*	*	*	*	*				*	*	*				*	*	*				*	*	*

In the first iteration, all blocks (0,1,2,3,4,5) miss in the cache (cold misses). From then on, blocks 0,1,and 2 stay in cache and the block in line 3 is constantly victimized to hold lines 3, 4 and 5. So, we have 3 misses per cycle.

Total number of misses = 6 + (3 * 9) = **33 misses**

2-way SA with LRU REPLACEMENT:

Table 52: 2-way SA with LRU per set

Cycle #	1						2						3						4					
Block Address	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5
Line0	0	0	0	0	4	4	4	4	2	2	2	2	0	0	0	0	4	4	4	4	2	2	2	2
Line1	-	-	2	2	2	2	0	0	0	0	4	4	4	4	2	2	2	2	0	0	0	0	4	4
Line2	-	1	1	1	1	5	5	5	5	3	3	3	3	1	1	1	1	5	5	5	5	3	3	3
Line3	-	-	-	3	3	3	3	1	1	1	1	5	5	5	5	3	3	3	3	1	1	1	1	5
Misses	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*

Set 0 is made of Line 0 and 1 and set 1 is made of Line 2 and 3. We can see the reference string of 5 as two interleaved strings one with odd numbered block and one with even numbered block. Each of the interleaved string access an LRU cache of size 2. As with the case for the FA LRU cache, since each string length (3) is greater than the cache size (2), every access misses. So we have 6 misses per cycle.

Total number of misses = 6 * 10 = **60 misses.**

The interesting observation from all this is that, in cyclic reference patterns (such as to instructions in loops), the FA-LIFO policy is the best, followed by direct-mapped mapping.

b.
COLD MISSES:
The number of cold misses is independent of the cache organization and replacement policy. The number of cold misses is equal to the number of blocks in the trace:

Nb_cold_misses = **6 misses**

CAPACITY MISSES:
The number of capacity misses is also independent of the cache organization and replacement policy. It is given by the number of misses in an FA LRU cache minus the number of cold misses:

Nb_capacity_misses = 60 - 6 = **54 misses**

CONFLICT MISSES:
The number of conflict misses depends on the cache organization and replacement policy. It is obtained by subtracting both cold and capacity misses from the total number of misses of the cache:

i) Direct-mapped: Nb_conflict_misses = 42 - (6 + 54) = **-18 misses**

ii) FA with LRU: Nb_conflict_misses = 60 - (6 + 54) = **0**

iii) FA with FIFO: Nb_conflict_misses = 60 - (6 + 54) = **0**

iv) FA with LIFO: Nb_conflict_misses = 33 - (6 + 54) = **-27 misses**

v) 2-way SA with LRU for each set: Nb_conflict_misses = 60 - (6 + 54) = **0**

Since we are using a FA cache with LRU to calculate the capacity misses, we end up having negative number of conflict misses for some cache organization and this of course is not acceptable. The problem is FA cache with LRU is not the best cache and hence can not be used, instead we should use a FA cache with optimal replacement algorithm as we will do in part c and part d.

c.
FA with OPT REPLACEMENT:
The cache states for the entire sequence of references is shown in Table 53. After the cache is filled, after access #4, two accesses misses and refill the LIFO position of the stack, then three accesses hit on the other three line. This pattern with cycle 5 repeats until the end of the trace. Because the number of remaining accesses after 4 is not a multiple of 5, the last cycle is truncated.
Total number of misses = 4 + (59-4)/5*2 + 1 = 4+22+1 = **27 misses**

d.
Conflict misses are counted for each cache by subtracting the number of misses in FA-OPT from the number of misses in each cache:

i) Direct-mapped: Nb_conflict_misses = 42 - 27 = **15 misses**

ii) FA with LRU: Nb_conflict_misses = 60 - 27 = **33 misses**

iii) FA with FIFO: Nb_conflict_misses = 60 - 27 = **33 misses**

iv) FA with LIFO: Nb_conflict_misses = 33 - 27 = **6 misses**

v) 2-way SA with LRU for each set: Nr_conflict_misses = 60 - 27 = **33 misses**
By using a FA cache with OPT replacement policy as the baseline to count capacity misses, the number of conflict misses is positive for all cache organizations. This is clearly the appropriate way to classify misses. However, the number of misses in FA-LRU is easier to count and FA-LRU may

be acceptable as a baseline for all practical purposes.

Table 53: FA with OPT

Cycle #	**1**						**2**						**3**						**4**					
Block Address	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5
Line0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	2	2	2	2
Line1	-	1	1	1	1	1	1	1	1	1	1	1	1	1	2	3	3	3	3	3	3	3	3	3
Line2	-	-	2	2	2	2	2	2	2	3	4	4	4	4	4	4	4	4	4	4	4	4	4	4
Line3	-	-	-	3	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
Misses	*	*	*	*	*	*				*	*				*	*				*	*			
Cycle #	**5**						**6**						**7**						**8**					
Block Address	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5
Line0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	4	4	4	4	4	4	4	4
Line1	3	3	3	3	3	3	3	3	3	3	4	5	5	5	5	5	5	5	5	5	5	5	5	5
Line2	4	4	4	4	4	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Line3	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	3	3	3
Misses	*	*				*	*				*	*				*	*				*	*		
Cycle #	**9**						**10**																	
Block Address	0	1	2	3	4	5	0	1	2	3	4	5												
Line0	4	4	4	4	4	4	4	4	4	4	4	5												
Line1	5	5	5	5	5	5	0	1	1	1	1	1												
Line2	0	1	2	2	2	2	2	2	2	2	2	2												
Line3	3	3	3	3	3	3	3	3	3	3	3	3												
Misses		*	*				*	*				*												

Problem 4.6

a.

i) LRU:

Table 54: LRU

trace	a	a	b	c	a	a	d	e	f	f	e	f	e	f	e	f	a	b	g	c	a	e	f
Miss/Hit	M	H	M	M	H	H	M	M	M	H	H	H	H	H	H	H	H	M	M	M	H	M	M
LRU priority list	a	a	b a	c b a	a c b	a c b	d a c b	e d a c	f e d a	f e d a	e f d a	f e d a	e f d a	f e d a	e f d a	f e d a	a f e d	b a f e	g b a f	c g b a	a c g b	e a c g	f e a c

LRU miss rate = (number of misses) / (number cache accesses) = 11 / 23 = 0.478

ii) FIFO:

Table 55: FIFO

trace	a	a	b	c	a	a	d	e	f	f	e	f	e	f	e	f	a	b	g	c	a	e	f
Miss/Hit	M	H	M	M	H	H	M	M	M	H	H	H	H	H	H	H	M	M	M	M	H	M	M
FIFO priority list	a	a	b a	c b a	c b a	c b a	d c b a	e d c b	f e d c	f e d c	f e d c	f e d c	f e d c	f e d c	f e d c	f e d c	a f e d	b a f e	g b a f	c g b a	c g b a	e c g b	f e c g

FIFO miss rate = 12 / 23 = 0.522

iii) Pseudo-LRU (PLRU):

Table 56: PLRU

trace	a	a	b	c	a	a	d	e	f	f	e	f	e	f	e	f	a	b	g	c	a	e	f
Miss/Hit	M	H	M	M	H	H	M	M	M	H	H	H	H	H	H	H	H	M	M	M	H	M	M
PLRU priority list	a	a	b a	c b a	a b c	a b c	d c a b	e a d c	f d e a	f d e a	e a f d	f d e a	e a f d	f d e a	e a f d	f d e a	a e f d	b f a e	g a b f	c b g a	a g c b	e c a g	f a e c

PLRU miss rate = 11 / 21 = 0.478

iv) LFU:

Table 57: LFU

trace	a	a	b	c	a	a	d	e	f	f	e	f	e	f	e	f	a	b	g	c	a	e	f
Miss/Hit	M	H	M	M	H	H	M	M	M	H	H	H	H	H	H	H	H	M	M	M	H	H	H
LFU priority list	a	a	a b	a b c	a b c	a b c	a b c d	a c d e	a d e f	a f d e	a f e d	a f e d	a f e d	a f e d	a f e d	a f e d	a f e d	a f e b	a f e g	a f e c	a f e c	a f e c	a f e c

LFU miss rate =9 / 23 = 0.391

v) Optimum:

Table 58: OPT

trace	a	a	b	c	a	a	d	e	f	f	e	f	e	f	e	f	a	b	g	c	a	e	f
Miss/Hit	M	H	M	M	H	H	M	M	M	H	H	H	H	H	H	H	H	H	M	M	H	H	H

Table 58: OPT

trace	a	a	b	c	a	a	d	e	f	f	e	f	e	f	e	f	a	b	g	c	a	e	f
Opti- mum priority list	a	a	a b	a b c	a b c	a b c	a b c d	e a b c	f e a b	e f a b	f e a b	e f a b	f e a b	e f a b	f a b e	a b e f	b a e f	a e f b	a e f g	a e f c	e f c a	f c a e	c a e f

Optimum miss rate = 8/23 = 0.348

b.

COLD MISSES:
The number of cold misses is independent of the cache organization and replacement policy. The number of cold misses is equal to the number of different blocks in the trace:
Nb_cold_misses = **7 misses**

i) Using FA-LRU to calculate capacity misses:
The number of capacity misses is also independent of the cache organization and replacement policy. It is given by the number of misses in an FA LRU cache minus the number of cold misses:
Nb_capacity_misses = 11 - 7 = **4 misses**

CONFLICT MISSES:
1. FA with LRU: Nb_conflict_misses = 11 - (7 + 4) = **0**
2. FA with FIFO: Nb_conflict_misses = 12 - (7 + 4) = **1 miss**
3. FA with PLRU: Nb_conflict_misses = 11 - (7 + 4) = **0**
4. FA with LFU: Nb_conflict_misses = 9 - (7 + 4) = **-2 misses**
5. FA with OPT: Nb_conflict_misses = 8 - (7 + 4) = **-3 misses**

ii) Using FA-OPT to calculate capacity misses:
The number of capacity misses is also independent of the cache organization and replacement policy. It is given by the number of misses in an FA-OPT cache minus the number of cold misses:
Nb_capacity_misses = 8 - 7 = **1 miss**

CONFLICT MISSES:
1. FA with LRU: Nb_conflict_misses = 11 - (7 + 1) = **3 misses**
2. FA with FIFO: Nb_conflict_misses = 12 - (7 + 1) = **4 misses**
3. FA with PLRU: Nb_conflict_misses = 11 - (7 + 1) = **3 misses**
4. FA with LFU: Nb_conflict_misses = 9 - (7 + 1) = **1 miss**
5. FA with OPT: Nb_conflict_misses = 8 - (7 + 1) = **0**

Problem 4.7

1. The pseudocode for Matrix Multiplication X = Y*Z is

```
// X = Y*Z
initialize X = 0;
for (i = 0; i<N; i++)
    for (j = 0; j<N; j++)
        for (k = 0; k<N; k++)
            X[i][j] += Y[i][k] * Z[k][j];
```

There are N^3 fp multiplications and N^3 fp additions. Thus a total of $2N^3 = O(N^3)$ fp operations are involved in this matrix multiplication problem. With N=256, the total number of FP ops is 2^{25} =

32M FP ops.

Each cache line of size 8 bytes can contain a single FP operand. Because the cache size can easily contain one matrix, accesses to rows X and Y have no capacity misses. Take each element of X. Each element X[i][j] is computed in a register then stored back to memory once it is computed. Thus X[i][j] is accessed only once and only has a single cold miss. Y is accessed row by row. The reuse distance (the number of distinct blocks accessed between two consecutive accesses to the same block) of Y is 1 column of Z plus one row of Y, i.e. 16N $=2^{12}$=4K. This fits well within the cache size of 128K. Thus the only misses on X and Y are cold misses.

However Z (of size $8N^2=2^{19}$=512K) is too big to fit in the cache and must be reloaded in cache for the computation of every row of X. The total number of misses on elements of Z is N^3.
Thus the total number of misses is $N^3+2N^2= 2^{24}+2^{17}$ = 16,908,288.

2. In order to minimize cache misses, k should be at least 4, so a tile can be kept in cache while it is computed. The code for k=4 is:

```
// X = Y*Z
initialize X = 0;
for (i = 0; i<N; i+=N/4)
    for (j = 0; j<N; j+=N/4)
        for (k = 0; k<N; k+=N/4)
            for (ii = i; ii<min(i+N/4-1, N); ii++)
                for (jj = j; jj<min(j+N/4-1, N); jj++)
                    for (kk = k; kk<min(k+N/4-1, N); kk++)
                        X[ii][jj] += Y[ii][kk] * Z[kk][jj];
```

A total of 8 * $2(N/2)^3 = 2N^3$ fp operations are involved in this matrix multiplication problem, the same number as before.
Now let’s count the misses. We multiply two matrices of 4x4 submatrices. For instance the computation of the first tile of X is given by:

$$X_{11} = Y_{11} \times Z_{11} + Y_{12} \times Z_{21} + Y_{13} \times Z_{31} + Y_{14} \times Z_{41}$$

16 such tiles of X must be computed. The 3 submatrices (each of size 64x64x8 = 2^{15} =32KB) accessed in each term now all fit in cache. The first term for each X_{ij} must bring in cache matrix tiles X_{ij}, Y_{i1} and Z_{1j} in cache. After that we only need to bring in cache tiles Y_{i2}, Y_{i3}, Y_{i4} and Z_{2j}, Z_{3j}, Z_{4j}. So for each X_{ij}, we have misses for 3 submatrices for the first term, then for the 3 remaining terms we only have misses for 2 submatrices (as X_{ij} stays in cache.)

