1   Branch Prediction I

Assume the following piece of code that iterates through a large array populated with **completely (i.e., truly) random** positive integers. The code has four branches (labeled B1, B2, B3, and B4). When we say that a branch is *taken*, we mean that the code *inside* the curly brackets is executed.

```
for (int i=0; i<N; i++) { /* B1 */
  val = array[i];        /* TAKEN PATH for B1 */
  if (val % 2 == 0) {    /* B2 */
    sum += val;        /* TAKEN PATH for B2 */
  }
  if (val % 3 == 0) {    /* B3 */
    sum += val;        /* TAKEN PATH for B3 */
  }
  if (val % 6 == 0) {    /* B4 */
    sum += val;        /* TAKEN PATH for B4 */
  }
}
```

(a) Of the four branches, list all those that exhibit *local correlation*, if any.

> Only B1.
>
> B2, B3, B4 are not locally correlated. Just like consecutive outcomes of a die, an element being a multiple of N (N is 2, 3, and 6, respectively for B2, B3, and B4) has no bearing on whether the next element is also a multiple of N.

(b) Which of the four branches are *globally correlated*, if any? Explain in less than 20 words.

> B4 is correlated with B2 and B3. This means that if B2 and B3 are taken then B4 should always be taken (as 6 is a common multiple of 2 and 3).

Now assume that the above piece of code is running on a processor that has a global branch predictor. The global branch predictor has the following characteristics.

- Global history register (GHR): 2 bits.

- Pattern history table (PHT): 4 entries.

- Pattern history table entry (PHTE): 11-bit signed saturating counter (possible values: -1024–1023)

- Before the code is run, all PHTEs are initially set to 0.

- As the code is being run, a PHTE is incremented (by one) whenever a branch that corresponds to that PHTE is taken, whereas a PHTE is decremented (by one) whenever a branch that corresponds to that PHTE is not taken.
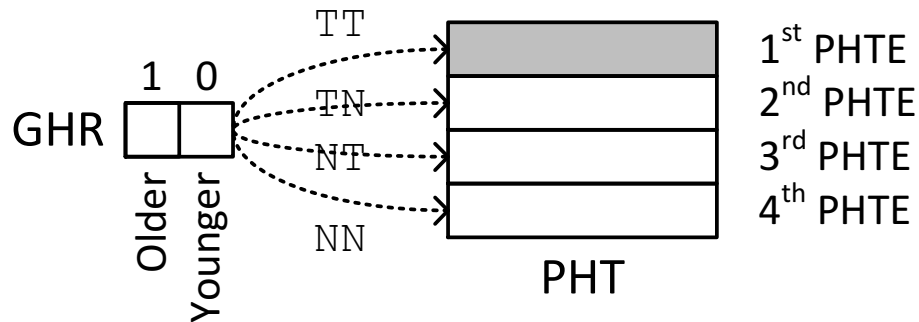
(c) After 120 iterations of the loop, calculate the **expected** value for only the first PHTE and fill it in the shaded box below. (Please write it as a base-10 value, rounded to the nearest one's digit.)

Hint. For a given iteration of the loop, first consider, what is the probability that both B1 and B2 are taken? Given that they are, what is the probability that B3 will increment or decrement the PHTE? Then consider...

Show your work.

Without loss of generality, let's take a look at the numbers from 1 through 6..

For a **single** iteration of the loop, the PHTE has four chances of being incremented/decremented, once at each branch.

• B3's contribution to PHTE. The probability that both B1 and B2 are taken is denoted as P(B1_T && B2_T), which is equal to P(B1_T)*P(B2_T) = 1*1/2 = 1/2 (out of the six numbers, only 2, 4, and 6 are divisible by 2). Given that B1 and B2 are taken, the probability that B3 is also taken, is equal to Q = 1/3 (out of 2, 4, 6 only 6 is divisible by 3). Therefore, the PHTE will be incremented with probability 1/2*1/3 = 1/6 and decremented with probability 1/2*(1-1/3) = 1/3 (out of 2, 4, 6 2 and 4 are not divisible by 3). The net contribution of B3 to PHTE is 1/6-1/3 = -1/6.

