Andrew ID:_____

Name:_____

Homework 1
18640 – Fall 2014
due on 9/30/2014

1 Multiple choice questions (2 points each) 10 points

1. RAW register dependencies between instructions
   a. Exist because of pipelined processors
   b. **Are an inherent property of virtually all non-trivial programs**
   c. Will occur only if a pipeline has an earlier register read stage and a later register write stage
   d. Will occur only if a pipeline has a later register read and an earlier register write stage
2. 2. A VLIW instruction set processor
   a. Usually relies on software to resolve pipeline hazards
   b. Can operate at much higher frequency than other approaches
   c. Exposes a lot more instruction-level parallelism than other approaches
   d. Packs multiple operations into a single instruction
   e. **Both (a) and (d)**
3. 3. A branch that is mispredicted as not-taken requires the processor control logic to:
   a. Restart fetching instructions from the not-taken path
   b. Clear out all instructions that were not tagged with the mispredicted branch's tag
   c. Wait for the entire reorder buffer to drain before dispatching new instructions
   d. both (b) and (c)
   e. **None of the above**
4. 11. 4. Local Branch history in a Smith-style dynamic branch predictor is used to:
   a. Predict a branch based on how neighboring branches were resolved recently

b. **Predict a branch based on how that same branch resolved recently**

c. Predict a branch based on the sign bit of its offset field

d. Predict a branch based on a profiling run collected with a representative input set

5. An out-of-order processor could allow a load to issue:

a. As soon as its operands are ready and the load issue port is free

b. Only after all prior store addresses are known

c. Only after all prior stores have retired (written into the cache)

d. **Any of the above**

2 Calculation of CPI 10 points

2.A 5 points

A program's run time is determined by the product of instructions per program, cycles per instruction, and clock frequency. Assume the following instruction mix for a MIPS-like RISC instruction set: 15% stores, 20% loads, 20% branches, and 30% integer arithmetic, 10% integer shift, and 5% integer multiply. Given that stores require one cycle, load instructions require two cycles, branches require four cycles, integer ALU instructions require one cycle, and integer multiplies require ten cycles, compute the overall CPI.

Ans : 2.25

2.B 5 points

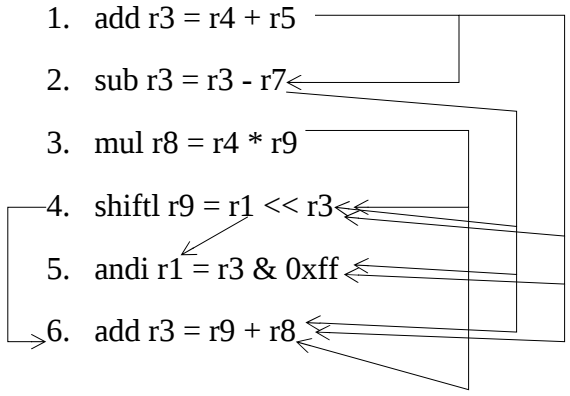
Given the parameters of above problem, consider a strength-reducing optimization that converts multiplies by a compile-time constant into a sequence of shifts and adds. For this instruction mix, 50% of the multiplies can be converted to shift-add sequences with an average length of three instructions. Assuming a fixed frequency, compute the change in instructions per program, cycles per instruction, and overall program speedup.

There are 5 % more instructions per program, the CPI is reduced to 1.97
And overall speed up is 2.25/2.075 = 1.084 or 8.4 %

3 Program Data Dependence Analysis 20 points

Given the following straight-line (no branches, no loops) assembly-language program, identify all program data dependencies that the program contains. Draw all dependencies on the program listing and also specify each dependent pair in the table by including a pair notation entry like {3->6} if instruction 6 has dependence on instruction 3.

1. add r3 = r4 + r5
2. sub r3 = r3 - r7
3. mul r8 = r4 * r9
4. shiftl r9 = r1 << r3
5. andi r1 = r3 & 0xff
6. add r3 = r9 + r8

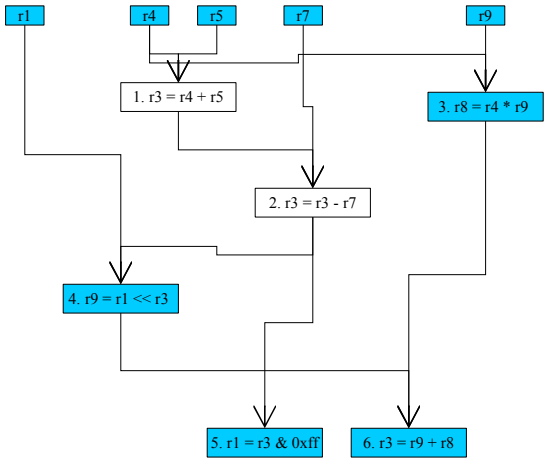


3.A Fill in the table 12 points

RAW	WAR	WAW
1 → 2 2 → 4 3 → 6 4 → 6	3 → 4 4 → 5 5 → 6 4 → 6 2 → 6	1 → 2 2 → 6

3.B 8 points

Given 1 cycle latency for add/sub/shiftl/andi, and 3 cycle latency for mul, how many cycles will it take to execute these instructions at the dataflow limit, assuming all false dependencies are eliminated? *HINT: Draw Data flow graph*