Thus the total number of misses per tile X_{ij} is the number of misses to reload 3 submatrices plus 3 times the number of misses to reload 2 submatrices. The total number of misses is 9 times the number of components in a tile, which is $N^2/16$ misses. Since there are 16 tiles in X, the total number of misses is $9*16*N^2/16$. Thus the total number of misses is $9N^2$ = 589,824.

3. Tiling (blocking) dramatically reduces the total number of cache misses when the cache size is smaller than the data (or problem) size. The total number of fp operations remains the same,i.e. $2N^3$. The total number of misses without sub blocking the matrices is N^3+2N^2; while it is $9N^2$ with subblocking. The code for matrix subblocking is more complex and the loop overhead is larger as the nested loop is twice deeper. This overhead involves branch handling and some extra integer calculations for loop indexes. However, this overhead is usually much smaller than the penalty caused by long latency cache misses.

Problem 4.8

a.

Each block contains 16 words of 32 bits (4 bytes) each. There is no block re-use in the code. All misses are cold misses and therefore the cache size is irrelevant. Let's look at the time taken by a loop iteration with a load hit and a loop iteration with a load miss.
(i) cache hit:

```
LOOP:   LW R4,0(R3)           (1)
        ADDI R3,R3,stridex4   (1)
        ADD R1,R1,R4          (1)
        BNE R3,R5,LOOP        (1+2flushes)
```

(ii) cache miss:

```
LOOP:   LW R4,0(R3)           (1)
        ADDI R3,R3,stridex4   (1)
        ADD R1,R1,R4          (1+ 30 cache-latency + 1 retry)
        BNE R3,R5,LOOP        (1+2flushes)
```

The times taken by an iteration with a hit or a miss are 6 or 37 (30+1+6) respectively:

If the stride is greater than or equal to 16, every load incurs a cache miss and the average execution time per iteration is 37 cycles.

For every stride less than 16, the miss/hit sequence in a string of references is cyclic. A cycle is the minimum forward distance in the reference string so that a word with the same block offset is accessed again in the blocks that are touched. Since LCM(16,stride) is the amount of memory spanned between two consecutive accesses to the same word in blocks (LCM is the least common multiple) the number of references in a cycle is LCM(16, stride)/stride. Since the stride is less than 16, all blocks accessed in a cycle are contiguous. Thus, the number of misses in each cycle is LCM(16, stride)/16.
For example, if the stride is 1, the length of miss/hit sequence cycles is 16 (or LCM(16,1)/1), and only 1 cache miss occurs (LCM(16, 1)/16) in each cycle. The number of accesses is also the number of loop iterations, i.e., 16. So the average execution time per iteration i when the stride is 1 is (37*1 +6*15)/16 = 7.9375 cycles/iteration.

The average execution time for each loop as a function of the stride is as follows:

i) stride >= 16:
The cache misses on every iteration. The average iteration execution time per iteration is 37 cycles.

ii) stride < 16:
average iteration execution time = $\frac{(Nbofmisses \times 37) + (Nbofhits \times 6)}{Nbofaccesses}$ =

$$\frac{((LCM(16, stride)/16) \times 37) + ((LCM(16, stride)/stride - LCM(16, stride)/16) \times 6)}{(LCM(16, stride)/stride)} =$$

$$\frac{37 \times stride}{16} + 6 - \frac{6 \times stride}{16} = \left(\frac{31}{16} \times stride\right) + 6$$

To obtain this result more directly, observe that, as the length of the reference string *N* grows to infinity, the number of contiguous blocks accessed (i.e. the number of misses) tends to *N*x*stride*/16 on the average. The miss rate therefore tends to *stride*/16. The number of clocks added to an iteration that misses is 31. Thus the execution time is: $6 + \frac{stride}{16} \times 31$.

b.
The number of cycles taken by a loop iteration with a cache hit increases to 7 because the PW instruction consumes one additional cycle.
Because the prefetch instruction is only one iteration ahead of the load instruction and the stride of the prefetch instruction is the same as the stride of the load instruction, the prefetch instruction is beneficial when the prefetch instruction accesses the next cache block. It cuts the penalty of a cache miss by the total number of cycles between the prefetch instruction and the load instruction of the next iteration (i.e., 6 cycles), so that the miss cost is 26 cycles (=32-6). Because the prefetch consumes one cycle, the time taken by an iteration with a miss is 37 + 1 -6 = 32. We just plug this new penalty number into the equation obtained in part a.

i) stride >= 16
As in (a) above, every load incurs a cache miss. However, now, because of the prefetch, the miss latency of the load is reduced to 26 cycles.
average iteration execution time = 32 cycles

ii) stride < 16
average iteration execution time = $\frac{(Nbofmisses \times 32) + (Nbofhits \times 7)}{Nbofaccesses}$

$$= \frac{((LCM(16, stride)/16) \times 32) + (((LCM(16, stride)/stride) - (LCM(16, stride)/16)) \times 7)}{(LCM(16, stride)/stride)}$$

$$= \left(\frac{25}{16} \times stride\right) + 7$$

c.
i) For strides that are greater than or equal to 16, prefetch is always more effective as the execution time of every iteration is 32 cycles, which is less than without prefetch (37 cycles.)

ii) stride < 16
For prefetch to be effective we must have:

$$\left(\frac{25}{16} \times stride\right) + 7 < \left(\frac{31}{16} \times stride\right) + 6$$

$$1 < \left(\frac{6}{16} \times stride\right)$$

$$\frac{16}{6} < stride$$

$$3 \le stride$$

CHAPTER 5

Problem 5.1

a) The first step is to create a local variable my_sum_of_difference that accumulates the sum_of_difference for the work done by each thread. We must also calculate the lower and upper bound of the loop indices for the two nested loops with respect to a given thread. All these changes are reflected in the code at lines 1-3 in the pseudocode of Figure 4. Moreover, to accumulate the sum_of_difference over all threads we use one critical section per loop nest at lines 9b-e and 16b-e.

```
1a low = (pid-1)*N/nproc;/* Exercise 6.1a
1b hi = low + N/nproc;/* Exercise 6.1a
1c while (not!converged) {
2    sum_of_diff = 0;my_sum_of_diff = 0;
3    for (i=low;i<high+1;i++)/* Exercise 6.1a
4       for (j=i%2;j<N;j+=2){
5          oldA = A[i,j]
6          A[i,j] = oldA + A[i-1,j] + A[i,j+1] +
7                   A[i-1,j] + A[i,j-1];
8          my_sum_of_diff += abs(A[i,j] - oldA);
9a        }
9b     BARRIER(B1); /* Exercise 6.1b
9c     LOCK(L1);    /* Exercise 6.1a
9d       sum_of_diff += my_sum_of_diff;
9e     UNLOCK(L1);
10   for (i=low;i<high+1;i++)
11      for (j=(i+1)%2;j<N;j+=2){
12         oldA = A[i,j]
13         A[i,j] = oldA + A[i-1,j] + A[i,j+1] +
14                  A[i-1,j] + A[i,j-1];
15         my_sum_of_diff += abs(A[i,j] - oldA);
16a       }
16b    BARRIER(B2); /* Exercise 6.1b
16c    LOCK(L2);   /* Exercise 6.1a
16d      sum_of_diff += my_sum_of_diff;
16e    UNLOCK(L2);
17a    BARRIER(B3); /* Exercise 6.1b
17b    if (sum_of_diff/(N*N) < TH) converged = 1;
18 }
```

Figure 4 Pseudocode for a parallel algorithm for the solution of a linear system of equations according to Gauss-Seidel.

b) We must insert barriers just before the entrance of the critical sections to make sure that all local sum of difference variables are up to date. Moreover, we have to add a barrier synchronization just after the last critical section and before the test for the convergence criterion to make sure that all partial sums have been accumulated in the sum of difference. These barriers appear at lines 9b, 16b, and 17a.

c) The time for the sequential algorithm is 1024 x 1024 x 10 ns = 10,485,760 ns. In the parallel execution, each thread will handle 1024/16 = 64 rows. 64 rows contain 64K elements. The time for a single sweep for each thread is thus 64 x 1024 x 10 ns. The sequential work in each sweep amounts to two critical sections and three barriers which together correspond to 500 ns. Moreover, it will take 16 x 100 ns = 1600 ns to initiate the threads. Thus, the total parallel execution time is 2100 ns and the amount of parallel work is 64 x 1024 x 10 ns + 500 ns + 2100 ns = 657,960 ns. Speedup is the ratio between the sequential execution time and the parallel execution time:

Speedup = 10,485,760/657,960= 15.93

Problem 5.2

a) The average memory access time is calculated as follows:

Average access time = Hit time + Miss rate x Miss penalty

Private cache organization:
The hit time in the private cache is a single cycle and the miss penalty is 100 cycles according to the assumptions. Since the private cache only can host eight blocks and 16 different blocks are accessed twice in a row, all accesses will result in a cache miss (the miss rate is 100%). Therefore, the average memory access time is:

Average = 1 + 100 cycles = 101 cycles

Shared cache organization:
Since there are four processors there are four banks where each bank contains eight blocks. The shared cache can thus host 4x8 = 32 blocks. As a result, the first 16 accesses will miss and the subsequent 16 accesses will hit (the miss rate is 50%). The average memory access time is

Average = 2 + 0.50x100 cycles = 52 cycles

The shared cache is almost twice as fast as the private cache.

b)Private cache organization:
Since each processor accesses 16 different blocks twice in a row, all accesses will miss and the average memory access time will be the same as in a) for the private cache.

Shared cache organization:
One of the processors (say processor 1) will miss on each of the first 16 accesses but will bring the blocks into the shared cache. All other processors (say processor 2, 3, and 4) will hit on each access. Hence:

Average memory access time for processor 1: (16 x (2 + 100) + 16 x 2)/32 cycles = 52 cycles
Average memory access time for processors 2-4: 2 cycles
Average memory access time = (52 + 3 x 2)/4 = 14.5 cycles

The shared cache is more than 7 times faster.

c) From the assumptions we note that each processor accesses the following blocks:
Processor 1: Blocks 0 through 7 twice
Processor 2: Blocks 8 through 15 twice

Processor 3: Blocks 16 through 23 twice
Processor 4: Blocks 24 through 31 twice

Private cache organization:
Since there are eight blocks in the private cache, the first eight accesses will miss whereas the next eight accesses will hit. Hence:

The average memory access time = (1+100 + 1)/2 cycles = 51 cycles

Shared cache organization:
The exact same miss behavior shows up for the shared cache; the first eight accesses from each processor will miss whereas the next eight accesses will hit. However, the cache hit time is two cycles.

The average memory access time = (2 + 100 + 2)/2 cycles = 52 cycles

d) Private cache organization:
According to the assumptions the eight blocks are already fetched into cache. The eight processor writes result in eight invalidations where each invalidation takes 10 cycles. The subsequent eight processor reads result in coherence misses where each coherence miss takes 10 cycles.

Average memory access time = 1 + 10 cycles = 11 cycles

Shared cache organization:
An advantage of a shared cache is that no cache coherence actions are needed. Each access will take two cycles:

Average memory access time = 2 cycles

e) There was not a single case where the private cache organization outperforms the shared cache. Yet, the disadvantage of the shared cache is its longer cache hit time. Obviously the tradeoff is between the miss rate and hit time. Let's assume that Shared_Hit_Time =KxPrivate_Hit_Time. The private cache organization has an advantage when

Private HT + Private MR x MP < K x Private HT + Shared MR x MP

Private MR x MP/Private HT - Shared MR x MP/Private HT < K - 1

MP(Private MR - Shared MR)/Private HT < K - 1

Assuming that the hit time for the shared cache (Shared HT) is K times that of a private cache and the miss penalty (MP) is 100 times that of the private hit time (Private HT). Then

Private MR - Shared MR < (K-1)/100
K>(Private_MR-Shared_MR)+1
Providing a fast access to a shared cache is not trivial. Also, contention effects which we have not taken into account will make the hit time for the shared cache longer. That's why private first-level caches are the preferred solution whereas having a shared secondary or tertiary cache can provide the benefits shown in this exercise.

Problem 5.3

Let's number the processors 1,2,...,8. The requests from each processor are issued as follows:

	P1	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**
slot 1	W1	W2	W3	W4	W5	W6	W7	W8
slot 2	R1	R2	R3	R4	R5	R6	R7	R8

The bus arbitration scheme will order the write (Wi) and read (Ri) requests from the processors as follows:

W1, W2, R1, R2, W3, W4, R3, R4, W5, W6, R5, R6, W7, W8, R7, R8.

Since each processor writes a value that corresponds to its identity, the following will be returned: R1 and R2 return 2; R3 and R4 return 4; R5 and R6 return 6; and R7 and R8 return 8.