• B4's contribution to PHTE. P(B2_T && B3_T) = 1/6. P(B4_T | B2_T && B3_T) = R = 1. B4's net contribution is 1/6*1 = 1/6.

• B1's contribution to PHTE. P(B3_T && B4_T) = 1/6. P(B1_T | B3_T && B4_T) = 1. B1's net contribution is 1/6*1 = 1/6.

• B2's contribution to PHTE. P(B4_T && B1_T) = 1/6*1 = 1/6. P(B2_T | B4_T && B1_T) = 1/2. This is because B4 is the last branch in the for loop and thus it has no effect on B2 from the next iteration. So B2 is taken whenever we have one of the values 2, 4, 6 out of the six values.B2's net contribution is 1/6*1/2 - 1/6*1/2 = 0.

For a single iteration, the net contribution to the PHTE, summed across all the four branches, is equal to 1/6. Since there are 120 iterations, the expected PHTE value is equal to 1/6*120=**20**.

2 Branch Prediction II

Assume a machine with a two-bit global history register (GHR) shared by all branches, which starts with Not Taken, Not Taken (2'b00). Each pattern history table entry (PHTE) contains a 2-bit saturating counter. The saturating counter values are as follows:

- 00 - Strongly Not Taken
- 01 - Weakly Not Taken
- 10 - Weakly Taken
- 11 - Strongly Taken

Assume the following piece of code runs on this machine. The code has two branches (labeled B1 and B2). When we say that a branch is taken, we mean that the code inside the curly brackets is executed. For the following questions, assume that this is the only block of code that will ever be run, and the loop-condition branch (B1) is resolved first in the iteration before the if-condition branch (B2).

```
for (int i = 0; i < 1000000; i++) { /* B1 */
                                    /* TAKEN PATH for B1 */
    if (i % 3 == 0) {              /* B2 */
        j[i] = k[i] -1;            /* TAKEN PATH for B2 */
    }
}
```

(a) Is it possible to observe that the branch predictor mispredicts 100% of the time in the first 5 iterations of the loop? If yes, fill in the table below with all possible initial values each entry can take. We represent Not Taken with N, and Taken with T.

Table 1: PHT

PHT Entry	Value
TT	01
TN	00
NT	01
NN	00 or 01

Show your work here.

Yes, it is possible.
The pattern after 5 iterations: TTTNTNTTTN.
In order to be more clear, we add indices to each branch outcome in the pattern above, to represent their positions in the pattern: T_1 T_2 T_3 N_4 T_5 N_6 T_7 T_8 T_9 N_{10}

- For GHR=NN, the only observed branch is T_1. Therefore, the PHTE for NN has to be either 00 or 01 so that the branch predictor mispredicts the taken branch.
- For GHR=TT, the observed branches are T_3 N_4 T_9 N_{10}. The PHTE for TT has to be initialized to 01 in order to cause the predictor to always mispredict. This way, each N and T moves the saturating counter to their respective direction. This will cause misprediction for the next branch which is always in the opposite direction.
- For GHR=TN, the observed branches are T_5 T_7. Thus, the initial PHTE value for TN has to be 00 to mispredict both taken branches.
- For GHR=NT, the observed branches are T_2 N_6 T_8. Similar to the TT entry, NT's PHTE has to be initialized 01.

(b) At steady-state, we observe the following pattern which repeats over time: TTTNTN, with T representing Taken, and N representing Not Taken. When GHR pattern equals to NT or TT, the predictor will observe that the branch outcome will be either T or N. Therefore, no matter what the initial values for these two entries are in the pattern history table (PHT), only one of the branches can be predicted correctly. Thus prediction accuracy will never reach 100%. Explain how using local history registers instead of the global history register will help bring the prediction accuracy up to 100% during the steady state, by showing what will each PHTE saturate to.

For the outer loop, we will keep observing all Ts, and the counters will be set to 2'b11 for TT and lead to 100% accuracy for this branch.
The second branch will keep observing this repeated pattern: TNN. So entry TN will be saturated to 2'b00, entry NN will saturate to 2'b11, and entry NT will saturate to 2'b00.

