Name: SOLUTIONS   Student ID:

Final Exam

Design of Digital Circuits (252-0028-00L)

ETH Zürich, Spring 2018

Prof. Onur Mutlu

Problem 1 (30 Points):	Potpourri	
Problem 2 (30 Points):	Verilog	
Problem 3 (15 Points):	Boolean Algebra	
Problem 4 (50 Points):	Finite State Machine	
Problem 5 (45 Points):	ISA and Microarchitecture	
Problem 6 (35 Points):	Pipelining	
Problem 7 (45 Points):	Out-of-order Execution	
Problem 8 (40 Points):	Vector Processing	
Problem 9 (45 Points):	GPUs and SIMD	
Problem 10 (40 Points):	Memory Hierarchy	
Problem 11 (35 Points):	Dataflow Meets Logic	
Problem 12 (BONUS: 40 Points):	Branch Prediction	
Total (450 (410 + 40 bonus) Points):		

Examination Rules:

1. Written exam, 180 minutes in total.
2. No books, no calculators, no computers or communication devices. 6 pages of handwritten notes are allowed.
3. Write all your answers on this document, space is reserved for your answers after each question. Blank pages are available at the end of the exam.
4. Clearly indicate your final answer for each problem. Answers will only be evaluated if they are readable.
5. Put your Student ID card visible on the desk during the exam.
6. If you feel disturbed, immediately call an assistant.
7. Write with a black or blue pen (no pencil, no green or red color).
8. Show all your work. For some questions, you may get partial credit even if the end result is wrong due to a calculation mistake.
9. Please write your initials at the top of every page.

Tips:

- **Be cognizant of time. Do not spend too much time on one question.**
- **Be concise. You may be penalized for verbosity.**
- **Show work when needed. You will receive partial credit at the instructors' discretion.**
- **Write legibly. Show your final answer.**

This page intentionally left blank

1 Potpourri [30 points]

1.1 Microarchitecture or ISA? [10 points]

Based on your knowledge of a basic MIPS design and the computer architecture techniques you learned throughout this course, put an "X" in the box corresponding to whether each of the following design characteristics is *better* classified as "microarchitecture" or "ISA":

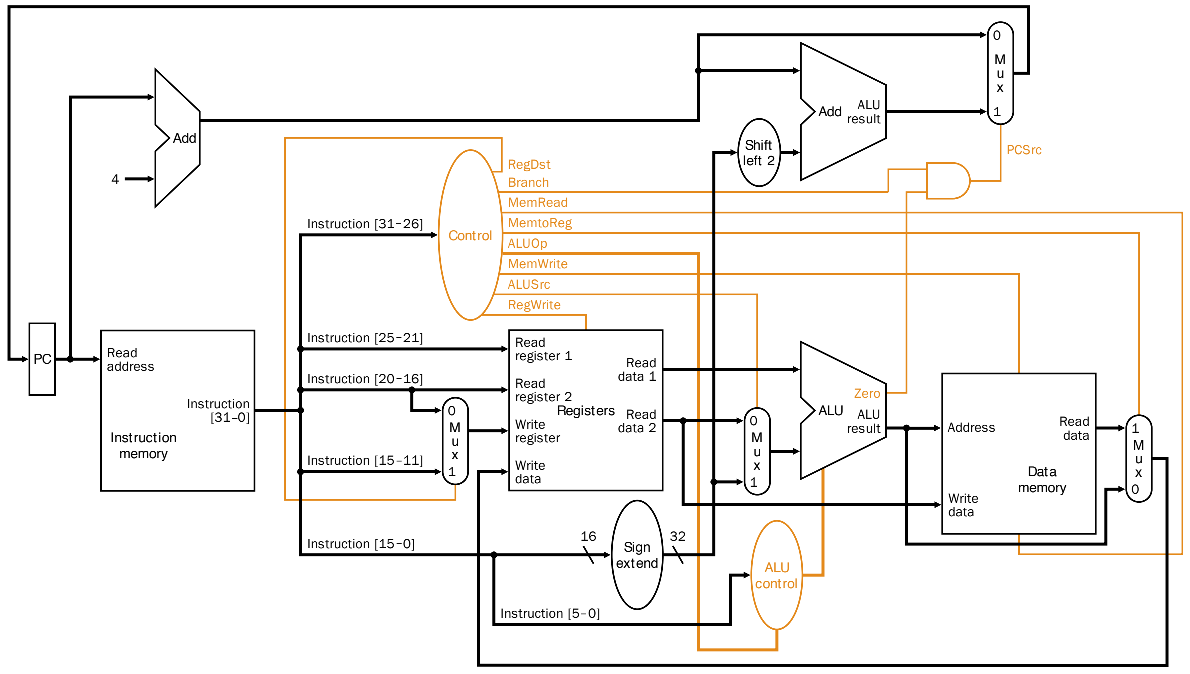
Characteristic	Microarchitecture	ISA
General purpose register \$29 is the stack pointer		X
Maximum bandwidth between the L2 and the L3 cache	X	
Maximum reservation station capacity	X	
Hardware floating point exception support		X
Instruction issue width	X	
Vector instruction support		X
Memory-mapped I/O Port Address		X
Arithmetic and Logic Unit (ALU) critical path	X	
CPU endianness		X
Virtual page size		X

1.2 Single-Cycle Processor Datapath [10 points]

Modify the single-cycle processor datapath to include a version of the `lw` instruction, called `lw2`, that adds two registers to obtain the effective address. The datapath that you will modify is provided below. Your job is to implement the necessary data and control signals to support the new `lw2` instruction, which we define to have the following semantics:

`lw2`: Rd ← Memory[Rs + Rt]
PC ← PC + 4

Add to the datapath any necessary data and control signals (if necessary) to implement the `lw2` instruction. Draw and label all components and wires very clearly (give control signals meaningful names; if selecting a subset of bits from many, specify exactly which bits are selected; and so on).



ALU opcode	Operation
00	Add
01	Subtract
10	Controlled by `funct`
11	Not used

There is no need for new components and wires. The main difference is that the ALU must use "Read data 2", instead of the output of the sign extend unit. The new `lw2` will be R-type, not I-type.
The values of the control signals need to be:
RegDst = 1;
ALUScr = 0;
MemtoReg = 1;
RegWrite = 1;
MemRead = 1;
MemWrite = 0;
ALUop = 00;
Branch = 0.

1.3 Performance Evaluation [10 points]

The execution time of a given benchmark is 100 ms on a 500 MHz processor. An ETH alumnus, designing the next generation of the processor, notices that a new implementation enables the processor to run at 750 MHz. However, the modifications increase the CPI by 20% for the same benchmark.

(a) [4 points] What is the execution time expressed in terms of the number of cycles taken for the **old** generation of the processor (i.e., before the modifications)?

Assuming that the IPC is 2, what is the number of instructions in the benchmark?

Answer: Execution time is **50 Million cycles**. The benchmark has **100 Million instructions**.

Explanation:
Clock frequency is 500 MHz. Then each cycle takes $1/(500 \times 10^{-6}) = 2ns$.
Total execution time in cycles is $100ms/2ns = 50Million$ cycles.

2 instructions per cycle. Then, the total number of instructions: $2x50M = 100M$

(b) [3 points] What is the execution time of the benchmark in *milliseconds* for the **new** generation of the processor?

Answer: 80 ms.

Explanation:
$Execution\ Time = [Number\ of\ Instructions] \times [CPI] \times [Frequency^{-1}]$
Let's say that the CPI of baseline is c, and number of instructions is i.
Then the execution time of baseline:
$(c \times i)/(500x10^6) = 100x10^{-3}\ seconds \quad => \quad (c \times i) = 5 \times 10^7$

The execution time after modifications: $((1.2 \times c) \times i)/(750x10^6)$
$T = ((1.2 \times (c \times i))/(750 \times 10^6)\ seconds.$
$T = ((1.2 \times (5 \times 10^7))/(750 \times 10^6)\ seconds.$
$T = 8 \times 10^{-2} = 80ms.$

(c) [3 points] What is the speedup or slowdown of the new generation processor *over* the old generation?

Answer: 25% speedup

Explanation:
$Speedup\ = (OldExecutionTime\ /\ [NewExecutionTime]) - 1$
$Speedup\ = 100/80 - 1$
$Speedup\ = 0.25$

Then the modification introduces 25% speedup.

2 Verilog

Please answer the following four questions about Verilog.

(a) [6 points] Does the following code result in a sequential circuit or a combinational circuit? Explain why.

```verilog
module concat (input clk, input data_in1, input data_in2,
                                output reg [1:0] data_out);
 always @ (posedge clk, data_in1)
   if (data_in1)
      data_out = {data_in1, data_in2};
   else if (data_in2)
      data_out = {data_in2, data_in1};
endmodule
```

Answer and concise explanation:

Sequential circuit.

Explanation.
This code results in a sequential circuit because `data_in2` is *not* in the sensitivity list, and thus a latch is inferred for `data_out`.

(b) [6 points] In the following code, the input `clk` is a clock signal. What is the hexadecimal value of the output `c` right after the third positive edge of `clk` if initially `c = 8'hE3` and `a = 4'd8` and `b = 4'o2` during the entire time?

```verilog
1  module mod1 (input clk, input [3:0] a, input [3:0] b, output reg [7:0] c);
2  always @ (posedge clk)
3    begin
4      c <= {c, &a, |b};
5      c[0] <= ^c[7:6];
6    end
7  endmodule
```

Please answer below. Show your work.