Problem 5.4

a) According to the assumptions, the fixed-priority arbitration scheme will give the requests access to the bus in the following order: R1/X, R2/X, R3/Y, R4/X, W1/X, R2/X, R3/Y, R4/X.

Starting with R1, it will first issue a snoop action (5 cycles) and get the block from the next level (40 cycles). In total, the block is installed after 45 cycles. We note that R2, R3, and R4 take the same amount of time. Hence, the first four requests take 4 x 45 cycles = 180 cycles.

The write request (W1) involves a snoop action and a bus update which add up to 15 cycles. The next request, R2/X will involve a snoop action and a read request serviced by the next level because the block was invalidated by processor 1. It will thus take 5 + 40 cycles = 45 cycles. R3/Y will hit in the private cache of processor 3 but because the tag directory is not duplicated there will be a delay of 5 cycles. This does not affect the total time though as it is done in parallel with the requests from the other processors. Finally R4/X will miss because of the invalidation by processor 1 and will add another 45 cycles. The last four requests in the sequence take 15 + 45 + 45 cycles = 105 cycles. In total, it takes 180 + 105 cycles = 285 cycles until the entire sequence of requests is completed.

b) Processor 1 will be done after W1/X is completed: 180 + 15 cycles = 195 cycles.

Processor 2 will be done after the second R2/X is completed: 180 + 15 + 45 cycles = 240 cycles.

Processor 3 will be done after the second R3/Y is completed. This request can be carried out right after the first R3/Y and will only take a single cycle as the tag directory is duplicated. Hence, 45 + 45 + 45 + 1 cycles = 136 cycles.

Processor 4 will be done 45 cycles after processor 2. Hence, it is done after 285 cycles.

Problem 5.5

a) MSI: The first request will have to be serviced by the next level and will take 45 cycles. The second request will be a bus update and a snoop action and will take 15 cycles. Now the block is in state M in the private cache of processor 1 so the third request will not incur any cost. We note that the same sequence of events is repeated for processor 2, 3 and 4. Hence, the entire sequence will take 4(45 + 15) cycles =240 cycles.

MESI: The first request will bring the block into the private cache of processor 1 in state E. This means that the second write can be carried out locally and there will be a state transition to M. As in the MSI protocol, the second write request can be carried out locally. This means that the write from processor 2 can gain access to the bus after 45 cycles. When R2 has brought a block into cache (45 cycles), the state of the block is S. Therefore, the write request from processor 2 causes a bus upgrade and a snoop (15 cycles). The same sequence of events is needed for processor 3 and 4. Hence, the time to carry out the entire sequence is 45 + 60 + 60 + 60 cycles = 225 cycles.

b) **MSI:** We simply count the number of requests in each category:
Read request serviced by next level: 1 (traffic: 6 + 32 bytes = 38 bytes)
Bus update: 4 (traffic: 4 x 10 bytes = 40 bytes)
Read request serviced by private cache: 3 (traffic: 3 (6 + 32) bytes = 114 bytes)
Total traffic: 38 + 40 + 114 bytes = 192 bytes

MESI:
Read request serviced by next level: 1 (traffic: 6 + 32 bytes = 38 bytes)
Bus update: 3 (traffic: 3 x 10 bytes = 30 bytes)
Read request serviced by private cache: 3 (traffic: 3 (6 + 32) bytes = 114 bytes)
Total traffic: 38 + 30 + 114 bytes = 182 bytes

Problem 5.6

a) **MSI:** Same as in 5.5a

MOESI: The MSI and MESI protocols involve the next level of the hierarchy on each read request that does not hit in the private cache. On the other hand, under MOESI, the owner will respond with a block copy. The first three requests will be handled in the same way as in the MESI protocol and take 45 cycles. As for the requests from processor 2 the read request will be serviced by the private cache of processor 1 and will take only 25 cycles. The write request will take 15 cycles. Hence, the time to carry out the requests from processor 2 is 40 cycles. It will take the same time to complete the requests from processor 1 and 2. Hence, the total time is 45 + 3 x 40 = 165 cycles. Note that this is significantly shorter than the time it takes under the MESI protocol in Problem 5.5.

b) Same as in 5.5b

c) Note that this is significantly shorter than the time it takes under the MESI protocol in Problem 6.6. This is because MOESI supports cache-to-cache transfers unlike the MESI protocol where the next (slower) level of the memory hierarchy has to be involved. The traffic is the same though.

Problem 5.7

a) MESI w/o read snarfing: The first four requests will take 45 cycles each to carry out yielding 180 cycles. The write by processor 2 will lead to a bus update which takes 15 cycles. The subsequent read requests from processor 1 and 3 will take 45 cycles each. This adds another 105 cycles. Finally, the last three requests will take the same amount of time. Hence, the time for the entire sequence is 390 cycles.

MESI w/ read snarfing. Read snarfing will not have any impact on the first four requests. Hence they will also take 180 cycles. As for the subsequent three requests, we note that the read request from processor 1 will bring the block into the private cache of processor 1 as well as 3 thanks to read snarfing. This adds 15 + 45 cycles = 60 cycles. For the same reason, the time it takes to deal with the last three requests is the same. Therefore, the time for the entire sequence is 180 + 60 + 60 cycles = 300 cycles. This is significantly better than for the MESI protocol.

b) Traffic for MESI w/o read snarfing:
Number of read requests: 8 (traffic: 8(6 +32) bytes = 304 bytes)
Number of bus updates: 2 (traffic: 2 x 10 bytes = 20 bytes)
Total traffic: 324 bytes

Traffic for MESI w/ read snarfing:
Note: We can save the traffic of two read requests.
Number of read requests: 6 (traffic: 6(6 +32) bytes = 228 bytes
Number of bus updates: 2 (traffic: 2 x 10 bytes = 20 bytes)
Total traffic: 248 bytes

Problem 5.8

a) MESI: The access sequence is the same as in Problem 5.5 so the time it takes under MESI is the same.

Write-Update: The first four read requests will take 180 cycles. The write request will result in a bus update and will incur 15 cycles. As for the read requests from processor 2 and 3, they will hit in the cache. The same will apply to the last three requests. Hence, the total time is 180 + 15 + 15 + 4= 214 cycles.

b) MESI: See Problem 5.5b.

Write-Update:
Number of read requests: 4 (traffic: 4(6 + 32) bytes = 152 bytes)
Number of bus updates: 2 (traffic: 2 x 10 bytes = 20 bytes)
Total traffic: 172 bytes

Problem 5.9

a) MESI w/o migratory detection/optimization. The first read request will incur 45 cycles and the subsequent write request will incur no cost as there will be a silent transition from E to M. The subsequent read-modify-write sequences will incur a cost of 45 +15 cycles. Hence the total latency is 45 + 3(45 + 15) = 225 cycles.

MESI w/ migratory detection/optimization. The migratory sharing pattern is detected when processor 2 issues its read request. It means that the read-modify-write sequences from processor 3 and 4 will only have to invoke a read-exclusive request; the subsequent bus update request is eliminated. This means that the first two read-modify-write sequence incurs the same latency as for the basic MESI protocol whereas the last two only incur 45 cycles each. The total latency time is: 45 + 60 + 45 + 45 = 195 cycles.

b) MESI w/o migratory detection/optimization.
Number of read requests: 4 (traffic: 4(6 + 32) bytes = 152 bytes)

Number of bus update requests: 3 (traffic: 3 x 10 bytes = 30 bytes)
Total traffic: 182 bytes

MESI w/ migratory detection/optimization.
Number of read requests: 4 (traffic: 4(6 + 32) bytes = 152 bytes)
Number of bus update requests: 1 (traffic: 10 bytes)
Total traffic: 162 bytes

Problem 5.10

A transient state denoted IM has been added. A processor write to a block in state I will trigger a transition to state IM. Before the access to the bus has been granted a BusRdX or BusUpdate from another cache can show up. It is important then to stay in state IM and issue a BusRdX once access to the bus is granted. At this time, a BusRdX is issued to the bus and once the coherence transaction is completed (Complete) there will be a transition to state M.The state diagram is shown in Figure 5.

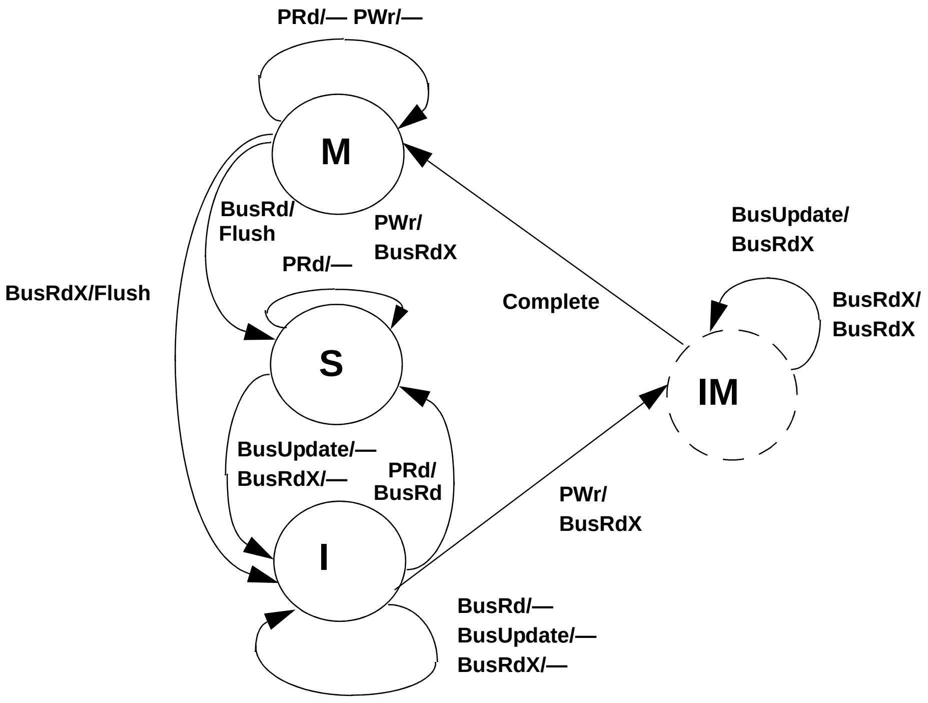


Figure 5

Problem 5.11

	Processor 1	Processor 2	Processor 3	Miss type
1	R_A			cold
2		R_B		cold
3			R_C	cold
4	W_A			

	Processor 1	Processor 2	Processor 3	Miss type
5			R_C	false-sharing miss
6		R_A		true-sharing miss
7	W_B			
8			R_A	true-sharing miss
9			R_B	

a) It is obvious that the first three accesses result in cold misses as it is the first access by each processor to the block. The miss at time slot 5 is a false-sharing miss because word A will not be accessed for as long as the block is in the cache; it is invalidated at time-slot 7. As for the miss experienced by processor 2 at time slot 6, it is obviously a true-sharing miss as it brings in a new value in the cache. For the same reason, the miss at slot 8 is also a true-sharing miss as processor 3 will subsequently access word B.

b) The false-sharing miss at time-slot 5 can be ignored.

c)
Number of read requests: 6 (traffic: 6 x 38 bytes = 228 bytes)
Number of bus updates: 2 (traffic: 2 x 10 bytes = 20 bytes)
Total traffic: 248 bytes

The only non-essential traffic is the read request associated with the false-sharing miss: 38 bytes. The essential traffic is 248 - 38 bytes = 210 bytes. The fraction of essential traffic is 210/248 = 85%

Problem 5.12

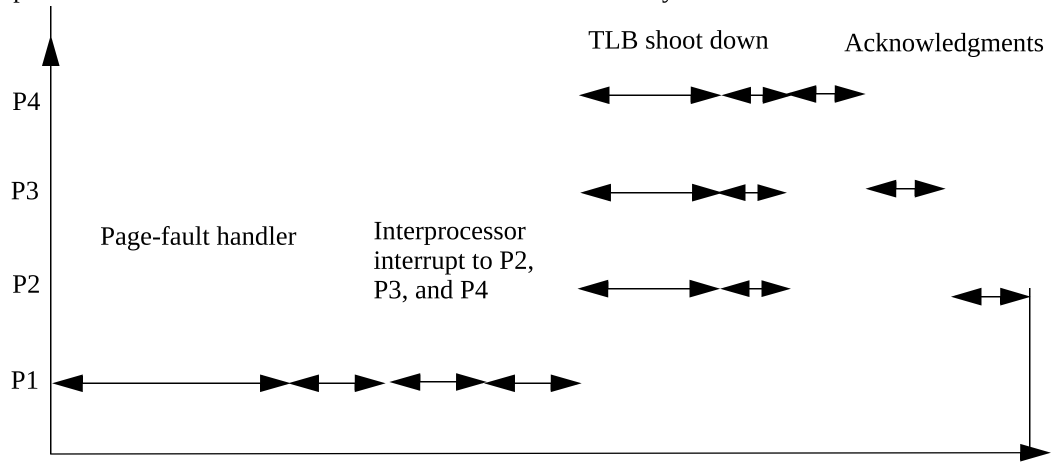
	Processor 1	Processor 2	Processor 3	Miss type
1	R_A			cold miss
2		R_B		cold miss
3			R_C	cold miss
4	W_A			
5			R_D	cold miss; replaces block with A, B & C
6		R_B		false-sharing miss
7	W_B			
8			R_C	replacement miss
9		R_B		true-sharing miss

a) At time-slot 5 the block containing variables A, B and C is replaced. The miss at time-slot 6 is obviously a false-sharing miss as a new word is not accessed while the block is resident in the cache and it ceases to do so at time-slot 7. The miss that occurs at time-slot 8 is a replacement miss rather than a false-sharing miss because the block was overwritten (replaced) by the block containing variable D at time-slot 5. Finally, it is obvious that the miss at time-slot 9 is a true-sharing miss as it brings a new value (B) into cache.

b) Only the false-sharing miss that occurs at time-slot 6.