3 Branch Prediction III

A processor implements an *in-order* pipeline with multiple stages. Each stage completes in a single cycle. The pipeline stalls upon fetching a conditional branch instruction and resumes execution once the condition of the branch is evaluated. There is no other case in which the pipeline stalls.

3.1 Part I: Microbenchmarking

You create a microbenchmark as follows to explore the pipeline characteristics:

```
LOOP1:
    SUB R1, R1, #1  // R1 = R1 - 1
    BGT R1, LOOP1   // Branch to LOOP1 if R1 > 0

LOOP2:
    B   LOOP2        // Branch to LOOP2
                     // Repeats until program is killed
```

The microbenchmark takes one input value `R1` and runs until it is killed (e.g., via an external interrupt).

You carefully run the microbenchmark using three different input values as summarized in Table 2. You terminate the microbenchmark using an external interrupt such that each run is guaranteed to execute the same number of *dynamic instructions*. Unfortunately, your testing infrastructure does *not* give you the actual number of instructions executed.

Initial R1 Value	Number of Cycles Taken
4	51
8	63
16	87

Table 2: Microbenchmark results.

Using this information, you need to determine the following three experiment characteristics. *Clearly show all work to receive full points!*

1. How many dynamic instructions are executed?

2. How many stages are in the pipeline?

3. For how many cycles does a conditional branch instruction cause a stall?

---

1. 3 cycles
2. It is only possible to know that $P + I = 40$.

Explanation: We have a system of equations in the variables:

- C is the total number of cycles taken
- P is the total number of pipeline stages
- I is the total number of dynamic instructions executed
- B is the number of conditional branch instructions executed
- D is the number of cycles stalled for each conditional branch

The total number of cycles can be expressed as $C = P + I - 1 + B * D$.
Table 2 gives us B and C, which results in a system of three equations with three unknowns:

- $51 = P + I - 1 + 4 * D$
- $63 = P + I - 1 + 8 * D$
- $87 = P + I - 1 + 16 * D$

Solving this system, we obtain $D = 3$ and $P + I = 40$.

3.2 Part II: Performance Enhancement

To improve performance, the architects add a *mystery* branch prediction mechanism. They keep the rest of the design exactly the same as before. You re-run the microbenchmark for the same number of total dynamic instructions with the new design, and you find that with $R1 = 4$, the microbenchmark executes in 48 cycles.

Based on this given information, determine which of the following branch prediction mechanisms could be the *mystery* branch predictor implemented in the new version of the processor. For each branch prediction mechanism below, you should circle the configuration parameters that makes it match the performance of the mystery branch predictor.

(a) **Static Branch Predictor**

Could this be the mystery branch predictor: YES   NO

If YES, for which configuration below is the answer YES? Pick an option for each configuration parameter.

   I) Static Prediction Direction

      Always taken   Always not taken

Explain:

YES, if the static prediction direction is `always not taken`.
Explanation: The execution time corresponds to 3 mispredictions and 1 correct prediction. The correct prediction occurs when the branch condition evaluates to TRUE and execution falls through to the following instruction (i.e., NOT TAKEN).

(b) **Last Time Branch Predictor**

Could this be the mystery branch predictor?

YES   NO

If YES, for which configuration is the answer *YES*? Pick an option for each configuration parameter.

   I) Initial Prediction Direction

      Taken   Not taken

   II) Local for each branch instruction (PC-based) or global (shared among all branches) history?

      Local   Global

Explain:

NO.
Explanation: The last-time predictor will make a correct prediction at least three times, which means that it cannot be the mystery predictor.

(c) **Backward taken, Forward not taken (BTFN)**

Could this be the mystery branch predictor?

YES   NO

Explain:

> *NO.*
>
> **Explanation:** The BTFN predictor makes exactly one *mis*prediction, which is the opposite of what the mystery predictor achieves.

(d) **Forward taken, Backwards not taken (FTBN)**

Could this be the mystery branch predictor?