8'hC4.

Explanation.
Cycle 1: c <= {c, &a, |b} → c <= {1110_0011, 0, 1} → c <= {1000_1101}
c[0] <= ^c[7:6] → c[0] <= ^{11} → c[0] <= 0
At the first positive edge of *clk*, $c = 8'b1000\_1100$
Cycle 2: c <= {c, &a, |b} → c <= {1000_1100, 0, 1} → c <= {0011_0001}
c[0] <= ^c[7:6] → c[0] <= ^{10} → c[0] <= 1
At the second positive edge of *clk*, $c = 8'b0011\_0001$
Cycle 3: c <= {c, &a, |b} → c <= {0011_0001, 0, 1} → c <= {1100_0101}
c[0] <= ^c[7:6] → c[0] <= ^{00} → c[0] <= 0
At the third positive edge of *clk*, $c = 8'b1100\_0100 \rightarrow c = 8'hC4$

Note that since the assignments to c are non-blocking, $c[7:6]$ in line 5 is not affected by the assignment to c in line 4 in the same cycle.

(c) [6 points] Is the following code syntactically correct? If not, please explain the mistake(s) and how to fix it/them.

```
module 1nn3r ( input [3:0] d, input op, output[1:0] s);
  assign s = op ? (d[1:0] - d[3:2]) :
                      (d[3:2] + d[1:0]);
endmodule

module top ( input wire [6:0] instr, input wire op, output reg z);

  reg[1:0] r1, r2;

  1nn3r i0 (.instr(instr[1:0]), .op(instr[7]), .z(r1) );
  1nn3r i1 (.instr(instr[3:2]), .op(instr[0]), .z(r2) );
  assign z = r1 | r2;

endmodule
```

Answer and concise explanation:

The code is not syntactically correct.

Explanation.

- Module names cannot start with a number → '1nn3r' is not a legal module name.
- The output signal 'z' has to be declared as a 'wire' but not 'reg'.
- 'r1' and 'r2' has to be declared as 'wire's.
- The module '1nn3r' does not have ports named 'instr' and 'z'. Those need to be changed to 'd' and 's', respectively.

(d) [6 points] Does the following code correctly implement a counter that counts from 1 to 11 by increments of 2 (e.g., 1, 3, 5, 7, 9, 11, 1, 3 ...)? If so, say "Correct". If not, correct the code with minimal modification.

```verilog
module odd_counter (clk, count);
  wire clk;
  reg[2:0] count;
  reg[2:0] count_next;

  always@*
  begin
    count_next = count;
    if(count != 11)
      count_next = count_next + 2;
    else
      count_next <= 1;
  end

  always@(posedge clk)
    count <= count_next;
endmodule
```

Answer and concise explanation:

No, the implementation is not correct.

Explanation.
The correct implementation:

```verilog
module odd_counter (input clk, output count);
  wire clk;
  reg[3:0] count;
  reg[3:0] count_next;

  always@*
  begin
    count_next = count;
    if(count != 11)
      count_next = count_next + 2;
    else
      count_next = 1;
  end

  always@(posedge clk)
    count <= count_next;
endmodule
```

(e) [6 points] Does the following code correctly instantiate a 4-bit adder? If so, say "Correct". If not, correct the code with minimal modification.

```
module adder(input a, input b, input c, output sum, output carry);
assign sum = a ^ b ^ c;
assign carry = (a&b) | (b&c) | (c&a);
endmodule


module adder_4bits(input [3:0] a, input [3:0] b, output [3:0] sum, carry);
wire [2:0]s;

adder u0 (a[0],b[0],1'b0,sum[0],s[0]);
adder u1 (a[1],s[0],b[1],sum[1],s[1]);
adder u2 (a[2],s[1],b[2],sum[2],s[2]);
adder u3 (a[3],s[2],b[3],sum[3],carry);
endmodule
```

Yes.
Explanation: Even though the wire *s* is swapped with the input *b*, the final computation produced by the module *adder* is still going to be correct since the *or* and *and* operations are commutative.

3 Boolean Algebra [15 points]

(a) [5 points] Find the simplest sum-of-products representation of the following Boolean equation. Show your work step-by-step.

$F = B + (A + \overline{C}).(\overline{A} + \overline{B} + \overline{C})$

Answer: $F = A + B + \overline{C}$

Explanation:
$F = B + (A.\overline{A} + A.\overline{B} + A.\overline{C} + \overline{A}.\overline{C} + \overline{B}.\overline{C} + \overline{C}.\overline{C})$
$F = B + 0 + A.\overline{B} + \overline{C}.(A + \overline{A}) + \overline{B}.\overline{C} + \overline{C}$
$F = (B + A.\overline{B}) + \overline{C}.(A + \overline{A}) + (\overline{B}.\overline{C} + \overline{C})$
$F = (B + A) + \overline{C} + \overline{C}.(\overline{B} + 1)$
$F = A + B + \overline{C}$

(b) [5 points] Convert the following Boolean equation so that it only contains NAND operations. Show your work step-by-step.

$F = \overline{(A + B.C)} + \overline{C}$

Answer: $F = \overline{(\overline{(\overline{(A.A)}.\overline{(B.C)})}.C)}$

Explanation:
$F = \overline{(\overline{(\overline{(A + B.C)} + \overline{C})})}$
$F = \overline{((A + B.C).C)}$
$F = \overline{(\overline{\overline{(A + B.C)}}.C)}$
$F = \overline{(\overline{(\overline{A}.\overline{(B.C)})}.C)}$
$F = \overline{(\overline{(\overline{(A.A)}.\overline{(B.C)})}.C)}$

(c) [5 points] Using Boolean algebra, simplify the following min-terms: $\sum(3, 5, 7, 11, 13, 15)$

Show your work step-by-step.

Answer: $F = D.(B + C)$

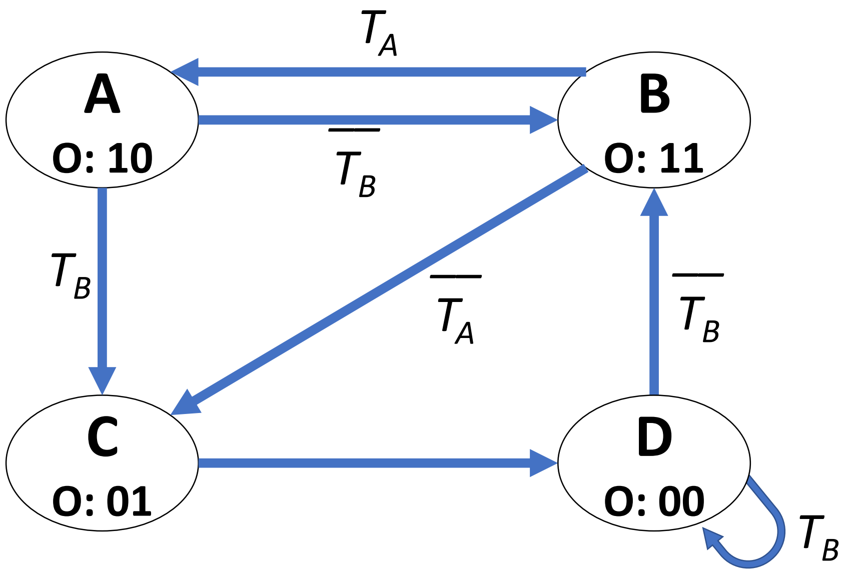
Explanation:

$\{3, 5, 7, 11, 13, 15\} = \{0011, 0101, 0111, 1011, 1101, 1111\}$

$F = (\overline{A}.\overline{B}.C.D) + (\overline{A}.B.\overline{C}.D) + (\overline{A}.B.C.D) + (A.\overline{B}.C.D) + (A.B.\overline{C}.D) + (A.B.C.D)$
$F = (C.D.((\overline{A}.\overline{B}) + (\overline{A}.B) + (A.\overline{B}) + (A.B))) + (B.D.((\overline{A}.\overline{C}) + (A.\overline{C})))$
$F = (C.D) + (B.\overline{C}.D)$
$F = D.(C + (B.\overline{C}))$
$F = D.(B + C)$

4 Finite State Machine [50 points]

You are given the following FSM with two one-bit input signals (T_A and T_B) and one two-bit output signal (O). You need to implement this FSM, but you are unsure about how you should encode the states. Answer the following questions to get a better sense of the FSM and how the three different types of state encoding we dicussed in the lecture (i.e., one-hot, binary, output) will affect the implementation.



(a) [3 points] There is one critical component of an FSM that is *missing* in this diagram. Please write what is missing in the answer box below.

The reset line or indication for initial state.

(b) [2 points] Of the two FSM types, what type of an FSM is this?

Moore

(c) [5 points] List one major advantage of each type of state encoding below.