Problem 5.13

The actions taken at the processor that initiates the TLB shootdown operation (P1) and locally at each processor (P2, P3, and P4) are shown in the diagram below. Given the assumptions the entire operation takes 1000 + 3 x 100 + 200 + 20 + 3 x 100 = 1820 cycles.



Problem 5.14

a) When the home node is the same as the requesting node the miss penalty is the same as the time it takes to process a directory request which is 50 cycles according to the assumptions.
Cache miss time: 1 + 50 = 51 cycles.
Traffic: There is no traffic outside the node.

b) In the case when the memory copy is dirty some other node will have to service the cache miss. The time it takes to do that involves a directory lookup (50 cycles) at the home node, a remote read request (20 cycles), a lookup at the remote node (50 cycles), a flush operation that brings a copy of the block back to the home node, which is the same as the local node (100 cycles), and finally installing the block copy (50 cycles). Hence,
Cache miss time: 1 + 50 + 20 + 50 + 100 + 50 cycles = 271 cycles
Traffic involves sending a remote read request and flushing the block.
Traffic: 6 + 6 + 32 = 44 bytes

c) This scenario involves sending a read request to the home node (BusRd), doing a directory lookup, flushing the block back, and installing it at the requesting node.
Cache miss time: 1 + 20 + 50 + 100+ 50 cycles = 221 cycles
Traffic: 6 + 6 + 32 bytes = 44 bytes

d) This scenario involves sending a read request to the home node, doing a directory lookup, flushing the block back to the requesting node, and installing it there.
Cache miss time: 1 + 20 + 50 + 100 + 50 cycles = 221 cycles
Traffic: 6 + 6 + 32 bytes = 44 bytes

e) This scenario involves sending a read request to the home node (20 cycles), doing a directory lookup (50 cycles), sending a remote read request to the remote node (20 cycles), doing a lookup at the remote node (50 cycles), flushing the block back to the home node (100 cycles), processing it at

the directory (50 cycles), flushing the block back to the local node (100 cycles) and installing it there (50 cycles).
Cache miss time: 1 + 20 + 50 + 20 + 50 + 100 + 50 + 100 + 50 cycles = 441 cycles
Traffic: 6 + 6 + 6 + 32 + 6 + 32 bytes = 88 bytes

Problem 5.15

a) Same as in Problem 5.14a.

b) Same as in Problem 5.14b

c) Same as in Problem 5.14c

d) Same as in Problem 5.14d

e) This scenario involves sending a read request to the home node, doing a directory lookup, sending a request to the remote node, flushing the block back to the requesting node, and installing it there.
Cache miss time: 1 + 20 + 50 + 20 + 50 + 100 + 50 cycles = 291 cycles
Traffic: 6 + 6 + 6 + 32 bytes = 50 bytes

f) The time to process a cache miss will be shorter only in the case when the requesting node, the home node, and the remote node are all different. In that case a four-hop transaction is converted into a three-hop transaction.

Problem 5.16

a) This scenario involves doing a directory lookup and marking the block as dirty and making a transition from state S to state M in the private cache.
Cache write time: 1 + 50 cycles = 51 cycles
Traffic: none.

b) This scenario involves doing directory lookup at home which is the same as the requesting node(50 cycles), send out invalidations to four nodes (20 cycles), having each remote node process the invalidation (50 cycles), sending an acknowledgment back (20 cycles), and finally changing the state to Modified (50 cycles). We assume that invalidations happen in parallel.
Cache write time: 1 + 50 + 20 + 50 + 20 + 50 = 191 cycles
Traffic: 4 x 6 bytes = 24 bytes

c) This scenario involves sending a request to the home node, doing a directory lookup, sending a response back to the requesting node.
Cache write time: 1 + 20 + 50 + 20 + 50 = 141 cycles
Traffic: 6 + 6 bytes = 12 bytes

d) This scenario involves sending a request to the home node, doing a directory lookup, invalidating the local copy, flushing the block and responding with an acknowledgment, and finally changing the state to Modified.
Cache write time: 1 + 20 + 50 + 100 + 50 cycles = 221 cycles
Traffic: 6 + 6 + 32 bytes = 44 bytes

e) This scenario involves sending a request to the home node (20 cycles), doing a directory lookup (50 cycles), having the home node send a remote read-exclusive request to the remote node (20 cycles), having the remote node process the invalidation (50 cycles), flushing the block back to the home node (100 cycles), processing it at the home node (50 cycles), and finally responding back to the local node (100 cycles) where the block is installed in state M in the private cache (50 cycles).
Cache write time: 1 + 20 + 50 + 20 + 50 + 100 + 50 + 100 + 50 cycles = 441 cycles
Traffic: 6 + 6 + 6 + 32 + 6 + 32 bytes = 88 bytes

Problem 5.17

a) Intra-cluster protocol:
A directory entry is associated with each 2-level cache block frame. Since there are eight processors per cluster, each presence-flag vector consists of eight bits. With a 32-byte block size, the overhead becomes 8/(32x8) = 1/32 = 0.03. The overhead is approximately 3%.

Inter-cluster protocol:
Since there are 128 processors and eight processors per cluster, there are 16 clusters. Therefore, each directory entry at the memory contains 16 bits and yields twice as high an overhead as inside each cluster: Approximately 6%.

b) A limited-pointer directory protocol with two pointers inside each cluster and a coarse-vector directory protocol that partitions the clusters into groups of four clusters in each across clusters.

Intra-cluster protocol:
With eight processors per cluster each pointer contains 3 bits. With two pointers per 2-level cache block frame, the overhead is 6/(8 x 32) = 0.023. The overhead is approximately 2%.

Inter-cluster protocol:
There are 16 clusters. If we group them into groups of four clusters each, there are 4 groups and each presence flag in the coarse vector represents a group. There will be four bits in the presence flag vector which yields an overhead of 4/(8 x 32) = 1/64 = 0.016. The overhead is approximately 1.6%.

Since there are 128 processors and eight processors per cluster, there are 16 clusters. Therefore, each directory entry at the memory contains 16 bits and yields almost 3X as high an overhead as inside each cluster: Approximately 6%.

c) Intra-cluster protocol:
A pointer that points to one of the 1-level caches is associated with each 2-level cache block frame. This pointer contains log2 8 bits = 3 bits. Since two pointers are associated with each first-level cache block frame, the pointers will occupy 6 bits.
Overhead at the second-level cache: 3/(8 x 32) = 0.015. The overhead is approximately 1.5%
Overhead at the first-level cache: 6/(8 x 32) = 0.03. The overhead is approximately 3%

Inter-cluster protocol:
Coherence is maintained using a limited-pointer scheme with four pointers across clusters. Since there are 16 clusters, each pointer contains four bits. With four pointers per memory block the overhead is 16/(8 x 32) = 1/16 = 0.0625. The overhead is approximately 6%.

Problem 5.18

In the cache-centric protocol an invalidation has to propagate down the list of sharers in a serial fashion. If the writing node is part of the list it first has to be inserted as the first element in the list. This action involves sending a request to the home node. The home node will respond with a pointer to the current head. The home node will also send a request to the current head and ask it to link to the writing node. When the writing node has been established as the new head, it will send an invalidation request to its tail node (the old head node) which recursively send an invalidation request to its tail node:

1. Request to home node: 2a. Response from home node. 2b Request to head node. 3. Response from old head node to writing node. 4 Invalidation request to tail node. 4 + N-2 Invalidation of last node in list of sharers. 4 + N - 2 + 1: Acknowledgment back to home.
The total write time: K (N + 3) cycles

Under a presence-flag vector all invalidations are carried out in parallel. Moreover, the writing node does not have to be linked in as the first member of the sharing list. There will be only three hops.
The total write time: 3 K cycles.

Problem 5.19

a) Each row consists of (1024 x 8) / 4096 = 2 pages. 1024/16 = 64 rows in the result matrix is calculated by each thread (running on a distinct processor). Each processor will access two elements from matrix B and C to calculate an element in matrix A. In total, each processor will do 2 x 1024 * 64 =128 K reads and 1/16th of these are local and the rest are to remote memory modules.
Local accesses: 8192 accesses
Remote accesses: 122 880 accesses

b) Let’s assume that a page migration scheme is used. The migration cost is the same as for 16 remote accesses and a page is migrated when the cost of remote accesses exceeds twice the migration cost. How many accesses to the local versus remote memory will each node encounter?

A page will migrate after 32 accesses. With a page size of 4096 bytes, there will be 4096/8= 512 accesses to each page. 32 of the accesses will go to remote nodes whereas 512 - 32 = 480 accesses will be to local because the page will migrate after having encountered 32 remote accesses.

The total number of pages accessed by a single processor is the number of matrices (2) times the number of pages per row times the number of rows, i.e., 2 x 2 x 64 = 256 pages. 15/16th of these pages are allocated remotely, i.e., 240. 32 accesses per page will be to remote memory whereas 480 accesses will be local.

Number of accesses to locally allocated pages: 8192 accesses (as in (a))
Number of accesses to remotely allocated pages **before** migration: 32 x 240 = 7680 accesses
Number of accesses to remotely allocated pages **after** migration: 480 x 240 = 115 200 accesses
In total, 8192 + 115 200 = 123 392 accesses are local and 7680 accesses are remote.

CHAPTER 6

Problem 6.1

A packet contains 128 + 16 = 144 bits. With a phit size of 8 bits, it consists of 18 phits of which 2 occupy the routing address.

a) Recall the equation for the unloaded end-to-end latency:
End-to-end packet latency = Sender OH + Time of flight + Transmission time + Routing time + Receiver OH

The packet must traverse 5 links/switches on its way from the source to destination. Under cut-through switching the first two phits must be received at a switch before a routing decision can be taken. The time to buffer them is equivalent to two network cycles which is 2 ns (two 1-GHz clock periods). Therefore, the Time of flight is 2 ns x 5 = 10 ns. An additional cycle is needed for the first phit to reach the destination. Hence, the Time of flight is 11 ns. The rest of the packet (17 phits) will then be delivered with one phit each network clock cycle (1 ns). Hence, the Transmission time is 17 ns as there are 17 additional phits. The routing algorithm is assumed to not cause any overhead so the Routing time is zero. Cut-through routing does not impose any overhead as the packet is pipe-lined through the switches (with two pipeline stages in each to accommodate the routing address) so the Switching time is zero. Finally, according to the assumptions, the sender and receiver do not impose any overheads. As a result, the unloaded end-to-end latency is 11 + 17 ns = 28 ns.

b) Recall that the Effective bandwidth = Packet size/max(Sender OH, Receiver OH, Transmission time)
The packet size is 128 bits disregarding meta data. The Transmission time is 17 ns. Hence, the Effective Bandwidth is 128/17 Gbits/s = 7.53 GBits/s.

c) With a Sender and Receiver OH of 64 and 128 ns, respectively, the Effective bandwidth will be 128/max(64,128,16) GBits/s = 1 GBit/s.

Problem 6.2

a) With store-and-forwarding instead of cut-through switching the entire packet has to be buffered at each switch before it is transferred to the next. The number of network cycles to buffer a packet at each switch is equal to the number of phits in a packet, which is 18. With five switches, it will take 5 x 18 ns = 90 ns and then one more cycle until the first phit is received at the destination. The rest of the packet will then arrive at the rate of the network clock frequency with one phit per cycle so the transmission time counted in network cycles equals the number of remaining phits which is 17. As a result, the end-to-end latency is 91 ns + 17ns = 108 ns.

b) Each switch has to accommodate buffer space corresponding to the number of phits which is 18.

c) Assuming that the Sender and Receiver overheads are zero, the Effective bandwidth is 128/17 GBits/s = 7.53 GBits/s.

Problem 6.3

a) The network diameter is the longest route (counted in number of links for example) in a particular interconnection network. In a 16-by-16 mesh, the longest route is 15 links in one dimension and then 15 links in the other dimension. In a torus, the end points are connected so this cuts the number of links between the furthest nodes by a factor of two in each dimension. Therefore, the longest route is 16 link traversals in total.

b) If we make a cut across the torus so that it forms two identical network graphs, the cut goes across 32 links; 16 nodes are connected to each other plus the link that connects the end points. With 100 Mbits/s for each link the bisection bandwidth is 3.2 GBits/s.

c) Let's first establish the entire bandwidth across all links. The number of links in an N-by-N mesh is 2N(N-1). Hence, a 16-by-16 mesh has 2 x 16 x 15 = 480 links. For a 16-by-16 torus the number of links that connect the end points is 16+16=32. So in total, there are 480 + 32 = 512 links. The total bandwidth is 512 x 100 Mbits/s = 51.2 Gbits/s. Since the number of nodes is 256, the bandwidth per node is 200 Mbits/s.