YES   NO

Explain:

> *YES.*
>
> **Explanation:** The FTBN predictor makes exactly one correct prediction, which is what we observe from the microbenchmark.

(e) **Two-bit Counter Based Prediction** (using saturating arithmetic)

Could this be the mystery branch predictor?

YES   NO

If YES, for which configuration is the answer *YES*? Pick an option for each configuration parameter.

   I) Initial Prediction Direction

   00 (Strongly not taken)   01 (Weakly not taken)
   10 (Weakly taken)   11 (Strongly taken)

   II) Local for each branch instruction (i.e., PC-based, without any interference between different
   branches) or global (i.e., a single counter shared among all branches) history?

   Local   Global

Explain:

> *YES*, using either *local* or *global* history registers with an initial value of *00*.
> **Explanation:** There is only one branch, so we cannot differentiate local vs. global registers. Either way, only the *00* configuration results in 3 mispredictions and 1 correct prediction.

4 Branch Prediction IV

Consider the following high level language code segment:

```
int array[1000] = { /* random values */ };
int sum1 = 0, sum2 = 0, sum3 = 0, sum4 = 0;

for (i = 0; i < 1000; i ++)      // Branch 1: Loop Branch
{
    // Branch 1: Taken
    if (i % 2 == 0)              // Branch 2: If Condition 1
        // Branch 2: Taken
        if (i % 3 == 0)         // Branch 3: If Condition 2
            sum1 += array[i];   // Branch 3: Taken
        else
            sum2 += array[i];
    else
        if (i % 4 == 0)         // Branch 4: If Condition 3
            sum3 += array[i];   // Branch 4: Taken
        else
            sum4 += array[i];
}
```

(a) What is the prediction accuracy for each of the four branches using a per-branch last-time predictor (assume that every per-branch counter starts at "not-taken")? Please show all of your work.

Branch 1:

999/1001. The branch is mispredicted the first and last time it's executed.

Branch 2:

$0/1000 \times 100 = 0\%$. The branch changes direction every time it's executed.

Branch 3:

33.2%. The pattern repeats Miss, Miss, Hit

Branch 4:

100%. The branch is never taken.

(b) What is the prediction accuracy for each of the four branches when a per-branch 2-bit saturating counter-based predictor is used (assume that every per-branch counter starts at "strongly not-taken")? Please show all of your work.

Branch 1:

998/1001. The branch is mispredicted the first two times and last time it's executed.

Branch 2:

$500/1000 \times 100 = 50\%$. The counter changes between "strongly not-taken" to "weakly not-taken" every iteration. Branch is taken every other time.

Branch 3:

> 66.6%. The pattern repeats Miss, Hit, Hit

Branch 4:

> 100%. The branch is never taken.

(c) What is the prediction accuracy for both Branch 2 and Branch 3, when the counter starts at (i) "weakly not-taken" and (ii) "weakly taken"?

Branch 2 (i):

> (i) 0%. The counter alternates between "weakly not-taken" and "weakly taken" missing on each

Branch 2 (ii):

> (ii) 50%. The counter alternates between "weakly taken" and "strongly taken" hitting every other

Branch 3 (i):

(i) 66.8%. Miss, Hit, Hit repeats

Branch 3 (ii):

(ii) 66.2%. After the predictor settles to "strongly not-taken", the pattern repeats Miss, Hit, Hit

5 VLIW I

Explain the motivation for VLIW in one sentence.

Enable multiple instruction issue with simple hardware. Independent instructions can be statically scheduled into a single VLIW instruction that can fed into multiple functional units concurrently

You are the human compiler for a VLIW machine whose specifications are as follows:

- There are 3 **fully** pipelined functional units (ALU, MU and FPU).

- Integer Arithmetic Logic Unit (ALU) has a 1-cycle latency.

- Memory Unit (MU) has a 2-cycle latency.

- Floating Point Unit (FPU) has a 3-cycle latency, and can perform either FADD or FMUL (floating point add / floating point multiply) on **floating point registers**.