One-hot encoding	reduces next-state logic
Binary encoding	reduces FFs to hold state
Output encoding	reduces the output logic

(d) [10 points] Fully describe the FSM with equations given that the states are encoded with **one-hot** encoding. Assign state encodings such that numerical values of states increase monotonically for states A through D while using the **minimum** possible number of bits to represent the states with one-hot encoding. Indicate the values you assign to each state *and* simplify all equations:

State assignments: A: 0001, B: 0010, C: 0100, D: 1000
NS[3] = TB * TS[3] + TS[2]
NS[2] = TB * TS[0] + $\overline{TA}$ * TS[1]
NS[1] = $\overline{TB}$ * (TS[0] + TS[3])
NS[0] = TS[1] * TA
O[1] = TS[0] + TS[1]
O[0] = TS[1] + TS[2]

(e) [10 points] Fully describe the FSM with equations given that the states are encoded with **binary** encoding. Assign state encodings such that numerical values of states increase monotonically for states A through D while using the **minimum** possible number of bits to represent the states with binary encoding. Indicate the values you assign to each state *and* simplify all equations:

State assignments: A: 00, B: 01, C: 10, D: 11
NS[1] = $\overline{TS[1]}$ * ($\overline{TS[0]}$ * TB + TS[0] $\overline{TA}$) + TS[1] * ($\overline{TS[0]}$ + TS[0] * TB)
NS[0] = $\overline{TS[1]}$ * $\overline{TS[0]}$ * $\overline{TB}$ + TS[1]
O[1] = TS[1]
O[0] = TS[1] XOR TS[0]

(f) [10 points] Fully describe the FSM with equations given that the states are encoded with **output** encoding. Use the **minimum** possible number of bits to represent the states with output encoding. Indicate the values you assign to each state *and* simplify all equations:

State assignments: A: 10, B: 11, C: 01, D: 00
NS[1] = TS[1] * $\overline{TS[0]}$ * $\overline{TB}$ + TS[1] * TS[0] * TA + $\overline{TS[1]}$ * $\overline{TS[0]}$ * $\overline{TB}$
= $\overline{TS[0]}$ * $\overline{TB}$ + TS[1] * TS[0] * TA
NS[0] = TS[1] * $\overline{TS[0]}$ + TS[1] * TS[0] * $\overline{TA}$ + $\overline{TS[1]}$ * $\overline{TS[0]}$ * $\overline{TB}$
= TS[1] * ($\overline{TS[0]}$ + TS[0] * $\overline{TA}$) + $\overline{TS[1]}$ * $\overline{TS[0]}$ * $\overline{TB}$
O[1] = TS[1]
O[0] = TS[0]

(g) [10 points] Assume the following conditions:

- We can only implement our FSM with 2-input AND gates, 2-input OR gates, and D flip-flops.
- 2-input AND gates and 2-input OR gates occupy the *same* area.
- D flip-flops occupy 3x the area of 2-input AND gates.

Which state encoding do you choose to implement in order to **minimize the total area** of this FSM?

one-hot: 10 logics 4 FFs binary: 16 logics. 2 FFs output: 10 logics. 2 FFs
Output encoding has the least amount of circuitry elements.

5 ISA and Microarchitecture [45 points]

You are asked to complete the following program written in MIPS assembly with a sequence of MIPS instructions that perform **64-bit integer subtraction (A - B)**. The 64-bit integer to be subtracted *from (A)* is loaded into registers \$4 and \$5. Similarly, the 64-bit integer to subtract (B) is loaded into registers \$6 and \$7. Both numbers are in two's complement form. The upper 32-bit part of each number is stored in the corresponding even-numbered register.

```
Loop:  lw  $4, 0($1)
       lw  $5, 4($1)
       lw  $6, 8($1)
       lw  $7, 12($1)

       # 64-bit subtraction
       # goes here

       addi  $1, $1, 16
       j  Loop
```

(a) [15 points] Complete the above program to perform the 64-bit subtraction explained above **using at most 4 MIPS instructions**. (*Note: A summary of the MIPS ISA is provided at the end of this question.*)

A possible sequence of instructions is as follows:

```
subu $3, $5, $7 # Subtract the least significant part
sltu $2, $5, $7 # Check if borrowing is needed
add $2, $6, $2 # Add borrow
sub $2, $4, $2 # Subtract the most significant part
```

(b) [15 points] Assume that the program executes on a pipelined processor, which does *not* implement interlocking in hardware. The pipeline assumes that all instructions are independent and relies on the compiler to properly order instructions such that there is sufficient distance between dependent instructions. The compiler either moves other independent instructions between two dependent instructions, if it can find such instructions, or otherwise, inserts `nop`s. There is *no* internal register file forwarding (i.e., if an instruction writes into a register, another instruction *cannot* access the new value of the register until the next cycle). The pipeline does *not* implement any data forwarding. The datapath has the following *five pipeline stages*, similarly to the basic pipelined MIPS processor we discussed in lecture. Registers are accessed in the Decode stage. The execution stage contains *one ALU*.

(a) Fetch (one clock cycle)
(b) Decode (one clock cycle)
(c) Execute (one clock cycle)
(d) Memory (one clock cycle)
(e) Write-back (one clock cycle).

Reorder the existing instructions and insert as few as possible `nop` instructions to correctly execute the entire program that you completed in part (a) on the given pipelined processor. Show all the instructions necessary to correctly execute the **entire program**.

We reoder the `lw` instructions to first load the data that corresponds to the lower parts of the two numbers since we need the lower part first. We also reorder the completely independent `addi` instruction to hide part of the load latency. We insert sufficient number of `nop` instructions until the register is written before the dependent instruction reads the same register in the decode stage.

This is the resulting code:

```
Loop:  lw $5, 4($1)
       lw $7, 12($1)
       lw $4, 0($1)
       lw $6, 8($1)
       addi $1, $1, 16
       sltu $2, $5, $7
       subu $3, $5, $7
       nop
       nop
       add $2, $6, $2
       nop
       nop
       nop
       sub $2, $4, $2
       j Loop
```

(c) [5 points] What is the *Cycles Per Instruction (CPI)* of the program when executed on the pipelined processor provided in part (b)?

$CPI \approx 1.5$

Explanation.
Since the code is an infinite loop, the number of cycles to fill the pipeline becomes negligible after a large number of iterations. Thus, we can consider that the throughput is one instruction every cycle. We count the number of cycles for one loop iteration. It is 15 for 10 instructions. This way, $CPI \approx \frac{15}{10} = 1.5$.

(d) [10 points] Now, assume a processor with a *multi-cycle* datapath. In this multi-cycle datapath, each instruction type is executed in the following number of cycles: 4 cycles for R-type, 5 cycles for load, 4 cycles for store, and 3 cycles for jump. What is the CPI of the program in part (a) when executed on this multi-cycle datapath? Assuming the multi-cycle datapath runs at the same clock frequency as the pipelined datapath in part (b), how much speedup does pipelining provide?

CPI:

$CPI = 4.3$

Explanation.
For the multi-cycle datapath, we have to take into account the number of cycles for each instruction type: 4 cycles for R-type, 5 cycles for load, 4 cycles for store, and 3 cycles for jump.
Thus, $CPI = \frac{4\times5+5\times4+3\times1}{10} = 4.3$.

Speedup:

Pipelining provides 287% speedup.

Explanation.
We calculate the speedup as follows:
$Speedup = \frac{CPI_{multi-cycle}}{CPI_{pipelined}} = \frac{4.3}{1.5} = 2.87$.

MIPS Instruction Summary

Opcode	Example Assembly	Semantics
add	`add $1, $2, $3`	`$1 = $2 + $3`
sub	`sub $1, $2, $3`	`$1 = $2 - $3`
add immediate	`addi $1, $2, 100`	`$1 = $2 + 100`
add unsigned	`addu $1, $2, $3`	`$1 = $2 + $3`
subtract unsigned	`subu $1, $2, $3`	`$1 = $2 - $3`
add immediate unsigned	`addiu $1, $2, 100`	`$1 = $2 + 100`
multiply	`mult $2, $3`	`hi, lo = $2 * $3`
multiply unsigned	`multu $2, $3`	`hi, lo = $2 * $3`
divide	`div $2, $3`	`lo = $2/$3, hi = $2 mod $3`
divide unsigned	`divu $2, $3`	`lo = $2/$3, hi = $2 mod $3`
move from hi	`mfhi $1`	`$1 = hi`
move from low	`mflo $1`	`$1 = lo`
and	`and $1, $2, $3`	`$1 = $2 & $3`
or	`or $1, $2, $3`	`$1 = $2 | $3`
and immediate	`andi $1, $2, 100`	`$1 = $2 & 100`
or immediate	`ori $1, $2, 100`	`$1 = $2 | 100`
shift left logical	`sll $1, $2, 10`	`$1 = $2 « 10`
shift right logical	`srl $1, $2, 10`	`$1 = $2 » 10`
load word	`lw $1, 100($2)`	`$1 = memory[$2 + 100]`
store word	`sw $1, 100($2)`	`memory[$2 + 100] = $1`
load upper immediate	`lui $1, 100`	`$1 = 100 « 16`
branch on equal	`beq $1, $2, label`	`if ($1 == $2) goto label`
branch on not equal	`bne $1, $2, label`	`if ($1 != $2) goto label`
set on less than	`slt $1, $2, $3`	`if ($2 < $3) $1 = 1 else $1 = 0`
set on less than immediate	`slti $1, $2, 100`	`if ($2 < 100) $1 = 1 else $1 = 0`
set on less than unsigned	`sltu $1, $2, $3`	`if ($2 < $3) $1 = 1 else $1 = 0`
set on less than immediate	`sltui $1, $2, 100`	`if ($2 < 100) $1 = 1 else $1 = 0`
jump	`j label`	`goto label`
jump register	`jr $31`	`goto $31`
jump and link	`jal label`	`$31 = PC + 4; goto label`

6 Pipelining [35 points]

Consider two pipelined machines implementing MIPS ISA, Machine I and Machine II:

Both machines have the following *five pipeline stages*, very similarly to the basic 5-stage pipelined MIPS processor we discussed in lectures, and *one ALU*:

1. Fetch (one clock cycle)
2. Decode (one clock cycle)
3. Execute (one clock cycle)
4. Memory (one clock cycle)
5. Write-back (one clock cycle).