Problem 6.4

a) Assuming that the instruction cache hit rate is 100%, a miss rate per instruction of 1% implies that every hundred instruction is a memory load that misses in the cache. With a single-issue processor clocked at 2 GHz and with an ideal CPI=1 (disregarding memory stalls) it means that every 50 ns each processor will experience a cache miss on average. With 4 processors it means that the time between cache misses is 50ns/4 = 12.5 ns. A bus transaction takes one bus cycle (2ns) to take care of the request, 20 ns to have the second-level cache bank to respond to the miss request, and another bus cycle to return the requested cache block (2ns). So in total it takes 24 ns to carry out a miss request and during this time the bus is busy. This will make the time between consecutive misses longer by the same amount. So the time between misses is 50ns + 24 ns = 74 ns. With four processors the time between misses is 74 ns/4 = 18.5 ns. Since 18.5 ns < 24 ns, the non-pipelined bus is fully utilized (100%).

b) With a pipelined split-transaction bus, the bus can be released once the request is buffered at the second-level cache bank. So the bus will only be busy when the request is transferred and when the block is sent back. These two operations only occupy the bus for 2ns + 2ns = 4ns. Since there is a cache miss request every 18.5 ns (see Problem 5.4 a) the bus utilization is only 4/18.5 = 22%.

c) Based on the analysis in a) and b) it is quite clear that the non-pipelined bus is not an option whereas the pipelined bus can accommodate the bandwidth demands of the four processors quite well. Note, however, that this analysis is simplistic and does not take into account queuing delays.

Problem 6.5

a) Let's first derive the end-to-end latency as a function of the switch degree. Given a switch degree of *k*, the number of stages to connect *N* nodes of one kind to *N* nodes of another kind is $\log_k N$. In the table below, the number of stages for a given switch degree is derived. The end-to-end-latency is then derived by multiplying the switch-cycle time with number of stages. For the given assumptions, the minimum end-to-end latency is achieved at a switch degree of 4.

Switch degree	**Switch-cycle time [ns]**	**Number of stages**	**End-to-end latency [ns]**
2	12	6	72
4	15	3	45
8	30	2	60
64	70	1	70

b) See Figure 6.6 using 2-by-2 switches. There will be three stages and 64/4 = 16 switches in each stage.

c) By examining Figure 6.6 highlighting two routes that use the same resources in a MIN with 2-by-2 switches it is possible to identify two routes that use the same resources for a MIN using 4-by-4 switches.

Problem 6.6

a) In the table below we list the network diameter and bisection width for the two topologies at different scales based on the formulas in Table 6.1. As can be seen, the hypercube provides a higher bisection width than the torus at the scale of 64 nodes. In addition, the network diameter grows slower for the hypercube than for the torus.

N	Torus: Network diameter	Torus: Bisection width	Hypercube: Network diameter	Hypercube: Bisection width
4	2	4	2	2
16	4	8	4	8
64	8	16	6	32
256	16	32	8	128

b) As can be seen from the table, the network diameter is 6 at the scale of 64 nodes for the hypercube. The switch degree at that scale is 6 for the hypercube as compared to 4 for the torus.

c) While the torus and hypercube provide comparable bisection widths and network diameters at a low scale, the hypercube provides better scalability in terms of latency and bandwidth. But this is to the expense of a higher switch degree.

Problem 6.7

A 4-ary 3-cube is a three dimensional torus in which there are 4 nodes in each dimension. Hence, each node can be given a coordinate (X,Y,Z) that ranges from 0 to 3 in each dimension. Now, in a torus there are two possible routes in each dimension; one in each direction. Let's assume that node (0,0,0) sends a packet to node (3,0,0). One can either traverse in the positive X direction in three steps or in the negative X direction in one step. When forming the routing address both these routes have to be taken into account.

CHAPTER 7

Problem 7.1

(a)

```
            P1          P2          P3
            A:=1        R1:=A       R2:=B
                        B:=1        R3:=A
```

-NOT coherent.
All possible outcomes are coherent because all accesses to each memory variable are made by different threads. So interleaving of accesses to the same memory variable is arbitrary and all possible outcomes are coherent.
-NOT sequentially consistent
The two stores (A:=1 and B:=1) are in different threads, thus they are not ordered. However P2 and P3 cannot observe the two stores in different orders. According to SC, if P2 reads A=1 and P3 reads B=1 then P3 cannot read A=0. So executions resulting in (R1,R2,R3)=(1,1,0) are not sequentially consistent. In this outcome the load-to-store order has been violated since B:=1 is allowed to perform while the preceding load of A is not globally performed.
-NOT TSO
TSO is more relaxed than sequential consistency primarily because of the relaxation of the store-to-load order. However, the relaxation of the store-to-load order cannot affect this code, since the stores are not followed by loads in P1 and P2. Thus (1,1,0) is also not compliant with TSO.
-NOT weakly ordered
Since WO systems do not order regular loads and stores, all outcomes are possible.

(b) Answer the same question as in (a) for the following program:

```
            P1          P2
            A:=1        B:=1
            R1:=B       R2:=A
```

This is the sequence in Dekker's algorithm.
-NOT coherent.
All possible outcomes are coherent because all accesses to each memory variable are made by different processors. So interleaving is arbitrary.
-NOT sequentially consistent.
This is well-known. All outcomes are SC except for (R1,R2)=(0,0). In this outcome the store-to-load order is violated.
-NOT TSO
All outcomes are possible in TSO. All sequentially consistent outcomes are also TSO. The only invalid SC outcome would be (0,0), which is a valid one under TSO because of the store-to-load order relaxation.
-NOT weakly ordered
Since WO systems do not order regular loads and stores, all outcomes are possible.

(c) Answer the same question as in (a) for the following program:

```
            P1          P2
            A:=1        B:=1
            R1:=A       R3:=B
            R2:=B       R4:=A
```

To be at all correct, this sequence must return R1=1 and R3=1 (because of intra-process dependencies). So all outcomes in which R1 or R3 are 0 are incorrect under any model. Thus the only possible outcomes under ANY model are:
(R1,R2,R3,R4)=(1,0,1,0),(1,0,1,1),(1,1,1,0) or (1,1,1,1). In the following we restrict the discussion to these four outcomes.
-NOT coherent
All four outcomes are coherent. Considering accesses to A in isolation, the store to A is executed by P1, followed by the load of A (which must follow to respect intra-thread dependencies). The load of A in P2 can happen at any time, and thus can return either 0 or 1. The same outcome is true for B.
-NOT sequentially consistent
In SC, both R2 and R4 cannot be 0 at the same time. If P1 executes before P2, the outcome is (1,0,1,1). If P2 executes before P1, the outcome is (1,1,1,0). The outcome (1,1,1,1) is obtained when the two stores in P1 and P2 execute first and then all other accesses execute in any order after them. The only non-SC outcome is (1,0,1,0). Because R2 is 0, the code of P1 must precede P2's in any coherent order. Thus it is impossible for R4 to be 0.

Since all subsequent models are relaxed models, they must accept all SC outcomes. Therefore we focus on the one non-SC outcome only in the following, which is (1,0,1,0).
--NOT TSO
Is (1,0,1,0) a possible outcome of TSO? In TSO, the store-load order is not enforced. However load-load order must be enforced. Additionally, a load returning a value from the same thread's store is not ordered with the store, although it returns its value. Thus the two loads in each thread could be performed before the first store and (1,0,1,0) is possible. Therefore in TSO, all outcomes are possible. Note that, in a relaxed model that does not enforce store-load order and does not allow forwarding (such as IBM370), the stores and the loads in each thread would have to be in process order and (1,0,1,0) would be impossible.This is illustrated in Figure 6.

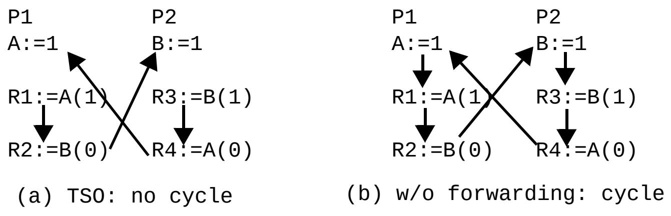


Figure 6. Impact of store to load forwarding

--NOT W.O.
In WO, no order is imposed on regular loads and stores and thus all outcomes are possible.

(d) Answer the same question as in (a) for the following program:

```
P1          P2
A:=1        B:=1
C:=1        C:=2
R1:=C       R3:=C
R2:=B       R4:=A
```

This problem is more complex because up to 3 values can be returned by loads. R1 and R3 could potentially be 0, 1, or 2 and R2 or R4 could be 0 or 1. The total number of possibilities is 3x2x3x2=36. However, any outcome that returns R1=0 or R3=0 is incorrect: Because of intra-thread dependencies, which are enforced in all cases, R1 and R3 must be equal to 1 or 2. So

(0,x,x,x) and (x,x,0,x) are impossible outcomes in all cases. The only possible values for R1 and R3 in all cases are either 1 or 2. This leaves us with 2x2x2x2=16 possible outcomes for all models. We focus on those now.
--NOT coherent.
The only memory location that could cause coherence problems is C, because accesses to A and B are in different threads and so are not ordered by process order. The code restricted to accesses to C is:

```
P1        P2
C:=1      C:=2
R1:=C     R3:=C
```

If we look at all the possible orderings of the 4 accesses to C, the following must be enforced: if R1=2 then R3 must be 2 and if R3=1 then R1 must be 1. So (2,x,1,x) is not coherent.
Therefore, the impossible outcomes under coherence are (0,x,x,x), (x,x,0,x) and (2,x,1,x).
There are 3 possible values of (R1, R3) and 4 possible values of (R1,R2). Thus the number of coherent outcomes is 3x4 = 12 coherent outcomes.
-- NOT sequentially consistent.
SC must be at least coherent so (2,x,1,x) is not SC as well. If we look at the other accesses (on A and B), they are the same as in Dekker’s algorithm. Thus we cannot have (R2,R4) =(0,0) and outcomes (x,0,x,0) are not SC.
Therefore the impossible outcomes under SC are (0,x,x,x), (x,x,0,x), (2,x,1,x) and (x,0,x,0).
Now we have only 3 possible values for (R1,R3) and 3 possible values for (R2,R4). The number of possible outcomes under SC is 3x3.
-- NOT TSO.
TSO relaxes the store-load order, including stores and loads to the same address. TSO is coherent (forwarding store buffer), so that (2,x,1,x) cannot be TSO. However, in TSO, the store of C and the load of C are not ordered. The load of C in P1 may return its value from the store buffer before the store to C has been globally performed. And the load of B in P1 may bypass the stores in the store buffer as well. Thus, (R2,R4) = (0,0) is a possible outcome under TSO, and the set of impossible outcomes under TSO are the same as under coherence.
Therefore, the impossible outcomes under TSO are (0,x,x,x), (x,x,0,x) and (2,x,1,x).
--NOT WO: Since WO systems do not order regular loads and stores, the only impossible outcomes are (0,x,x,x) and (x,x,0,x).

Problem 7.2

(a) This is because the value of BAR is monotonically increasing. If the load reading BAR while it is being updated, it might miss the current increment, but it will eventually observe that BAR reached N.

(b) The correct code is:

```
lock:   LB R1,bar_lock
        BNEZ R1,bar_lock
        T&S R1,bar_lock
        BNEZ R1, lock
        LW R2,BAR
        ADDI R2,R2,#1;
        SW R2,BAR
        SB R0,bar_lock
barrier:LW R2,BAR
        BNE R2,R3,barrier;    /assume (R3)=N
```

(c) With the F&A instruction, the new code is:

```
            F&A R1,BAR
      wait: LW R2,BAR
            BNE R2,R3,wait;/assume R3 contains the value of N
```

(d)
If R3 contains the value of N, then CAS(R3,R0,BAR) implements the condition atomically.
There is no need to implement the condition statement atomically because the processor that reads "BAR=N" is the one that increments BAR for the last time (which can only be done one at a time.) By that time all other processors will be in their while loop.
The code can deadlock. Remember that we cannot make any assumption about the relative speed of processors. For example, the process that arrives at the barrier last and resets BAR to 0 could be suspended right before starting its while loop. During that time other processes may execute the next iteration all the way to the next barrier execution and start incrementing BAR. When the suspended process is awakened, it will see BAR not zero and therefore will busy wait on the first barrier. Since it will never reach the second barrier to increment BAR, all processes will wait on the second barrier for ever.