- This machine has **only** 4 integer registers (r1 .. r4) and 4 floating point registers (f1 .. f4)

- The machine does not implement hardware interlocking or data forwarding.

(a) For the given assembly code on the next page, fill **Table 3** (on the next page) with the appropriate VLIW instructions for only one iteration of the loop (The C code is also provided for your reference). Provide the VLIW instructions that lead to the **best** performance. Use the minimum number of VLIW instructions. Table 3 should **only** contain instructions provided in the assembly example. For all the instruction tables, show the NOP instructions you may need to insert. Note that BNE is executed in the **ALU**.

The base addresses for A, B, C are stored in r1, r2, r3 respectively. The address of the last element in the array C[N-1] is stored in r4, where N is an integer multiplier of 10! (read: 10 factorial).

C Code

```
float A[N];
float C[N];
int   B[N];
...  // code to initialize A and B
for (int i=0; i<N; i++)
    C[i] = A[i] * A[i] + B[i];
```

Assembly Code

```
loop: LD   f1, 0 (r1)
      LD   f2, 0 (r2)
      FMUL f1, f1, f1
      FADD f1, f1, f2
      ADDI r3, r3, 4
      ST   f1, -4, (r3)
      ADDI r1, r1, 4
      ADDI r2, r2, 4
      BNE  r3, r4, loop
```

VLIW Instruction	**ALU**	**MU**	**FPU**
1	ADDI r1, r1, 4	LD f1, 0(r1)	NOP
2	ADDI r2,r2, 4	LD f2, 0(r2)	NOP
3	NOP	NOP	FMUL f1, f1, f1
4	NOP	NOP	NOP
5	NOP	NOP	NOP
6	NOP	NOP	FADD f1, f1, f2
7	NOP	NOP	NOP
8	ADDI r3, r3, 4	NOP	NOP
9	BNE r3, r4, loop	ST f1, -4(r3)	NOP
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			

Table 3

What is the performance in Ops/VLIW instruction (Operations/VLIW instruction) for this design? An operation here refers to an instruction (in the Assembly Code), excluding NOPs.

1

(b) Assume now we decide to unroll the loop once. Fill **Table 4** with the new VLIW instructions. You should optimize for latency first, then instruction count. **You can choose to use different offsets, immediates and registers, but you may not use any new instructions**.

VLIW Instruction	**ALU**	**MU**	**FPU**
1	NOP	LD f1, 0(r1)	NOP
2	ADDI r1,r1, 8	LD f3, 4(r1)	NOP
3	NOP	LD f2, 0(r2)	FMUL f1, f1, f1
4	ADDI r2, r2, 8	LD f4, 4(r2)	FMUL f3, f3, f3
5	NOP	NOP	NOP
6	NOP	NOP	FADD f1, f1, f2
7	NOP	NOP	FADD f3, f3, f4
8	NOP	NOP	NOP
9	ADDI r3, r3, 8	ST f1, 0(r3)	NOP
10	BNE r3, r4, loop	ST f3, -4(r3)	NOP
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			

Table 4

What is the performance in Ops/VLIW instruction for this design?

14/10

(c) Assume now we have **unlimited registers** and the loop is fully optimized (unrolled to the best performance possible). What is the performance in Ops/cycle for this design? Show your work and explain **clearly** how you arrived at your answer. You are not required to draw any tables, but you may choose to do so to aid your explanation. (Hint: trace the dependent instructions)

29/15. Notice that we can add 3 MU ops (2 LDs and 1 ST) and 2 FPU ops per unroll, while the ALU ops remain constant at 4. If you trace the table carefully, you will observe that the MU instruction stream will have 1 op/cycle by the time we unroll the loop five times. At this point, we have 4 + 15 + 10 = 29 instructions over 15 cycles. Any further unrolling will result in a smaller ops/cycle since the MU instruction stream is already saturated.