Machine I does *not* implement interlocking in hardware. It assumes all instructions are independent and relies on the compiler to order instructions such that there is sufficient distance between dependent instructions. The compiler either moves other independent instructions between two dependent instructions, if it can find such instructions, or otherwise, inserts `nop`s. Assume internal register file forwarding (an instruction writes into a register in the first half of a cycle and another instruction can correctly access the updated value of the same register in the next half of the cycle). Assume that the processor predicts all branches as *always-taken*.

Machine II implements data forwarding in hardware. On detection of a flow dependence, it can forward an operand from the memory stage or from the write-back stage to the execute stage. The load instruction (`lw`) can *only* be forwarded from the write-back stage because data becomes available in the memory stage but *not* in the execute stage like for the other instructions. Assume internal register file forwarding (an instruction writes into a register in the first half of a cycle and another instruction can access the updated value of the same register in the next half of the cycle). The compiler does *not* reorder instructions. Assume that the processor predicts all branches as *always-taken*.

Consider the following code segment:

```
Copy: lw   $2, 100($5)
      sw   $2, 200($6)
      addi $1, $1, 1
      bne  $1, $25, Copy
```

Initially, `$5` = 0, `$6` = 0, `$1` = 0, and `$25` = 25.

(a) [10 points] When the given code segment is executed on Machine I, the compiler has to reorder instructions and insert `nop`s if needed. Write the resulting code that has *minimal modifications* from the original.

```
Copy:  lw $2, 100($5)

       addi $1, $1, 1

       nop

       sw $2, 200($6)

       bne $1, $25, Copy
```

(b) [10 points] When the given code segment is executed on Machine II, dependencies between instructions are resolved in hardware. Explain **when** data is forwarded and **which instructions** are stalled and **when** they are stalled.

In every iteration, data are forwarded for `sw` and for `bne`. The instruction `sw` is dependent on `lw`, so it is stalled one cycle in every iteration

(c) [5 points] Calculate the *machine code size* of the code segments executed on Machine I (part (a)) and Machine II (part (b)).

Machine I: Machine I - 20 bytes (because of the additional `nop`)

Machine II: Machine II - 16 bytes

(d) [7 points] Calculate the number of cycles it takes to execute the code segment on Machine I and Machine II.

Machine I:

The compiler reorders instructions and places one `nop`. This is the execution timeline of the first iteration:

1	2	3	4	5	6	7	8	9
F	D	E	M	W				
	F	D	E	M	W			
		N	N	N	N	N		
			F	D	E	M	W	
				F	D	E	M	W

9 cycles for one iteration. As there are 5 instructions in each iteration and 25 iterations, the total number of cycles is 129 cycles.

Machine II:

The machine stalls `sw` one cycle in the decode stage. This is the execution timeline of the first iteration:

1	2	3	4	5	6	7	8	9
F	D	E	M	W				
	F	D	D	E	M	W		
		F	F	D	E	M	W	
				F	D	E	M	W

9 cycles for one iteration. As there are 4 instructions in each iteration and 25 iterations, and one stall cycle in each iteration, the total number of cycles is 129 cycles.

(e) [3 points] Which machine is faster for this code segment? Explain.

For this code segment, both machines take the same number of cycles. We cannot say which one is faster, since we do not know the clock frequency.

7 Out-of-order Execution [45 points]

In this problem, we will give you the state of the Register Alias Table (RAT) and Reservation Stations (RS) for an out-of-order execution engine that employs Tomasulo's algorithm, as we discussed in lectures. Your first task is to determine the original sequence of **four instructions** in program order.

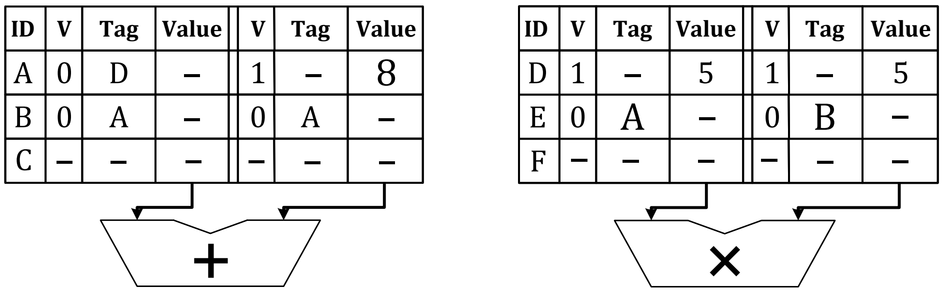
The out-of-order machine in this problem behaves as follows:

- The frontend of the machine has a one-cycle fetch stage and a one-cycle decode stage. The machine can fetch one instruction per cycle, and can decode one instruction per cycle.
- The machine executes *only* register-type instructions, e.g., $OP\ R_{dest} \leftarrow R_{src1},\ R_{src2}$., where OP can be ADD or MUL.
- The machine dispatches one instruction per cycle into the reservation stations, in program order. Dispatch occurs during the decode stage.
- An instruction always allocates the first reservation station that is available (in top-to-bottom order) at the required functional unit.
- When an instruction in a reservation station finishes executing, the reservation station is cleared.
- The adder and multiplier are **not** pipelined. An ADD operation takes 2 cycles. A multiply operation takes 3 cycles.
- The result of an addition and multiplication is broadcast to the reservation station entries and the RAT in the writeback stage. A dependent instruction can begin execution in the next cycle after the writeback if it has all of its operands available in the reservation station entry. There is *only* one broadcast bus, and thus multiple instructions *cannot* broadcast in the same cycle.
- When multiple instructions are ready to execute at a functional unit at the same cycle, the oldest ready instruction is chosen to be executed first.

Initially, the machine is empty. Four instructions then are fetched, decoded, and dispatched into reservation stations. Pictured below is the state of the machine when the final instruction has been dispatched into a reservation station:

RAT

Reg	V	Tag	Value
R0	–	–	–
R1	0	A	5
R2	1	–	8
R3	0	E	–
R4	0	B	–
R5	–	–	–



ID	V	Tag	Value	V	Tag	Value
A	0	D	–	1	–	8
B	0	A	–	0	A	–
C	–	–	–	–	–	–

ID	V	Tag	Value	V	Tag	Value
D	1	–	5	1	–	5
E	0	A	–	0	B	–
F	–	–	–	–	–	–

(a) [15 points] Give the four instructions that have been dispatched into the machine, in program order. The source registers for the first instruction can be specified in either order. Give instructions in the following format: "opcode destination ⇐ source1, source2."

MUL	R1	⇐	R1	,	R1
ADD	R1	⇐	R1	,	R2
ADD	R4	⇐	R1	,	R1
MUL	R3	⇐	R1	,	R4

(b) [15 points] Now assume that the machine flushes all instructions out of the pipeline and restarts fetch from the first instruction in the sequence above. Show the full pipeline timing diagram below for the sequence of four instructions that you determined above, from the fetch of the first instruction to the writeback of the last instruction. Assume that the machine stops fetching instructions after the fourth instruction.

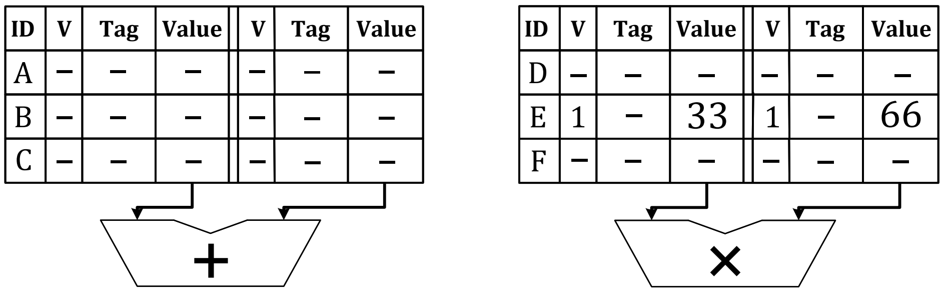
As we saw in lectures, use "F" for fetch, "D" for decode, "En" to signify the nth cycle of execution for an instruction, and "W" to signify writeback. Fill in each instruction as well. You may or may not need all columns shown.

Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
MUL R1 ← R1, R1	F	D	E1	E2	E3	W										
ADD R1 ← R1, R2		F	D				E1	E2	W							
ADD R4 ← R1, R1			F	D						E1	E2	W				
MUL R3 ← R1, R4				F	D								E1	E2	E3	W

(c) [15 points] Finally, show the state of the RAT and reservation stations at the end of the **12th cycle** of execution in the figure below. Complete all blank parts.