(e) No solution provided

Problem 7.3

(a) Code for F&A X, Rx, a, where a is a small immediate constant added to X.

```
            F&A(X,Rx,a) LL Rx,X
                        ADDI R1,Rx,a
                        SC R1,X
                        BEQZ R1,F&A
                        return
```

(b) Code for F&A X,Rx,Ry, where Ry is added to X and Rx returns the value of X before the ADD.

```
            F&A(X,Rx,Ry)LL Rx,X
                        ADD R1,Rx,Ry
                        SC R1,X
                        BEQZ R1,F&A
                        return
```

(c) Code for F&A X,Rx,Y, where memory location Y is added to memory location X (X!=Y) and the value of X before the ADD is returned in Rx

```
            F&A(X,Rx,Y) LW R1,Y
                wait:   LL Rx,X
                        ADD R1,Rx,R1
                        SC R1,X
                        BEQZ R1,wait
                        return
```

(d) Code for SWAP(X,Rx), where the values in Rx and X are swapped

```
            SWAP(X,Rx)  ADD R2,Rx,R0/save Rx
                wait:   LL R1,X
                        SC R2,X    /cannot lose Rx
                        BEQZ R2,wait
                        ADD Rx,R1,R0
                        return
```

(e) Code for CAS Rx, Ry, X, where the value in Rx is first compared to the value of X and if they are equal the values in Ry and X are swapped.

```
CAS(Rx,Ry,X)   LL R1,X
               ADD R2,R1,R0/save X
               BNE Rx,R1,exit
               ADD R1,Ry,R0
       exit:   SC R1,X  /store old X or Ry
               BEQZ R1,CAS
               ADD Ry,R2,R0
               return
```

(f) The major problem is that there are two memory stores in this code: one to A and one to B. All stores must be conditional while they are subject to conflicts. In this code the programmer wants the whole code to be a critical section and executed atomically, i.e. B should not be updated if A is updated. Conversely, A should not be updated if B is updated during the if statement. This is very hard (if not impossible) to do with the way LL and SC are defined. Even if an SC monitors the specific address and we execute the two SC's in sequence at the end of the code, an update could still reach the processor for one variable after the SC for the first variable has been already been successful and has updated memory. Too late! It would work if the two SCs could be executed as a critical section.

Problem 7.4

The solution to this problem is not as straightforward as it seems. Looking at Example 7.12, which relies on store synchronization, the answer depends on whether the store-load order is enforced (SC) or not (TSO). Store-load order is enforced by stalling a load until all prior stores in the store buffer have been performed. In Example 7.12, it is shown that provided the store-load order is enforced for different addresses, loads can return values from the store buffer and the memory system is store atomic and sequentially consistent. Thus we differentiate between the cases where store-load orders are enforced when addresses are different and where store-load orders are not enforced.

(a) Coherent: YES. See forwarding example in the text.
Sequentially consistent: NO because loads can bypass stores if store-load orders are enforced on different addresses; NO if they are not enforced.
Store atomic: YES if store-load orders are enforced on different addresses; NO if they are not enforced.
Weakly ordered: YES. Trivial, as regular load/store accesses are not ordered in WO.
TSO: YES. TSO allows store buffers and forwarding from the store buffer.

(b) Coherent: YES. See forwarding example in the text.
Sequentially consistent (SC): In general, NO. The problem is that when a store is combined with a previous store in the store buffer, it effective bypasses the stores that follow the previous store with which it combines, thus violating the FIFO order of stores and the store-store order. One way to deal with this problem is to combine only when the previous store is the last one in the store buffer, but this is very restrictive.
Store atomic: In general, NO. Same problem as for SC and the reasonings based on store synchronization become invalid.
Weakly ordered: YES. Trivial, as regular load/store accesses are not ordered in WO.
TSO: In general, NO. Same reason as for SC

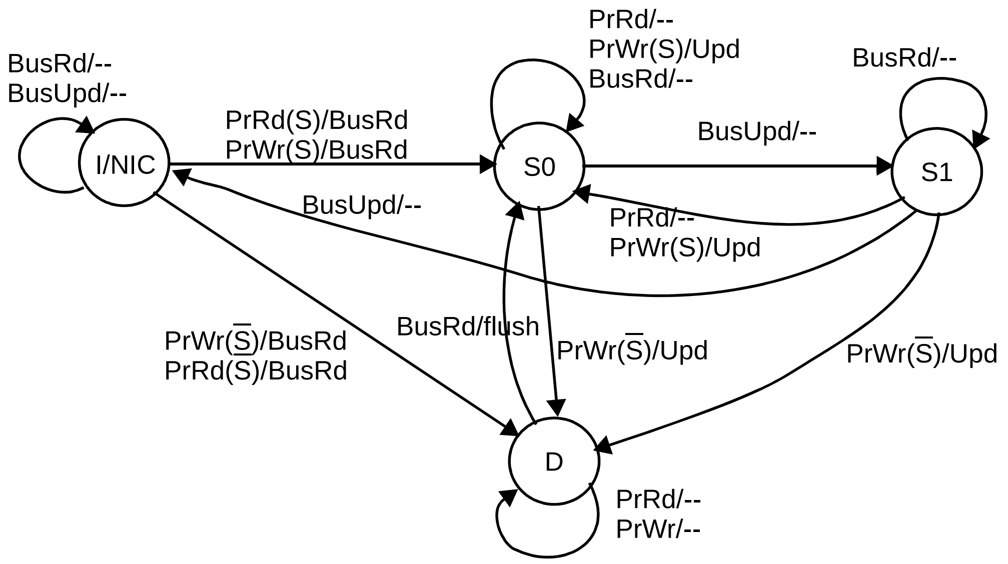
(c) This is incorrect in any model because intra-thread memory dependencies are violated.

(d) Coherent: YES. Values are always returned from a memory with a single copy of every location.
Sequentially consistent: YES if store-load orders are enforced on different addresses; NO if they are not enforced.
Store atomic: YES.
Weakly ordered: YES. Trivial, as regular load/store accesses are not ordered in WO.
TSO: YES. TSO allows store buffers and forwarding from the store buffer. The fact that the values are not returned from the SB complies with TSO.

(e) Coherent: YES. All values are returned from single copies in memory.
Sequentially consistent: YES. Store-load and store-store orders are respected.
Store atomic: YES. Values are returned from memory and memory is atomic.
Weakly ordered: YES. Trivial, as regular load/store accesses are not ordered in WO.
TSO: YES. TSO is weaker than sequential consistency.

(f) Repeat 1-5 under the following condition: Stores in the store buffer are now executed in the memory system as fast as possible in any order, without any regard for their process order.
(f-a) is still coherent provided intra-thread dependencies are respected. Coherence applies to a single address. It is not sequentially consistent because, among other things, store-store orders are violated. It is not store atomic because loads return values that are not globally performed. It is trivially weakly ordered but it cannot be TSO as the store-store order is violated.
(f-b) is still coherent provided intra-thread dependencies are respected. It is not sequentially consistent nor store atomic. It is weakly ordered but it still isn't TSO as, among other things, the store-store order is violated.
(f-c) is incorrect in every case, however the stores propagate.
(f-d) is still coherent. It is not sequentially consistent because of store-store order. It is store atomic as store atomicity deals with a single address, weakly ordered but it is not TSO anymore, since stores can propagate in any order.
(f-e) is still coherent, but it is not sequentially consistent because the stores can propagate out of thread order. It is still store atomic (values are returned from memory and memory is atomic). It is still weakly ordered but it is not TSO, since stores can propagate in any order.

Problem 7.5



Problem 7.6

(a) Assuming the system is sequentially consistent and processors run at the same speed, the memory access sequence will be as follows:
rY1(1) rY1(2) rY1(3) rY1(4) wX1(1) wX2(2) wX3(3) wX4(4) rY2(1) rY2(2) ... wX3(3) wX4(4) <barrier>
rX1(1) rX1(2) rX1(3) rX1(4) wY1(1) wY2(2) wY3(3) wY4(4) rX2(1) rX2(2)... wY3(3) wY4(4)
The numbers between parentheses indicate the thread number. For example, rY1(1) means "Read of Y1 by T1".
These sequences alternate between reads of X and writes of X.

1) MSI invalidate.
We use the state diagram of Figure 7.10(a). In this diagram, a write hit on a Shared copy is treated like a miss and issues a BusRdX that reloads the block in cache. (Instead of that a BusUpgrade could be issued to bus and invalidate all remote copies without reloading the copy).
At the beginning of the loop, each cache has a Shared copy of X and T4 has a Modified copy of Y (according to the problem statement all elements of Y hold in the same memory block).
In the first phase, each thread reads the components of Y, which results in 3 misses (T4 does not miss.) Each following write to X reloads the block containing X and invalidates other caches. These misses are all false sharing misses. Therefore, the total number of misses to write into X is 4*4=16.
Total for the first phase: 19 misses.
The second phase has the same access patterns and the same miss count.
The total number of coherence misses is 19+19=38.

2) MESI.
On every read miss, another copy of the block is present in remote caches. Thus, the cache state E is never visited and MESI behaves like MSI. The total number of misses per iteration is also 38.

3) MSI update.
This protocol has no invalidation, thus copies of X and Y remain in cache across iterations, and because there are multiple copies every time a block is updated the update is propagated on the bus. Copies in the caches of the 4 thread are updated whenever the threads compute the values.
The number of writes in each phase is 16. Thus the total number of bus updates per iteration is 32 updates.

4) Competitive protocol.
After the loop finishes an iteration, T3 and T4 have Y in their cache in state S1 and S0, respectively. The copies in the caches of T1 and T2 were self-invalidated and the copy in T3's cache received an update from T4. After the four reads from T1-4 in the first phase the block of Y in every thread's cache is in state S0 after the first phase because a local read resets the state to S0. In total we have two misses.
When T1 writes to X1 in an iteration, all threads already have the block of X in their cache. Hence, the block of X in T2,T3,and T4 are updated by T1 with UP-bit set to 1 (state S1). After the next memory access, wX2(2), the block of T1 is in state S1 and the block of T2 is in state S0. The blocks of T3 and T4 are invalidated. The write of X by T3 triggers a write miss; the block of T1 is invalidated and the block of T2 moves to state S1. Later, the block of a thread is invalidated every time a thread updates a value in the block of X.
Hence, there is a total of 2 + 14 = 16 read and write misses in the first phase. The number of invalidations is 16. Also, there are 16 updates in the first phase.
In the second phase, all memory access are very similar except that X and Y are switched for read and write, and so the number of update and invalidation misses are same as the number of update and invalidation misses in the first phase.
Therefore, the total number of updates is 32 and the number of invalidation misses is 32 per iteration of the Jacobi algorithm under SC.

(b) Here threads execute all their memory access in turn in each phase. In the first phase thread 1, then thread 2, then thread 3 and finally thread 4 perform all their accesses as a block. Then in the second phase we have the same behavior: thread 1, thread 2, thread 3, and thread 4.

The sequences of memory access are globally performed as follows:
rY1(1) wX1(1) rY2(1) wX1(1) rY3(1) wX1(1) rY4(1) wX1(1) rY1(2) wX2(2) rY2(2) wX2(2) ... rY4(4) wX4(4) <barrier> rX1(1) wY1(1) rX2(1) wY1(1) rX3(1) wY1(1)... rX1(4) wY4(4) rX2(4) wY4(4) rX3(4) wY4(4) rX4(4) wY4(4)

1) MSI invalidate:
At the beginning of the first phase, all cached copies of Y are invalid except in the cache of T4 which has a Modified (Dirty) copy of Y and all cached copies of X are in state Shared.
In the first phase, the first access to Y by T1 is a cache miss. wX1 by T1 causes a BusRdX, which reloads the block in state Modified in T1's cache, is counted as a miss and invalidates X in the caches of T2, T3 and T4. After T1 completes all its memory accesses, the next thread T2 must bring X and Y into its cache. Thread T4 does not need to bring Y into its cache because its copy is still in cache in state Shared (after the first read miss by T1). The number of misses is 4 (for write X) + 3 (for read Y) = 7 in the first phase.
Similarly, there are 7 misses including read and write misses in the second phase.
Hence, the total number of coherence misses per iteration is 14.

2) MESI invalidate:
Once the loop has iterated for a while, a block will never be in state E state because there is always another copy in other threads' cache on a read miss. Therefore, the MESI invalidate protocol behaves like the MSI invalidate protocol.
The total number of coherence misses per iteration is 14.

3) MSI update:
Copies of X and Y remain in all caches in state Shared. All 32 writes must propagate updates.
So the total number of bus updates is 32.

4) Competitive protocol:
In the first phase, T1,T2 and T3 need to bring Y into their cache. Since T4 has modified Y in the second phase of the prior iteration, it has a Modified (dirty) copy in its cache. Hence, there are 3 read misses to Y in the caches of T1,T2 and T3. At the beginning of the first phase, all threads have a copy of X in state S0 because they read X in the second phase of the previous iteration. When T1 modifies X for the first time, it updates remote copies which moves to state S1. Then, on the second modification of X by T1, copies of X in the caches of T2,T3 and T4 are all invalidated. Later, there will be a write miss whenever the next thread modifies for the first time. Therefore, there are 3 write misses for X by T2, T3 and T4 in the first phase. Also, there are 8 updates (4 of which end up invalidating caches).
Similarly, there are 6 misses and 8 updates in the second phase.
Therefore, the total number of misses per iteration is 12 and the total number of updates is 16.

(c) Release consistency.