4 cycles

4 Pipelined Processor Performance 20 points

The IBM study of pipelined processor performance assumed an instruction mix based on popular C programs. Object oriented languages like C++ and Java are more common now. One of the effects of these languages is that object inheritance and polymorphism can be used to replace conditional branches with virtual function calls. Given the IBM Instruction mix and CPI shown in the following table, perform the following transformations to reflect the use of C++ and Java, and recompute the overall CPI and speedup or slowdown due to the below changes

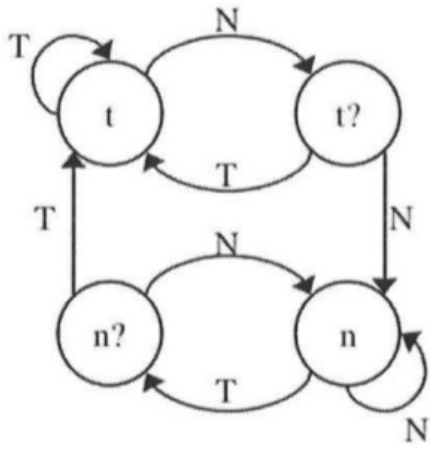
1. Replace 50 % of taken conditional branches with a load instruction followed by a jump register instruction(the load and jump register implement a virtual function call).
2. Replace 25 % of not-taken branches with a load instruction followed by a jump register instruction

Instruction type	Old Mix %	Latency	Old CPI	Cycles	New Mix %	Instructions	Cycles	New CPI
Load	25.0%	2	0.50	500	30.5%	305	610	0.58
Store	15.0%	1	0.15	150	15.0%	150	150	0.14
Arithmetic	30.0%	1	0.30	300	30.0%	300	300	0.28
Logical	10.0%	1	0.10	100	10.0%	100	100	0.09
Branch - T	8.0%	3	0.24	240	4.0%	40	120	0.11
Branch - NT	6.0%	2	0.12	120	4.5%	45	90	0.09
Jump	5.0%	2	0.10	100	5.0%	50	100	0.09
Jump register	1.0%	3	0.03	30	6.5%	65	195	0.18
Total	100.0%		1.54	1540	105.5%	1055	1665	1.58
Increase in CPI: 1.58/1.54 =			1.024805					
Increase in pathlength:			1.0550					
Total slowdown (product of the two)			1.081169	or	8%	slowdown		

5 Branch Prediction Problems 20 points

5.A 8 points

For this part, consider a basic 1-level history-based dynamic branch predictor that employs 2-bit hysteresis counters. The state transition for the 2-bit hysteresis counter is described below in tabular form and as a state transition diagram:



Current State	Actual branch resolution	Predicted branch direction	Next State
"t"	Taken	Taken	"t"
	Not Taken		"t?"
"t?"	Taken	Taken	"t"
	Not Taken		"n"
"n?"	Taken	Not Taken	"t"
	Not Taken		"n"
"n"	Taken	Not Taken	"n?"
	Not Taken		"n"

In the infinite loop below, fill in the conditional expression for the if statement, such that the branch predictor described above will mispredict the branch direction of the if statement, nearly 100 % of the time(You should assume that the branches for the if statement and while loop do not collide in the BTB).

```
{
    j = 0
    do {
        if( (j % 4 ) < 2                                          ) {
            /* some straight-line code that does not change j */
        }
        j++;
    }while(1);
}
```

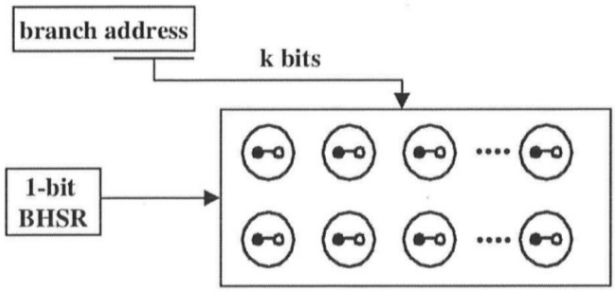
Explain the rationale behind your choice in no more than three sentences.

This automaton will always mispredict the branch if it always has the following behavior: TTNNTTNN...

This is an example of code that generates exactly this behavior

5.B 8 points

In this part, consider a “GA's” 2-level branch predictor with a 1-bit global branch history shift register(BHSR). Each entry of the BHT employs a 1-bit saturating counter FSMIn the infinite loop below, fill in the conditional expression for the if statement, such that the branch predictor described above will mispredict the branch direction of the if statement, nearly 100 % of the time(You should assume that the branches for the if statement and while loop do not collide in the BTB).




```
{
    j = 0
    do {
        if(    (j % 2) == 0                                    ) {
            /* some straight-line code that does not change j */
        }
        j++;
    }while(1);
}
```

Explain the rationale behind your choice in no more than three sentences.