RAT

Reg	V	Tag	Value
R0	–	–	–
R1	1	–	33
R2	1	–	8
R3	0	E	–
R4	1	–	66
R5	–	–	–



ID	V	Tag	Value	V	Tag	Value
A	–	–	–	–	–	–
B	–	–	–	–	–	–
C	–	–	–	–	–	–

ID	V	Tag	Value	V	Tag	Value
D	–	–	–	–	–	–
E	1	–	33	1	–	66
F	–	–	–	–	–	–

8 Vector Processing [40 points]

Assume a vector processor that implements the following ISA:

Opcode	Operands	Latency (cycles)	Description
SET	V_{st}, #n	1	$V_{st} \leftarrow$ n (V_{st} = Vector Stride Register)
SET	V_{ln}, #n	1	$V_{ln} \leftarrow$ n (V_{ln} = Vector Length Register)
VLD	V_i, #A	100, pipelined	$V_i \leftarrow Mem[Address]$
VST	V_i, #A	100, pipelined	$Mem[Address] \leftarrow V_i$
VMUL	V_i, V_j, V_k	10, pipelined	$V_i \leftarrow V_j * V_k$
VADD	V_i, V_j, V_k	5, pipelined	$V_i \leftarrow V_j + V_k$
VDIV	V_i, V_j, V_k	20, pipelined	$V_i \leftarrow V_j / V_k$

Assume the following:

- The processor has an in-order pipeline.
- The size of a vector element is 4 bytes.
- V_{st} and V_{ln} are 10-bit registers.
- The processor *does not* support chaining between vector functional units.
- The main memory has N banks.
- Vector elements stored in consecutive memory addresses are interleaved between the memory banks. E.g., if a vector element at address A maps to bank B, a vector element at address $A + 4$ maps to bank $(B + 1)\%N$, where % is the modulo operator and N is the number of banks. N is *not necessarily* a power of two.
- The memory is byte addressable and the address space is represented using 32 bits.
- Vector elements are stored in memory in 4-byte-aligned manner.
- Each memory bank has a 4 KB row buffer.
- Each memory bank has a single read and a single write port so that a load and a store operation can be performed simultaneously.
- There are separate functional units for executing VLD and VST instructions.

(a) [5 points] What should the minimum value of N be to avoid stalls while executing a VLD or VST instruction, assuming a vector stride of 1? Explain.

100 banks (because the latency of VLD and VST instructions is 100 cycles)

(b) [8 points] What should the minimum value of N be to avoid stalls while executing a VLD or VST instruction, assuming a vector stride of 2? Explain.

101 banks.
Explanation. To avoid stalls, we need to ensure that consecutive vector elements access 100 different banks.
With a vector stride of 2, consecutive elements of a vector will map to every other bank. For example, if the first element maps to bank 0, the next element will map to bank 2, and so on.
With 100 banks, the 51st element of a vector will map to bank $100\%100 = 0$, conflicting with the first element of the vector.
Howevert, with 101 banks, the 51st element will map to bank 1, which was skipped by the previous vector elements.

Let's assume there are 102 elements in a vector and the first elements accesses bank 0. The 101 banks will be accessed in the following order:

$(0, 2, ..., 100, 102, 104, ..., 200, 202)\%101 = (0, 2, ..., 100, 1, 3, ..., 99, 0)$
We can see that none of the elements conflict in the DRAM banks. Note that, when the last vector elements accesses bank 0, the bank is already available for a new access because the 100 cycle latency of accessing the first element is overlapped by accessing the other 101 elements.

(c) [12 points] Assume:

- A machine that has a memory with as many banks as you found is part (a).
- The vector stride is set to 1.
- The value of the vector length is set to M (but we do *not* know M)

The machine executes the following program:

```
VLD V1 ← A
VLD V2 ← (A + 32768)
VADD V3 ← V1, V1
VMUL V4 ← V2, V3
VST (A + 32768*2) ← V4
```

The total number of cycles needed to complete the execution of the above program is 4306. What is M?

$M = 1000$

Explanation.

```
VLD  |-100-|--(M-1)--|
VLD                  |-100-|--(M-1)--|
VADD                  |-5-|--(M-1)--|
VMUL                                 |-10-|--(M-1)--|
VST                                                |-100-|--(M-1)--|
```

$(M + 100 - 1) + 100 + (M - 1) + 10 + (M - 1) + 100 + (M - 1)$
$= 306 + 4 * M = 4306 \rightarrow M = 1000$ elements

(d) [15 points] If we modify the vector processor to *support chaining*, how many cycles would be required to execute the same program in part (c)? Explain.

```
VLD  |--100--|--(VLEN-1)--|
VLD                     |---100---|---(VLEN-1)---|
VADD                    |-1-|-5-|---(VLEN-1)---| (this is delayed because the processor
                                                 executes the instructions in order)
VMUL                              |-10-|---(VLEN-1)---|
VST                                    |-100-|---(VLEN-1)---|

    100 + (VLEN-1) + 100 + 10 + 100 + (VLEN-1) = 310 + 2*1000 - 2 = 2308 cycles
```

9 GPUs and SIMD [45 points]

We define the *SIMD utilization* of a program that runs on a GPU as the fraction of SIMD lanes that are kept busy with *active threads* during the run of the program. As we saw in lecture and practice exercises, the SIMD utilization of a program is computed across the *complete run* of the program.

The following code segment is run on a GPU. Each thread executes **a single iteration** of the shown loop. Assume that the data values of the arrays A and B are already in vector registers, so there are no loads and stores in this program. (Hint: Notice that there are 3 instructions in each iteration.) A warp in the GPU consists of 32 threads, and there are 32 SIMD lanes in the GPU.

```
for (i = 0; i < 1025; i++) {
    if (A[i] < 33) {         // Instruction 1
        B[i] = A[i] << 1;    // Instruction 2
    }
    else {
        B[i] = A[i] >> 1;    // Instruction 3
    }
}
```

Please answer the following six questions.

(a) [2 points] How many warps does it take to execute this program?

33 warps.

Explanation:
The number of warps is calculated as:
$\#Warps = \lceil \frac{\#Total\_threads}{\#Warp\_size} \rceil$,

where
$\#Total\_threads = 1025 = 2^{10} + 1$ (i.e., one thread per loop iteration),

and
$\#Warp\_size = 32 = 2^5$ (given).

Thus, the number of warps needed to run this program is:
$\#Warps = \lceil \frac{2^{10}+1}{2^5} \rceil = 2^5 + 1 = 33.$

(b) [10 points] What is the *maximum* possible SIMD utilization of this program? (Hint: The warp scheduler does *not* issue instructions when *no* threads are active).

$\frac{1025}{1056}$.

Explanation:
Even though all active threads in a warp follow the same execution path, the last warp will only have one active thread.

(c) [5 points] Please describe what needs to be true about array `A` to reach the maximum possible SIMD utilization asked in part (b). (Please cover all cases in your answer.)

For every 32 consecutive elements of `A`, every element should be lower than 33 (`if`), or greater than or equal to 33 (`else`). (NOTE: The solution is correct if both cases are given.)

(d) [13 points] What is the *minimum* possible SIMD utilization of this program?

$\frac{1025}{1568}$.

Explanation:
`Instruction 1` is executed by every active thread ($\frac{1025}{1056}$ utilization).
Then, part of the threads in each warp executes `Instruction 2` and the other part executes `Instruction 3`. We consider that `Instruction 2` is executed by α threads in each warp (except the last warp), where $0 < \alpha \leq 32$, and `Instruction 3` is executed by the remaining $32 - \alpha$ threads. The only active thread in the last warp executes either `Instruction 2` or `Instruction 3`. The other instruction is not issued for this warp.

The minimum SIMD utilization sums to $\frac{1025+\alpha\times32+(32-\alpha)\times32+1}{1056+1024+1024+32} = \frac{1025}{1568}$.

(e) [5 points] Please describe what needs to be true about array `A` to reach the minimum possible SIMD utilization asked in part (d). (Please cover all cases in your answer.)

For every 32 consecutive elements of `A`, part of the elements should be lower than 33 (`if`), and the other part should be greater than or equal to 33 (`else`).

(f) [10 points] What is the SIMD utilization of this program if `A[i] = i`? Show your work.

$\frac{1025}{1072}$.

Explanation:
`Instruction 1` is executed by every active thread ($\frac{1025}{1056}$ utilization).
`Instruction 2` is executed by the first 33 threads, i.e., all threads in the first warp and one thread in the second warp.
`Instruction 3` is executed by the remaining active threads.

The SIMD utilization sums to $\frac{1025+32+1+31+960+1}{1056+32+32+32+960+32} = \frac{2050}{2144} = \frac{1025}{1072}$.