<barrier> rY1(1) rY2(1)... rY3(4) rY4(4) wX1(1)wX2(2)wX3(3)wX4(4) <barrier> rX1(1) rX2(2) ... rX4(4) wY1(1) wY2(2) wY3(3)wY4(4)

1) MSI invalidate:
At the end of the previous iteration the cache of T4 has a Modified copy of Y. All caches have a shared copy of X.
In the first phase, there are 3 read misses to bring Y in every cache in state Shared. Also there are 4 write misses to X.
Hence, the total number of misses per iteration is 2x7=14.

2) MESI-invalidate:
Again, state E is never entered.
Therefore, the number of misses per iteration in MSI and MESI invalidate are the same,i.e., 14.

3) MSI-update:
All caches have a shared copy. Every thread updates the block of X or Y, and so there are 4 updates in each phase. Thus, the total number of updates per iteration is 8.

4) Competitive protocol:
At the start of the iteration, all threads' caches have a copy of X in state S0. The copies of Y in the caches of T1 and T2 have been invalidated (updated more than once), the copy of Y in the cache of T3 is in state S1 (updated once) and the copy of Y in the cache of T4 is in state S0. Thus the reads of Y miss in the caches of T1 and T2 (2 misses) and at the end of the read sequence all caches have a copy in state S0. The update of X by T1 propagates on the bus and brings the copies in the caches of T2, T3 and T4 into state S1; the update of X by T2 updates the cache of T1 (which moves to S1)

and invalidates the copies in the caches of T3 and T4; the update of X by T3 misses in cache, updates the cache of T2 and invalidates the copy in the cache of T1; finally the update of X by T4 misses in cache, updates the cache of T3 and invalidates the cache of T2.
The total number of misses is 2 read misses (read of Y) and 2 write misses (write of X). The number of updates is 4, one for each write.

The total number of misses per iteration is 8 and the total number of updates is also 8.

Table 59: Summary (Nb. of misses/Nb. of updates)

Interleaving	MSI-Inv/MESI	MSI-Update	Competitive
Sequential Consistency (fine grain)	38/0	0/32	32/32
Sequential Consistency (coarse grain)	14/0	0/32	12/16
Release Consistency	14/0	0/8	8/8

Problem 7.7

Consider the following reference stream:
r1 w1 r1 w1 r2 w2 r2 w2 r3 w3 r3 w3 r2 w2 w2 r2
In the following we designate events/bus transactions as follows:
M: BusRd or BusRdX
H: cache hit
Ud: BusUpd
Ug: BusUpg

1) MSI invalidate protocol

Table 60:

	r1	w1	r1	w1	r2	w2	r2	w2	r3	w3	r3	w3	r2	w2	w2	r2
event	M	H,Ug	H	H	M	H, Ug	H	H	M	H,Ug	H	H	M	H,Ug	H	H
state	S	M	M	M	S	M	M	M	S	M	M	M	S	M	M	M
cycles	150	20	1	1	150	20	1	1	150	20	1	1	150	20	1	1

The total cost of the trace is 689 cycles.

2) MSI update protocol

Table 61:

	r1	w1	r1	w1	r2	w2	r2	w2	r3	w3	r3	w3	r2	w2	w2	r2
event	M	H	H	H	M	H, Ud	H	H,Ud	M	H,Ud	H	H,Ud	H	H,Ud	H, Ud	H
state	M	M	M	M	S	S	S	S	S	S	S	S	S	S	S	S
cycles	150	1	1	1	150	20	1	20	150	20	1	20	1	20	20	1

The total cost of the trace is 557.

3) MESI invalidate protocol

Table 62:

	r1	w1	r1	w1	r2	w2	r2	w2	r3	w3	r3	w3	r2	w2	w2	r2
event	M	H	H	H	M	H, Ug	H	H	M	H,Ug	H	H	M	H,Ug	H	H
state	E	M	M	M	S	M	M	M	S	M	M	M	S	M	M	M
cycles	150	1	1	1	150	20	1	1	150	20	1	1	150	20	1	1

The total cost of the trace is 669.

4) Competitive protocol

Table 63:

	r1	w1	r1	w1	r2	w2	r2	w2	r3	w3	r3	w3	r2	w2	w2	r2
event	M	H,Ud	H	H	M	H,Ud	H	H,Ud	M	H,Ud	H	H,Ud	M	H,Ud	H,Ud	H
state	S0	D	D	D	S0	S0	S0	S0	S0	S0	S0	S0	S0	S0	S0	S0
cycles	150	20	1	1	150	20	1	20	150	20	1	20	150	20	20	1

The total cost of the trace is 745 cycles.
The comparison of the costs is shown in the table below:

Table 64:

Protocol	MSI invalidate	MSI update	MESI invalidate	Competitive
Cost (cycles)	689	557	669	745

In the given reference stream, MSI update has the least cost because it does not need to update other copies when it is the only copy once processor 1 has the block. Also processor 2 does not need to bring the block again after processor 3 modified it because the block in processor 2 is also updated.
In case of MESI protocol, it does not need to use BusUpgrade transaction when the block is in E state, while processor 1 send BusUpgrade transaction when it writes in MSI invalidate protocol.
In the competitive protocol with the given reference stream, the block in processor 2 is invalidated by w3, and so the block is reloaded by r2. Even though the protocol causes more BusUpdate transactions than other protocols, the updates are not helpful in this case. Hence, it has the highest cost among the four protocols.

Problem 7.8

Some explanations are required here here to describe the various scheduling policies.
In *static* tasks are distibuted at the beginning in a round-robin fashion, i.e., T0 to P1, T1 to P2, T2 to P3, T3 to P4, T4 to P1, etc. and each processor executes its set of assigned tasks.
In *semi-static*, tasks are intially distributed as in static. However when a processor runs out of work, it can steal a task that has not been started from other processors.

In *dynamic*, task descriptors are initially inserted in a shared FIFO queue, T0 first and T12 last. Then each processor fetches a descriptor from the FIFO queue and executes the task. When a task is finished, the processor goes back to the queue and fetches the next available task decriptor.
Finally, in *optimum*, task are distributed among processors to minimize the total execution time.

The schedules of tasks are given in Table 65 and Table 66.
The execution time is shown in bold.

Table 65: Schedule of tasks--T(i)=1+i

Strategy	Static				Semi-Static				Dynamic				Optimum			
processor	P1	P2	P3	P4	P1	P2	P3	P4	P1	P2	P3	P4	P1	P2	P3	P4
task	T0	T1	T2	T3	T0	T1	T2	T3	T0	T1	T2	T3	T0	T1	T2	T5
	T4	T5	T6	T7	T4	T5	T6	T7	T4	T5	T6	T7	T3	T9	T7	T6
	T8	T9	T10	T11	T8	T9	T10	T11	T8	T9	T10	T11	T4	T10	T11	T8
	T12				T12				T12				T12			
ex.time	**28**	18	21	24	**28**	18	21	24	**28**	18	21	24	**23**	23	23	22

In the semi-static policy the tasks are allocated initially as in the static policy. Each processor terminates its set of tasks at the times given in the static policy. The earliest processor to finish its work is P2, at clock 18. At that time, P1 has already started to work on T12 (at clock 15), P3 has already started on T10 (at clock 10) and P4 has already started on T11 (at clock 12). Thus task stealing does not alter the schedule. In dynamic, processors finish their work in the same order, after each task, and thus tasks are allocated in the same order as in static. In optimum, the work is allocated so that it is balanced among all processors. It is not possible to balance the work better, given the sizes of the tasks. All practically feasible schedules perform the same at 28 cycles, close to the optimum, at 23 cycles. Thus, in this case, the static policy is the best because it has less dynamic overhead and is feasible, even without prior knowledge of the tasks' execution times.

Table 66: Schedule of tasks--T(i)=13-i

Strategy	Static				Semi-Static				Dynamic				Optimum			
processor	P1	P2	P3	P4	P1	P2	P3	P4	P1	P2	P3	P4	P1	P2	P3	P4
task	T0	T1	T2	T3	T0	T1	T2	T3	T0	T1	T2	T3	T0	T1	T2	T3
	T4	T5	T6	T7	T4	T5	T6	T7	T7	T6	T5	T4	T8	T5	T4	T6
	T8	T9	T10	T11	T8	T9	T10	T11	T8	T9	T10	T11	T9	T10	T11	T7
	T12							T12				T12	T12			
ex.time	**28**	24	21	18	**27**	24	21	19	**24**	23	22	22	**23**	23	22	23

In the case of semi-static, the work is at first distributed as in static. At first T12 is allocated to P1. However P1 finishes T8 at clock 27, P2 finishes T9 at 24, P3 finishes T10 at 21 and P4 finishes T11 at 18. So P4 runs out of work at 18 and can steal T12 from P1. The outcome is a slightly better schedule (27 instead of 28 clocks) over pure static. In this case, dynamic is very close to the optimum schedule.

Summary of execution times and speedups

To compute the speedup, the maximum execution time taken to complete the given tasks by each processor is divided by the execution time of a single processor, which is 91.

T(i) = 1+i

Strategy	Execution Time	Speedup
STATIC	28	3.25
SEMI-STATIC	28	3.25
DYNAMIC	28	3.25
OPTIMUM	23	3.96

T(i) = 13-i

Strategy	Execution Time	Speedup
STATIC	28	3.25
SEMI-STATIC	27	3.37
DYNAMIC	24	3.79
OPTIMUM	23	3.96

Problem 7.9

No solution provided.

CHAPTER 8

Problem 8.1

Let H be the hit rate of the L2 cache and A be the accuracy of the L2-cache hit/miss predictor. The following table gives the penalty associated with the four scenarios.

Table 67: Penalties

Hit/Miss	**Prediction**	**Outcome**	**Fraction**	**Penalty**
Hit	Hit	Success	H*A	0
Hit	Miss	Failure	H*(1-A)	120
Miss	Hit	Failure	(1-H)*(1-A)	60
Miss	Miss	Success	(1-H)*A	20

Therefore the total penalty with the added predictor is:
H*(1-A)*120+(1-H)*(1-A)*60+(1-H)*A*20
The penalty without any predictor is: (1-H)*60.
The predictor is beneficial if (1-H)*60>H*(1-A)*120+(1-H)*(1-A)*60+(1-H)*A*20
or 60-60H>60H-40A-80H*A+60
or A>3H/(2*H+1)
If H=0.50 the predictor is beneficial if A is at least 75%.
If H=0.80 the predictor is beneficial if A is at least 92.3%.

Problem 8.2

Execution schedule for block (coarse grain) multithreading:

Table 68: Block multithreading

Instruction (latency)	Dispatch (issue Q)	Issue	Register fetch	Exec start	Exec complete	CDB	Retire (T1)	Retire (T2)
X1(2)	1(1)	2	3	4	5	6	7	
X2(2)	1(2)	4	5	6	7	8	9	
X3(4)	2(3)	4	5	6	9	10	11	
X4(2)	2(4)	8	9	10	11	12	13	
X5(1,20)	9(3)	10	11	12	12	13	14	
X6(1)	9(4)	11	12	13	13	14	15	
Y1(2)	15(1)	16	17	18	19	20		21
Y2(3)	15(2)	18	19	20	22	23		24
Y3(2)	16(3)	21	22	23	24	25		26
Y4(2)	16(4)	23	24	25	26	27		28
Y5(3)	24(3)	25	26	27	29	30		31
Y6(1)	24(4)	28	29	30	30	31		32

Execution schedule for interleaved (fine grain) multithreading:

Table 69: Speculative scheduling with interleaved multithreading

Instruction (latency)	Dispatch	Issue	Register fetch	Exec start	Exec complete	CDB	Retire (T1)	Retire (T2)
X1(2)	1(1)	2	3	4	5	6	7	
X2(2)	1(2)	4	5	6	7	8	9	
Y1(2)	2(3)	3	4	5	6	7		8
Y2(3)	2(4)	5	6	7	9	10		11
X3(4)	8(3)	9	10	11	14	15	16	
X4(2)	8(4)	13	14	15	16	17	18	
Y3(2)	11(3)	12	13	14	15	16		17
Y4(2)	11(4)	14	15	16	17	18		19
X5(1,20)	17(3)	18	19	**20**	20	21	22	
X6(1)	17(4)	19	20	21	21	22	23	
Y5(3)	19(3)	20	21	22	24	25		26
Y6(1)	19(4)	23	24	25	25	26		27

Looking at the issue, execution start and retire stages, interleaved multithreading is five cycles faster than block multithreading. It the clock rate for interleaved MT processor is 20% slower, the time taken by interleaved MT is 27*1.2=32.4 and is still faster than block MT (33).

Problem 8.3

Interconnection networks available are uni-directional ring (UR), bi-directional rings (BR) and N*N mesh (NM). Coherence protocols supported are snoop-based (SB) and directory-based (DB). A total of 64 cores are available and router latency is 1 cycle. A directory access takes 16 cycles.

To compare let's take both the worst case and the average case latencies. We make the following assumptions: We assume uniform traffic in the network for coherence messages. The latency is only calculated up to the point where the message reaches the target core and proceeds without receiving an acknowledgment. Contention in the network and at the directory is ignored. In SB, each coherence message is sent to all cores, whereas in DB a coherence message is only sent to ONE core and, with the help of directories a requesting core finds out where to send that message.
The latencies are given in Table 70.