The infinite loop branch is always taken - thus only one counter will be employed for the "if" branch.

That branch is taken every other time, and saturating counter is incapable of predicting such behavior.

5.C 4 points

Write a short pseudo-code segment to demonstrate the utility of the “GA's” predictor in Part B. In your pseudo-code, you need to identify (by circling) one branch that could be predicted much more accurately with the branch predictor in part B as opposed to the branch predictor in part A.

```
while(1) {
        if ( (i%2) ) {
                /*...*/
        }
        if ( !(i%2)) {
                /*...*/
        }
        i++;
}
```

Both "if" branches will be predicted accurately. They are taken every other time, but mutually exclusively, which
is why the first branch selects the correct counter for the second one and vice versa.

6 Two-Level Adaptive Branch Predictor 20 points

6.A 6 points

In this problem you will be given 3 code fragments. For each code fragment, you are to devise the smallest 2-level adaptive branch predictor(i.e fewest number of bits stored) that can correctly predict the direction of every branch in the fragment at over 90% accuracy. You are only allowed to use 1-bit state machines in your predictor. The state transition logic of a 1-bit state machine is summarized in the table below.

Current State	Actual Branch direction	Predicted branch direction	Next State
"t"	Taken	Taken	"t"
	Not Taken		"n"
"n"	Taken	Not taken	"t"
	Not Taken		"n"

In each part, you will state whether you have elected to use Gag, Gap, Gas, Pap, Pap or Pas structure and compute the total number of bits stored by your predictor. In addition, you must provide sufficient detail (in text or figure) to convey your design unambiguously. Important details include the dimensions of the 1-bit prediction-state storage arrays, length of shift-registers, width of pertinent data paths etc. Finally justify your design decision in each part is less than 4 sentences

To simplify the problem, you do not need to consider "address tag" storage overheads. You can assume the addresses of the branch instructions do not collide if your predictor has sufficient capacity(i.e each static branch will map to it's own entry). However, if your predictor does not have sufficient capacity, you cannot make any assumption about how the branches will collide.

Hint: Begin by determining the type of branch behavior exhibited and then choose a structure that best matches that behavior.

```
{
    int j;
    do {
        j = 0;
        do {
            if (j < 50) {
                /**straight-line code that does not affect j**/
            }
            if(j > 50) {
                /**straight line code that does not affect j**/
            }
            j++
        } while(j < 100);
    }while(1);
}
```

Hint: 4 bits

This code requires a degenerate form of "Gap" predictor with a 0-bit-wide global history shift register (i.e. no shift register). The storage array has 1 row (no indexing by the shift register) by 4 "per-branch" columns. This corresponds to the "bi-modal" structure in project 1.

Total storage is 4 bits.

The key to this answer is the "p" in Gap. Every branch resolution is highly repeatable. Thus, a 1-bit machine "per" branch suffices to achieve at over 90% accuracy in this example.

There is only very weak (relatively insignificant) correlation and only between some branches. Thus global history shift register does not help.

6.B 6 points

```
{
    int j;
    do {
        j = 0;
        do {
            if (j <= 3) {
                /**straight-line code that does not affect j**/
            }
            if(j >= 3) {
                /**straight line code that does not affect j**/
            }
            j++
        } while(j < 4);
    }while(1);
}
```

Hint: 32 bits

This code requires a "Pag" predictor with 4 4-bit-wide "per-branch" history shift registers. The storage array has 16 rows (indexed by the selected 4-bit HSR value) by 1 column. This corresponds to the "local" structure in project 1.

Total storage is 16 HSR bits and 16 state-machine bits.

This this case, the key to this answer is the "P" in Pag. Branches 0, 1 and 2 are repetetive but not enough to achieve 90% accuracy using simple schemes. However, each static branch's behavior repeats after exactly 4 dynamic invocations. Thus, giving each branch its own 4-level history allows the Pag predictor to be 100% accurate. For example, if the history is "1xxx" the current branch is always taken, and "0xxx" always means not taken.

6.C 8 points

```
{
    do {
        if (1) { “straight-line code” }                    B0 not taken
        if (0) { “straight-line code” }                    B1 taken
        if (1) { “straight-line code” }                    B2 not taken
        if (0) { “straight-line code” }                    B3 taken
        if (1) { “straight-line code” }                    B4 not taken
        if (0) { “straight-line code” }                    B5 taken
        if (1) { “straight-line code” }                    B6 not taken
    }while(1);                                                 B7
taken
}
```

Hint: 3 bits

This code requires a “Gag” predictor with a 1-bit-wide “global” history shift register. The storage array has 2 rows (indexed by the HSR) by 1 column. This corresponds to the “global” structure in project 1.

Total storage is 1 HSR bits and 2 state-machine bits.

The pattern NTNTNTNTN…… has perfect correlation between adjacent dynamic branches. Thus, only one bit of global history is required for 100% accuracy.