10 Memory Hierarchy [40 points]

An enterprising computer architect is building a new machine for high-frequency stock trading and needs to choose a CPU. She will need to optimize her setup for *memory access latency* in order to gain a competitive edge in the market. She is considering two different prototype enthusiast CPUs that advertise high memory performance:

(A) Dragonfire-980 Hyper-Z

(B) Peregrine G-Class XTreme

She needs to characterize these CPUs to select the best one, and she knows from Prof. Mutlu's course that she is capable of reverse-engineering everything she needs to know. Unfortunately, these CPUs are not yet publicly available, and their exact specifications are unavailable. Luckily, important documents were recently leaked, claiming that the two CPUs have:

- Exactly 1 high-performance core
- LRU replacement policies (for any set-associative caches)
- Inclusive caching (i.e., data in a given cache level is present upward throughout the memory hierarchy. For example, if a cache line is present in L1, the cache line is also present in L2 and L3 if available.)
- Constant-latency memory structures (i.e., an access to any part of a given memory structure takes the same amount of time)
- Cache line, size, and associativity are all size aligned to powers of two

Being an ingenious engineer, she devises the following simple application in order to extract all of the information she needs to know. The application uses a high-resolution timer to measure the amount of time it takes to read data from memory with a specific pattern parameterized by *STRIDE* and *MAX_ADDRESS*:

```
start_timer()
repeat N times:
    memory_address <- random_data()
    READ[(memory_address * STRIDE) % MAX_ADDRESS]
end_timer()
```

Assume 1) this code runs for a long time, so all memory structures are fully warmed up, i.e., repeatedly accessed data is already cached, and 2) N is large enough such that the timer captures ***only*** *steady-state information.*

By sweeping *STRIDE* and *MAX_ADDRESS*, the computer architect can glean information about the various memory structures in each CPU.

She produces Figure 1 for CPU A and Figure 2 for CPU B.

Your task: Using the data from the graphs, reverse-engineer the following system parameters. If the parameter does *not make sense* (e.g., L3 cache in a 2-cache system), mark the box with an "X". If the graphs provide *insufficient information* to ascertain a desired parameter, simply mark it as "N/A".

NOTE 1 TO SOLUTION READER:
This analysis provides insufficient information to determine the line size of the cache(s). This is because we are always 'striding' in power-of-two values starting at address 0. This means that either our access pattern entirely fits within the cache (in which case we observe constant latency since the cache is already warmed up), or the access pattern is striding using values larger than the line size, so we never see two accesses to the same cache line.

NOTE 2 TO SOLUTION READER:
This problem is not actually that hard.
The way to think about these plots is that each point is an access pattern. The easiest points to understand are those that result in an access pattern of {0, 0, 0, 0, ...} and randomly from {0, A}, where A is your stride. Just by looking at those you should be able to determine pretty much everything.
The access latencies and sizes are trivial to read off if you understand what the test code is trying to do. The associativities are nuanced, but you can tell from the aforementioned access patterns by simulating carefully.
If you want to go all-in, you can compute probabilities: if I access {0, A} then 50% of the time I'll hit and 50% miss. It's easy to get the cache latencies, so I can just match points from there on :)

(a) [15 points] Fill in the blanks for Dragonfire-980 Hyper-Z.

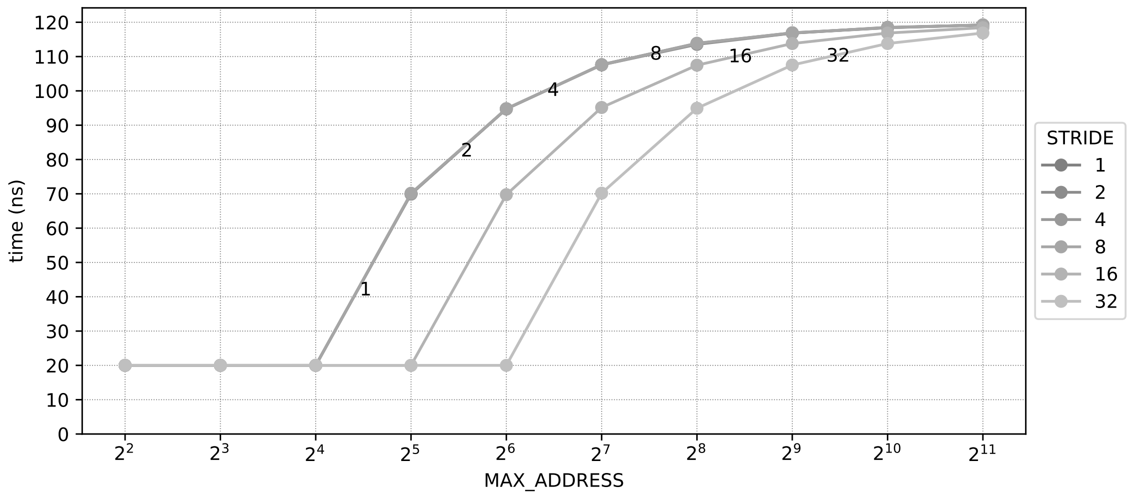


Figure 1: Execution time of the test code on CPU A for various values of *STRIDE* and *MAX_ADDRESS*. *STRIDE* values are labeled on curves themselves for clarity. Note that the curves for strides 1, 2, 4, and 8 overlap in the figure.

Table 1: Fill in the following table for CPU A (Dragonfire-980 Hyper-Z)

System Parameter	**CPU A: Dragonfire-980 Hyper-Z**			
	L1	L2	L3	DRAM
Cache Line Size (B)	N/A	N/A	N/A	N/A OR X
Cache Associativity	2	X	X	X
Total Cache Size (B)	16	X	X	X
Access Latency from (ns) [1]	20	X	X	100

[1] DRAM access latency means the latency of fetching the data from DRAM to L3, *not* the latency of bringing the data from the DRAM all the way down to the CPU. Similarly, L3 access latency means the latency of fetching the data from L3 to L2. L1 access latency is the latency to bring the data to the CPU from the L1 cache.

(b) [25 points] Fill in the blanks for Peregrine G-Class XTreme.

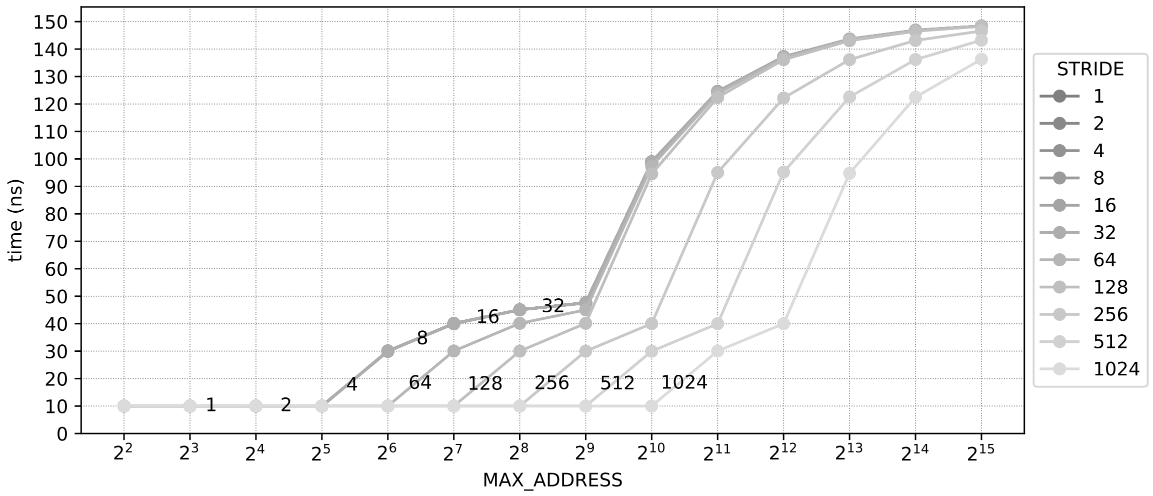


Figure 2: Execution time of the test code on CPU B for various values of *STRIDE* and *MAX_ADDRESS*. *STRIDE* values are labeled on curves themselves for clarity. Note that the curves for strides 1, 2, 4, 8, 16, and 32 overlap in the figure.

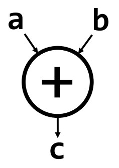
Table 2: Fill in the following table for CPU B (Peregrine G-Class XTreme)

System Parameter	**CPU B: Peregrine G-Class XTreme**			
	L1	L2	L3	DRAM
Cache Line Size (B)	N/A	N/A	N/A	N/A OR X
Cache Associativity	1	4	X	X
Total Cache Size (B)	32	512	X	X
Access Latency (ns) [1]	10	40	X	100

[1] DRAM access latency means the latency of fetching the data from DRAM to L3, *not* the latency of bringing the data from the DRAM all the way down to the CPU. Similarly, L3 access latency means the latency of fetching the data from L3 to L2. L1 access latency is the latency to bring the data to the CPU from the L1 cache.

11 Dataflow Meets Logic [35 points]

We often use the "addition node":



to represent the addition of two input tokens. If we think of the tokens as binary numbers, we can model a simple logic circuit using dataflow graphs.[1] Note that a token can be used as an input to only *one* node. If the same value is needed by more than one node, it first should be replicated using one or more copy nodes, and then each copied token can be supplied to one node only.