Table 70:

		Interconnection network type		
Latency	Coherence type	UR	BR	NM
Worst case	SB	63	32	14
Worst case	DB	16+63	16+32	16+14
Average	SB	63	32	7
Average	DB	16+32	16+32/2	16+7

Problem 8.4

T1 and T2 are concurrent threads. machine A has lock support and machine B has TM support. machine B's clock is 20% longer and the penalty for an aborted transaction is 20 cycles. CPI=1. Assume a lazy TM so that a conflict is detected at the end of a transaction.

(a) Each thread has 10 instructions. The probability of a conflict is 10%. In the lock-based system, two transactions take 20 cycles. In the TM system the time to execute two transactions is:
1.2*(10*0.9+0.1*(10+30))=15.6 cycles. Thus TM is better in this case.

(b) Now each thread has 1000 instruction. The probability of a conflict is 20%. In the lock-based system two transactions take 2000 cycles. In the TM system the two transactions take:
1.2*(1000*0.8+0.2*(1000+1020))=1444.8 cycles. TM is much better in this case.

Problem 8.5

Let's use 1 clock cycle of a core in a 16-way CMP as the baseline clock. Then the clock period of a single core that uses the same power as a 16-core CMP is 1/4 and the clock period of a single core that uses the same power as a 4-core CMP is 1/2 (based on the square relationship).
In the case of the EPI-throttled CMP the time is
((5+15)*1/4+40+40*1/2)=65 cycles.

(a) For a single core the time is:
1/4*((5+15)+16*40+4*40)=205 cycles
The speedup for the EPI-throttled core is 205/65=3.15

(b) For a traditional 16-way CMP the time is:
1*(5+40+40+15)=100
The speedup for the EPI-throttled core is 100/65=1.53

(c) Here power is proportional to frequency. Hence when all cores in a 16-core CMP are active the cycle time is 1. When only one core is active that consumes the same power as the 16-core CMP then its cycle time is 1/16th of the cycle time of 16 active cores in a CMP.
For the EPI-throttled CMP the time is:
((5+15)*1/16+40+40*1/4)=51.25
For the single core, the time is:
1/16*(5+16*40+4*40+15)=51.25. Speedup is 1.
For the traditional 16-way CMP the time is:
1*(5+40+40+15)=100. Speedup is 1.95

Problem 8.6

(a) The convergence test has to be done by a single thread. Let us assume that threads have thread id (or pid) from 1..n. Let thread#1 be the master thread that performs the convergence test.
The following code is run for every thread.

```
/** All Xi and Yi variables are shared. All aij variables are private for each
thread and loaded once at the start of this Jacobi loop. Any remaining vari-
ables are private **/
i =  pid * N / nproc;
While (True) {
    BARRIER(BAR); // BAR was initialized for N threads.
    #pragma omp parallel for private(i), num_threads(nproc)
```

```
        For (i = 0; i < N; ++i) {
            Xi = ai1Y1 + ai2Y2 + ai3Y3 + ai4Y4
        }
        BARRIER(BAR);
        #pragma omp parallel for private(i), num_threads(nproc)
        For (i = 0; i < N; ++i) {
            Yi = ai1X1 + ai2X2 + ai3X3 + ai4X4
        }
        BARRIER(BAR);
        // Perform convergence test in a single thread.
        If (i == 1) {
            If (Convergence_Test()) break;
       }
    }
```

(b) Open MP version of the code

```
    I = pid * N/ nproc;
    While (True) {
        BARRIER(BAR); // BAR was initialized for N threads.
        Xi = ai1Y1 + ai2Y2 + ai3Y3 + ai4Y4
        BARRIER(BAR);
        Yi = ai1X1 + ai2X2 + ai3X3 + ai4X4
        BARRIER(BAR);
        // Perform convergence test in a single thread.
        If (i == 1) {
            If (Convergence_Test()) break;
       }
    }
```

(c) TM version of the code.
Note that TM only provides atomicity. Condition variable checking still must be done.

```
    i =  pid * N / nproc;
    While (True) {
        BARRIER(BAR); // BAR was initialized for N threads.
        TM_START
            Xi = ai1Y1 + ai2Y2 + ai3Y3 + ai4Y4
        TM_END
        BARRIER(BAR);
        TM_START
             Yi = ai1X1 + ai2X2 + ai3X3 + ai4X4
        TM_END
        BARRIER(BAR);
        // Perform convergence test in a single thread.
        If (i == 1) {
            If (Convergence_Test()) break;
       }
    }
```

(d) Which is faster?
OpenMP version is faster than that of TM version. The hurdle for complete parallelization in this code is BARRIERs. OpenMP and TM cannot parallelize BARRIER here. So they can only parallelize internal loops. Internal loops are easily parallelizable, OpenMP is faster as it has less runtime overhead compared to TM.

Problem 8.7

Assume that the backward slice is executed separately and that there are sufficient compute resources to execute the slice. There are a total of 10 instructions in the code segment.

(a) The backward slice for load L2 is as follows:
R2=R7*R4
R4=R3*R4
R5=R4*2
Load R3,0(R5) ==> load L1 (always a hit)
R2=R3+R5
Load R1,0(R2) ==> load L2

The backward slice contains 5 instructions (not counting load L2) and hence the backward slice takes 5 cycles to execute and issue a prefetch. Then 10 cycles later data is guaranteed to be available for load L2. However there are only 9 instructions ahead of load L2 in the code segment. Hence the best one can do is to initiate the slice computation at the start of the code segment. The slice takes 5 cycles to initiate the prefetch. Four cycles later load L2 is executed but the prefetch is already issued and pending and hence four cycles of the miss penalty have been saved.

(b) If load L1 has a 20% miss probability it takes 10 additional cycles to satisfy that miss. Hence the average latency of load L1 is 1*0.8+10*0.2=2.8 cycles. Hence the execution of the backward slice must be initiated at least 16.8 cycles ahead of load L2 in order to completely hide its latency. However, a slice can only be initiated 9 cycles ahead at best, of which 6.8 cycles are needed for slice computation, leaving just 2.2 cycles to cover the latency of load L2. Therefore it is still beneficial to fire the backward slice for load L2 at the beginning of the code segment even if the miss rate of load L1 is 20%.

Problem 8.8

For simplicity assume that each thread executes 200 instructions.
(a) The CPI is (200+10+50)/200 = 1.3

(b) Cache access latencies are hidden because of the total overlap of thread executions with cache misses. The CPI improves to 1.

(c) With a switching overhead of 5 cycles the CPI is now:
(100+5+100+5)/200 = 1.05.
This increase reflects the overhead of switching threads.

(d) CPI(T1) = (100+50+100+500)/200 = 3.75
CPI(T2) = (100+50+100+500)/200 = 3.75
The CPI of the overall machine is 3.75 as well.

Problem 8.9

In this problem we have two variables: the number of cores (P) and the number of L2 cache banks (N). We need to find a configuration that achieves the maximum speedup with respect to the base machine with three cores and nine L2 cache banks, given that area is expanded by 4x. To obtain this optimum, we derive the following equations.

a. Area equation:
Since each core occupies one unit of area and each L2 cache bank occupies three units of area, the total area for the base machine is 30 (i.e., 3 + 27) units of area. With the extra real estate provided for the new machine, we have four times the area of the base machine, which is 120 (i.e., 30 * 4) units of area. Thus the number of cores (P) and the number of L2 cache banks (N) in the new

machine are such that (we must have at least one core and one cache bank):

$$P + 3N = 120 \text{ where } N = \{1, 2, 3, 4, \ldots, 39\},\ P = \{1, 2, 3, \ldots, 117\}$$

b. Speedup equation:
In this problem two kinds of enhancements apply simultaneously. The first enhancement is related to parallel execution and its acceleration factor is equal to P/3. This enhancement improves 60% of workload execution, which means that only 0.60xTbase is affected by this enhancement. The second enhancement is related to the stall time due to L2 misses and its acceleration factor is proportional to $\sqrt{N/3P}$. The second enhancement affects 30% of the execution time of each core in the base machine both in the parallel and serial sections of the execution. We divide the program execution time into four parts.

1-Serial execution and no L2 miss stall. This is 0.28 (i.e., 0.70 * 0.40) of the total execution time: no speedup is possible.
2-Serial execution and L2 miss stalls. This is 0.12 (i.e., 0.30 * 0.40) of the total execution time: speedup is possible by adding cache banks.
3-Parallel execution and no L2 miss stall. This is 0.42 (i.e., 0.70 * 0.60) of the total execution time: speedup is possible by adding cores.
4-Parallel execution and L2 miss stall. This is 0.18 (i.e., 0.30 * 0.60) of the total execution time: both enhancements apply.

Hence the speedup equation is as follows:

$$\frac{1}{0.28 + \frac{0.12}{\sqrt{\frac{N}{3P}}} + \frac{0.42}{\frac{P}{3}} + \frac{0.18}{\frac{P}{3} \times \sqrt{\frac{N}{3P}}}}$$

Now, we have to find the maximum of this expression. One way is to replace P with (120 - 3N) and compute the derivative and find the value of N for which the derivative is zero. Then the closest integer number is the answer.
An easier way is to compute the speedup for different possible number of cache banks and pick the best configuration.
We conclude that the best configuration is N = 34 and P = 18 and the maximum speedup is equal to 1.86 rounded off to the nearest hundredth.

ALTERNATE WAY TO FIND THE SAME RESULT:
Let $T_{P,N}$ be the execution time with P cores and N L2 cache banks. Thus $T_{3,9}$ is the execution time on the baseline system. Let's first compute $T_{P,3P}$. With P cores and 3P L2 cache banks, the stalls due to cache accesses are unchanged since the ratio of cache to core is unchanged. The speedup comes from the parallel section of the code.

$$T_{P,3P} = 0.6 \times \frac{T_{3,9}}{P/3} + 0.4 \times T_{3,9}$$

Next, let's compute $T_{P,N}$ by scaling the number of cache banks from 3P to N while keeping the number of cores at P. The number of cache banks per core in the base is 3. The new number of cache banks per core is N/P. The total stall time due to cache misses is a fraction 0.3 of the total execution time, whatever the parallelism. The acceleration factor for this fraction is $\sqrt{N/3P}$.

$$T_{P,N} = 0.3 \times \frac{T_{P,3P}}{\sqrt{N/3P}} + 0.7 \times T_{P,3P}$$

Substituting for $T_{P,3P}$ we obtain

$$T_{P,N}/T_{3,9} = 0.28 + \frac{0.12}{\sqrt{\frac{N}{3P}}} + \frac{0.42}{\frac{P}{3}} + \frac{0.18}{\frac{P}{3} \times \sqrt{\frac{N}{3P}}}$$

In this expression we substitute P by 120-3N and take the derivative for N and find the same result as in the first solution.

Problem 8.10

In the presence-flag vector protocol, each L2 cache line must have a presence bit per core plus one dirty bit. The total of bits per L2 line is therefore 8+1=9. The number of lines in the entire L2 is 8x3MB/64B=384K lines. Thus the total number of directory bits is 9x384K or 3.456 Mbits.

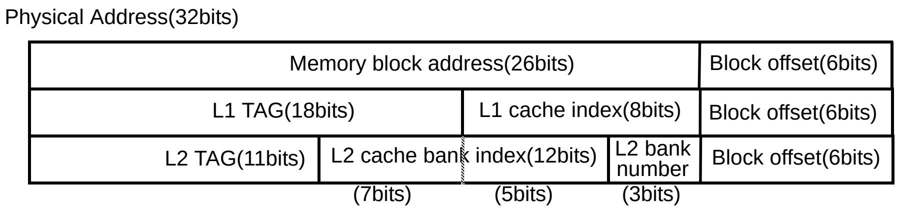


Figure 7

Figure 7 shows various relevant address fields with bit counts obtained from the data given in the problem statement.
With this data, let's consider a directory that replicates the tags in L1 caches. According to the figure, the size of L1 tags is 18 bits. Two state bits (V and D) must be added, for a total of 20 per L1 line. The number of lines in each L1 cache is 1K lines. So the L1-map directory that replicates the L1 cache directory entries contains 20x8x1K = 160Kbits, a significant storage savings of 9x384/20x8 = 22 over the presence-flag directory. Note however that the L1-map directory is now set-associative with a set size equal to the number of cores (8), whereas the presence-flag vector is accessed in parallel with the tag of L2.

Let's now consider the reduced L1-map directory of Figure 8.12. The L2 cache contains $3x2^{17}$ lines. So a total of 19 bits are needed to identify the line in L2 (3 to identify the bank, 12 to identify the set within the bank and 4 to identify one of the 12 ways in the set). However the L1 cache index bits are implicit: 3 bits are used to access the bank and 5 bits are used to access the set in the L1-map directory of the selected bank.

Thus the tag in the L1-map directory needed to identify the line must have 19-8=11bits. Two state bits are also needed, for a total of 13bits. Therefore the total number of bits in the reduced L1-map directory is 13x8x1K = 104Kbits.