6 VLIW II

You are using a tool that transforms machine code that is written for the MIPS ISA to code in a VLIW ISA. The VLIW ISA is identical to MIPS except that multiple instructions can be grouped together into one VLIW instruction. Up to N MIPS instructions can be grouped together (N is the machine width, which depends on the particular machine). The transformation tool can reorder MIPS instructions to fill VLIW instructions, as long as loads and stores are not reordered relative to each other (however, independent loads and stores can be placed in the same VLIW instruction).

You give the tool the following MIPS program (we have numbered the instructions for reference below):

```
(01) lw   $t0 ← 0($a0)
(02) lw   $t2 ← 8($a0)
(03) lw   $t1 ← 4($a0)
(04) add  $t6 ← $t0, $t1
(05) lw   $t3 ← 12($a0)
(06) sub  $t7 ← $t1, $t2
(07) lw   $t4 ← 16($a0)
(08) lw   $t5 ← 20($a0)
(09) srlv $s2 ← $t6, $t7
(10) sub  $s1 ← $t4, $t5
(11) add  $s0 ← $t3, $t4
(12) sllv $s4 ← $t7, $s1
(13) srlv $s3 ← $t6, $s0
(14) sllv $s5 ← $s0, $s1
(15) add  $s6 ← $s3, $s4
(16) add  $s7 ← $s4, $s6
(17) srlv $t0 ← $s6, $s7
(18) srlv $t1 ← $t0, $s7
```

(a) Draw the dataflow graph of the program. Represent instructions as numbered nodes (01 through 18) and flow dependencies as directed edges (arrows).
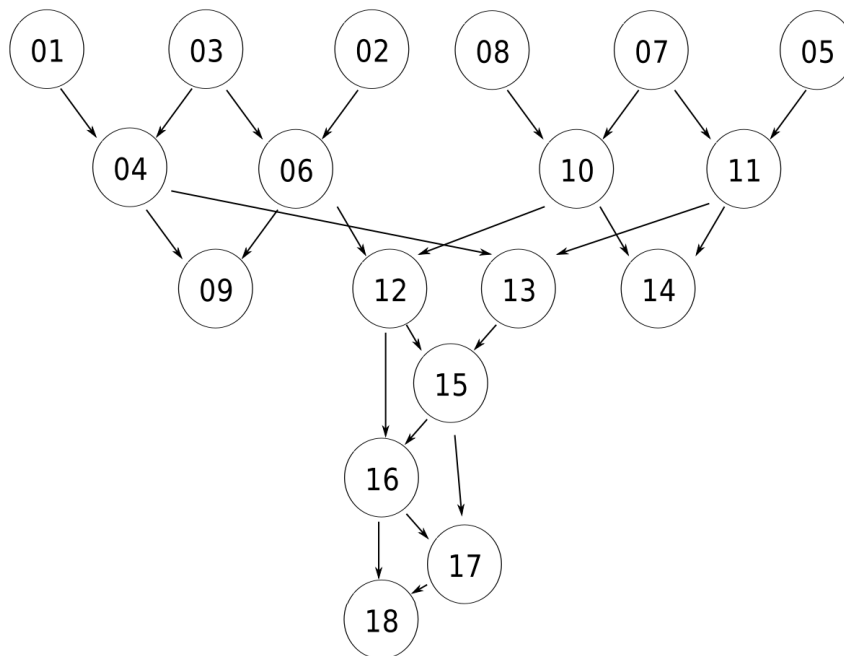
(b) When you run the tool with its settings targeted for a particular VLIW machine, you find that the resulting VLIW code has 9 VLIW instructions. What minimum value of N must the target VLIW machine have?

> N = 3. If N = 2, then the VLIW program must have at least 11 MIPS instructions, and the number of VLIW instructions either stays the same or decreases as width is increased by one MIPS instruction.

(c) Write the MIPS instruction numbers (from the code above) corresponding to each VLIW instruction, for this value of N. When there is more than one MIPS instruction that could be placed into a VLIW instruction, choose the instruction that comes earliest in the original MIPS program.