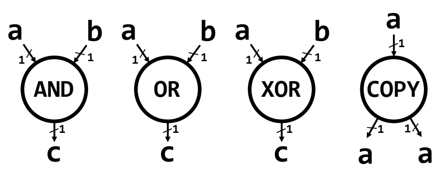
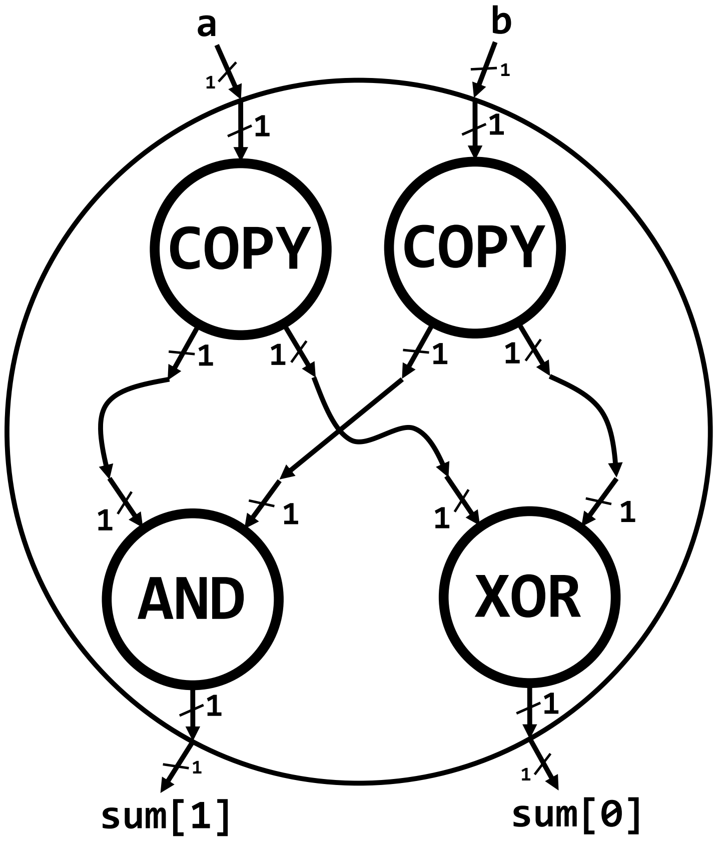


Figure 3: Dataflow nodes of basic bitwise operations allowed in Part (a).

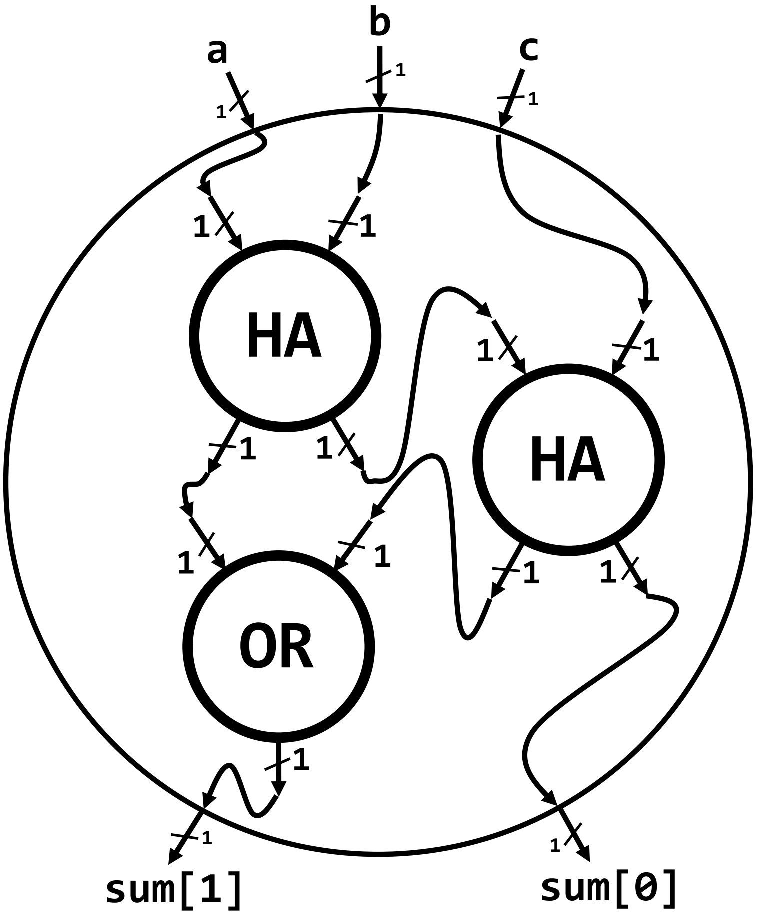
(a) [5 points] Implement the single-bit binary addition of two "1-bit" input tokens a and b as a dataflow graph using *only* 2-input {AND, OR, XOR} nodes and COPY nodes if necessary (illustrated in Figure 3). Fill in the internal implementation below, where inputs and outputs (labeled with their corresponding bit-widths) have been provided:



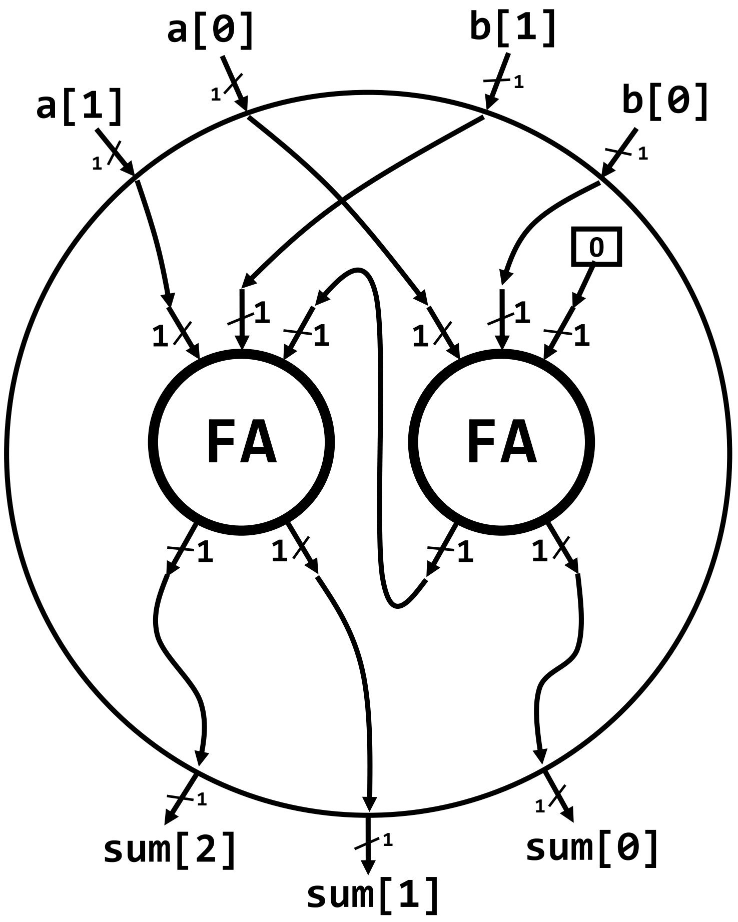
[1]Note: this is not an accurate electrical model of a circuit. Instead, the dataflow analogy is best thought of in terms of the desired flow of *information* rather than physical phenomena.

(b) [5 points] You may recognize the node we designed in part (a) as a model for a so-called "half-adder (HA)", which is not very useful by itself since it is only useful for adding 1-bit input tokens. In order to extend this design to perform binary addition of 2-bit input tokens `a[1:0]` and `b[1:0]`, the `sum[1]` token from half-adding `a[0]` and `b[0]` will have to act as an input token for *another* half-adder node used for adding `a[1]` and `b[1]`. This results in a 3-input adder called a "full-adder (FA)".

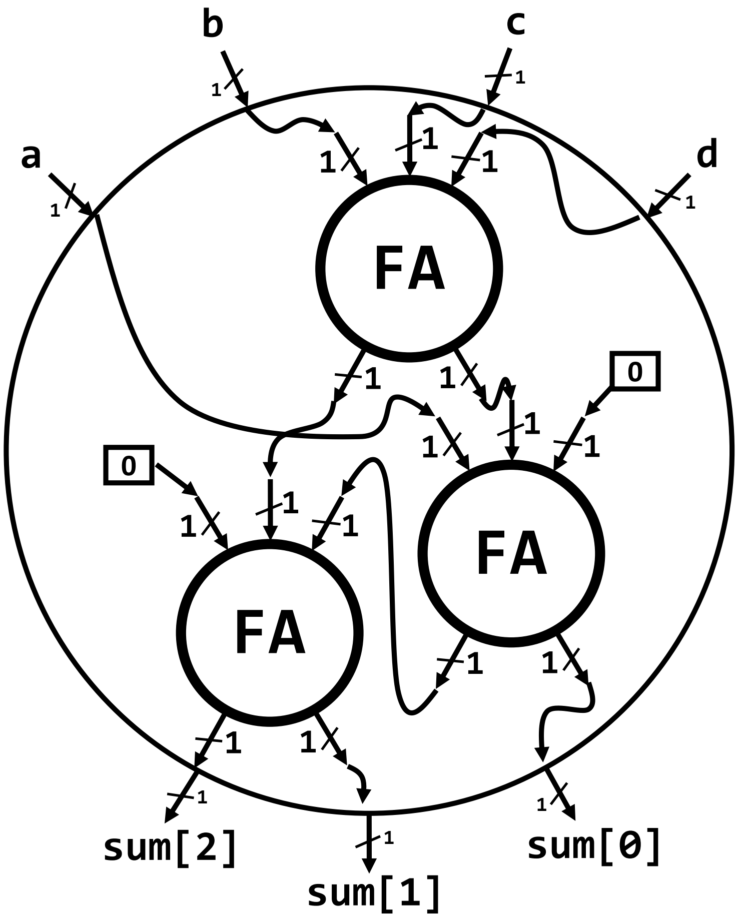
Fortunately, we can implement a full-adder (FA) using half-adders (HA) (i.e., the node we designed in part (a). Implement the full-adder using a *minimum* number of half-adders and *at most* 1 additional 2-input {AND, OR, XOR} node.



(c) [5 points] The full-adder (FA) is a versatile design that can be used to implement n-bit addition. Show how we might use it to implement 2-bit binary addition of two input tokens `a[1:0]` and `b[1:0]`. Use only a *minimum* number of full-adders (i.e., the dataflow node you designed in Part 2). *Hint: you may use constant input tokens if necessary.*



(d) [5 points] Interestingly, the full-adder can also be used to add four 1-bit input tokens. This is a natural extension of the full-adder in the same way we extended the half-adder to create the full-adder itself (in part (b)). Implement the 4-input node below using only a *minimum* number of full-adders (FA) (i.e., the dataflow node you designed in part (b)). *Hint: you may use constant input tokens if necessary.*



(e) [15 points] As it turns out, any $n \geq 3$ 1-bit input binary adders can be implemented purely using full-adders. Fill in the table below for the *minimum* number of required full adders to implement an n-input 1-bit adder.

n	# **required full-adders**
3	1
4	3
5	3
6	4
7	4
8	7

12 BONUS: Branch Prediction [40 points]

Assume a processor that implements an ISA with eight registers (R0-R7). In this ISA, the main memory is byte-addressable and each word contains 4 bytes. The processor employs a branch predictor. The ISA implements the instructions given in the following table:

Instructions	**Description**
la R_i, Address	load the *Address* into R_i
move R_i, R_j	$R_i \leftarrow R_j$
move R_i, (R_j)	$R_i \leftarrow$ Memory[R_j]
move (R_i), R_j	Memory[R_i] $\leftarrow R_j$
li R_i, Imm	$R_i \leftarrow$ Imm
add R_i, R_j, R_k	$R_i \leftarrow R_j + R_k$
addi R_i, R_j, Imm	$R_i \leftarrow R_j +$ Imm
cmp R_i, R_j	Compare: Set sign flag, if $R_i < R_j$; set zero flag, if $R_i = R_j$
cmp R_i, (R_j)	Compare: Set sign flag, if $R_i <$ Memory[R_j]; set zero flag, if $R_i =$ Memory[R_j]
cmpi R_i, Imm	Compare: Set sign flag, if $R_i <$ Imm; set zero flag, if $R_i =$ Imm.
jg label	Jump to the target address if **both** of sign and zero flags are zero.
jnz label	Jump to the target address if zero flag is zero.
halt	Stop executing instructions.

The processor executes the following program. Answer the questions below related to the accuracy of the branch predictors that the processor can potentially implement.

```
       la R0, Array
       move R6, R0
       li R1, 4
       move R5, R1
       move R7, R1
       move R2, R0
       addi R2, R2, 4
Loop:
       move R3, (R2)
       cmp R3, (R0)
       jg Next_Iteration
       move R4, (R0)
       move (R0), R3
       move (R2), R4
Next_Iteration:
       addi R0, R0, 4
       addi R2, R2, 4
       addi R1, R1, -1
       cmpi R1, 0
       jnz Loop
       move R1, R7
       addi R5, R5, -1
       move R0, R6
       move R2, R0
       addi R2, R2, 4
       cmpi R5, 0
       jnz Loop
       halt
.data
Array: word 5, 20, 1, -5, 34
```

(a) [15 points] What would be the prediction accuracy using a global one-bit-history (last-time) branch predictor shared between *all* the branches? The initial state of the predictor is "taken".

Answer: 19/36.

Note that initial values of both R_1 and R_5 are 4; and they change only before the branches in lines 20 and 27 respectively. Both branches follow the pattern of T-T-T-NT, which creates a nested loop.

At each iteration of the internal loop, adjacent elements (pointed by R_0 and R_2) are swapped, if $Memory[R_0] \leq Memory[R_2]$. Then, both R_0 and R_4 are incremented by 4. So they point to the next element in the next iteration.

Therefore, the code sorts the elements in *Array* in increasing order.

Table below shows the behavior of each branch through the code. Here T means that the corresponding branch is taken at specified turn, whereas N indicates that it is not taken.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Line11	T		N		N		T			N		N		T		T		
Line20		T		T		T		N			T		T		T		N	
Line27									T									T

	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
Line11	N		T		T		T			T		T		T		T		
Line20		T		T		T		N			T		T		T		N	
Line27									T									N

One-bit-history branch predictor suggests that the next branch's behavior will be the same with the last one. Table below shows the predictor states, hits, and misses through the execution.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Predictor State	T	T	T	N	T	N	T	T	N	T	N	T	N	T	T
Branch Behavior	T	T	N	T	N	T	T	N	T	N	T	N	T	T	T
Hit/Miss	H	H	M	M	M	M	H	M	M	M	M	M	M	H	H

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Predictor State	T	T	N	T	N	T	T	T	T	T	T	N	T	T	T
Branch Behavior	T	N	T	N	T	T	T	T	T	T	N	T	T	T	T
Hit/Miss	H	M	M	M	M	H	H	H	H	H	M	M	H	H	H

	31	32	33	34	35	36
Predictor State	T	T	T	T	T	N
Branch Behavior	T	T	T	T	N	N
Hit/Miss	H	H	H	H	M	H

(b) [15 points] What would be the prediction accuracy using a global two-bit-history (two-bit counter) branch predictor shared between *all* the branches? Assume that the initial state of the two-bit counter is "weakly taken". The "weakly taken" state transitions to the "weakly not-taken" state on misprediction. Similarly, the "weakly not-taken" state transitions to the "weakly taken" state on misprediction. A correct prediction in one of the "weak" states transitions the state to the corresponding "strong" state.

Answer: 26/36.

Explanation:
Table below shows the predictor states, hits, and misses through the code. Used abbreviations are as follows: ST: Strongly Taken, WT: Weakly Taken, WN: Weakly Not-taken, SN: Strongly Not-taken.

Branch behavior is the same with question (a), since both of them are shared predictors.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Predictor State	WT	ST	ST	WT	ST	WT	ST	ST	WT	ST	WT	ST	WT	ST
Branch Behavior	T	T	N	T	N	T	T	N	T	N	T	N	T	T
Hit/Miss	H	H	M	H	M	H	H	M	H	M	H	M	H	H

	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Predictor State	ST	ST	ST	WT	ST	WT	ST	ST	ST	ST	ST	ST	WT	ST
Branch Behavior	T	T	N	T	N	T	T	T	T	T	T	N	T	T
Hit/Miss	H	H	M	H	M	H	H	H	H	H	H	M	H	H

	29	30	31	32	33	34	35	36
Predictor State	ST	ST	ST	ST	ST	ST	ST	WT
Branch Behavior	T	T	T	T	T	T	N	N
Hit/Miss	H	H	H	H	H	H	M	M

(c) [10 points] What would be the prediction accuracy using a local two-bit-history (two-bit counter) branch predictor that is separate for *each* branch? The initial state is "weakly taken" and the state transitions are the same as in part (b).

Answer:

- L11: 8/16
- L20: 12/16
- L27: 3/4
- All Branches: 23/36

Explanation: Private predictors update their states only based on the behaviors of corresponding branches.

	1	2	3	4	5	6	7	8	9	10	11	12
L11 Predictor State	WT		ST		WT		WN			WT		WN
L11 Branch Behavior	T		N		N		T			N		N
L11 Hit/Miss	H		M		M		M			M		H
L20 Predictor State		WT		ST		ST		ST			WT	
L20 Branch Behavior		T		T		T		N			T	
L20 Hit/Miss		H		H		H		M			H	
L27 Predictor State									WT			
L27 Branch Behavior									T			
L27 Hit/Miss									H			

	13	14	15	16	17	18	19	20	21	22	23	24
L11 Predictor State		SN		WN			WT		WN		WT	
L11 Branch Behavior		T		T			N		T		T	
L11 Hit/Miss		M		M			M		M		H	
L20 Predictor State	ST		ST		ST			WT		ST		ST
L20 Branch Behavior	T		T		N			T		T		T
L20 Hit/Miss	H		H		M			H		H		H
L27 Predictor State						ST						
L27 Branch Behavior						T						
L27 Hit/Miss						H						

	25	26	27	28	29	30	31	32	33	34	35	36
L11 Predictor State	ST			ST		ST		ST		ST		
L11 Branch Behavior	T			T		T		T		T		
L11 Hit/Miss	H			H		H		H		H		
L20 Predictor State		ST			WT		ST		ST		ST	
L20 Branch Behavior		N			T		T		T		N	
L20 Hit/Miss		M			H		H		H		M	
L27 Predictor State			ST									ST
L27 Branch Behavior			T									N
L27 Hit/Miss			H									M