	MIPS Instr No	MIPS Instr No	MIPS Instr No	MIPS Instr No	MIPS Instr No	MIPS Instr No	MIPS Instr No	MIPS Instr No	MIPS Instr No	MIPS Instr No
VLIW Instr.1:	01	02	03							
VLIW Instr.2:	04	05	06							
VLIW Instr.3:	07	08	09							
VLIW Instr.4:	10	11								
VLIW Instr.5:	12	13	14							
VLIW Instr.6:	15									
VLIW Instr.7:	16									
VLIW Instr.8:	17									
VLIW Instr.9:	18									

(d) You find that the code is still not fast enough when it runs on the VLIW machine, so you contact the VLIW machine vendor to buy a machine with a larger machine-width "N". What minimum value of N would yield the maximum possible performance (i.e., the fewest VLIW instructions), assuming that all MIPS instructions (and thus VLIW instructions) complete with the same fixed latency and assuming no cache misses?

> N = 6. This is the maximum width of the dataflow graph and results in 7 VLIW instructions (see below). If N = 5, then the VLIW program will instead have 8 VLIW instructions. Increasing N further does not allow any more MIPS instructions to be parallelized in wider VLIW instructions.

(e) Write the MIPS instruction numbers corresponding to each VLIW instruction, for this optimal value of N. Again, as in part (c) above, pack instructions such that when more than one instruction can be placed in a given VLIW instruction, the instruction that comes first in the original MIPS code is chosen.

	MIPS Instr No	MIPS Instr No	MIPS Instr No	MIPS Instr No	MIPS Instr No	MIPS Instr No	MIPS Instr No	MIPS Instr No	MIPS Instr No	MIPS Instr No
VLIW Instr.1:	01	02	03	05	07	08				
VLIW Instr.2:	04	06	10	11						
VLIW Instr.3:	09	12	13	14						
VLIW Instr.4:	15									
VLIW Instr.5:	16									
VLIW Instr.6:	17									
VLIW Instr.7:	18									
VLIW Instr.8:										
VLIW Instr.9:										

(f) A competing processor design company builds an in-order superscalar processor with the same machine-width N as the width you found above in part(b). The machine has the same clock frequency as the VLIW processor. When you run the original MIPS program on this machine, you find that it executes slower than the corresponding VLIW progra m on the VLIW machine in part (b). Why could this be the case?

Concurrently fetched instructions can be dependent in a superscalar processor, requiring bubbles in the pipeline to be processed. A VLIW code translator can reorder instructions to minimize such bubbles. *Note that the superscalar processor is in-order in this question.*

(g) When you run some other program on this superscalar machine, you find it runs faster than the corresponding VLIW program on the VLIW machine. Why could this be the case?

VLIW code must have explicit NOPs; the superscalar processor does not require these NOPs. Higher code density results in a higher I-cache hit rate and lower required fetch bandwidth.

7 VLIW III

Consider a VLIW (very long instruction word) CPU that uses the long instruction format shown in Table 5. Each long instruction is composed of four short instructions, but there are restrictions on which type of instruction may go in which of the four slots.

MEMORY	INTEGER	CONTROL	FLOAT

Table 5: VLIW instruction format.

Table 6 provides a detailed description of the available short instructions and the total execution latency of each type of short instruction. Each short instruction execution unit is fully pipelined, and its result is available *on the cycle* given by the latency, e.g., a `CONTROL` instruction's results (if any) are available for other instructions to use *in the next cycle*.

Category	Latency (cycles)	Instruction(s)	Description	Functionality
CONTROL	1	BEQ LABEL, Rs1, Rs2	Branch IF equal	IF Rs1 == Rs2: PC = LABEL
		NOP	No operation	PC = Next PC
MEMORY	3	LD Rd, [Rs]	Memory load	Rd = MEM[Rs]
INTEGER	2	IADD Rd, Rs1, Rs2	Integer add	Rd = Rs1 + Rs2
FLOAT	4	FADD Rd, Rs1, Rs2	Floating-point add	Rd = Rs1 + Rs2

Table 6: Instruction latencies and descriptions.

Consider the piece of code given in Table 7. Unfortunately, it is written in terms of short instructions that cannot be directly input to the VLIW CPU.

	Instruction		Notes
	< Initialize R0-R2 >		R0-R2 point to valid memory
	LOOP:		
1	LD	R0, [R0]	R0 <- MEM[R0]
2	LD	R1, [R1]	R1 <- MEM[R1]
3	IADD	R4, R0, R1	R4 <- R0 + R1
4	FADD	R5, R0, R4	R5 <- R0 + R4
5	LD	R6, [R2]	R6 <- MEM[R2]
6	LD	R2, [R0]	R2 <- MEM[R0]
7	FADD	R3, R1, R6	R3 <- R1 + R6
8	IADD	R4, R2, R4	R4 <- R2 + R4
9	IADD	R5, R5, R4	R5 <- R5 + R4
10	IADD	R0, R6, R2	R0 <- R6 + R2
11	IADD	R0, R0, R3	R0 <- R0 + R3
12	BEQ	LOOP, R0, R5	GOTO LOOP if R0 == R5

Table 7: Proposed code for calculating the results of the next Swiss referendum.

(a) Warm-up: which of the following are goals of VLIW CPU design (circle all that apply)?

  (I) Simplify code compilation.

 (II) Simplify application development.

(III) Reduce overall hardware complexity.

(IV) Simplify hardware inter-instruction dependence checking.

(V) Reduce processor fetch width.

(b) Your task is to determine the optimal VLIW scheduling of the short instructions by hand. Fill in the following table with the highest performance (i.e., fewest number of execution cycles) instruction sequence that may be directly input into the VLIW CPU and have the same functionality as the code in Table 7. Where possible, you may write instruction IDs corresponding to the numbers given in Table 7 and leave any NOP instructions as blank slots.

Consider **only one loop iteration** (including the BEQ instruction), **ignore initialization** and any cross-iteration optimizations (e.g., loop unrolling), and **do not** optimize the code by removing or changing existing instructions.

Cycle	MEMORY	INTEGER	CONTROL	FLOAT
1	1 (LD R0, [R0])			
2	2 (LD R1, [R1])			
3	5 (LD R6, [R2])			
4	6 (LD R2, [R0])			
5		3 (IADD R4, R0, R1)		
6				7 (FADD R3, R1, R6)
7		8 (IADD R4, R2, R4)		4 (FADD R5, R0, R4)
8		10 (IADD R0, R6, R2)		
9		8 (IADD R4, R2, R4)		
10		11 (IADD R0, R0, R3)		
11		9 (IADD R5, R5, R4)		
12				
13			12 (BEQ LOOP, R0, R5)	
14				
15				
16				
17				
18				
19				
20				

Hint: you should not require more than 20 cycles.

Note: Instruction 8 may go in EITHER of the red slots.

(c) How many total cycles are required to complete execution of all instructions in the previous question? Ignore pipeline fill overheads and assume the instruction latencies given in Table 6.

13

(d) What is the utilization of the instruction scheduling slots (computed as the ratio of utilized slots to total execution slots throughout execution)?

(12 slots used) / (13 cycles * 4 slots/cycle) = $\frac{3}{13}$ ≈ 23.1%

8   Systolic Arrays I

Figure 1 shows a systolic array processing element.

Each processing element takes in two inputs, M and N, and outputs P and Q. Each processing element also contains an "accumulator" R that can be read from and written to. The initial value of the "accumulator" is 0.

Figure 2 shows a systolic array composed of 9 processing elements. The smaller boxes are the inputs to the systolic array and the larger boxes are the processing elements. You will program this systolic array to perform the following calculation:

$$
\begin{bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix}
$$

In each time cycle, each processing element will take in its two inputs, perform any necessary actions, and write on its outputs. The time cycle labels on the input boxes determine which time cycle the inputs will be fed into their corresponding processing elements. Any processing element input that is not driven will default to 0, and any processing element that has no output arrow will have its output ignored.

After all the calculations finish, each processing element's "accumulator" will hold one element of the final result matrix, arranged in the correct order.

(a) Please describe the operations that each individual processing element performs, using mathe- matical equations and the variables M, N, P, Q and R.
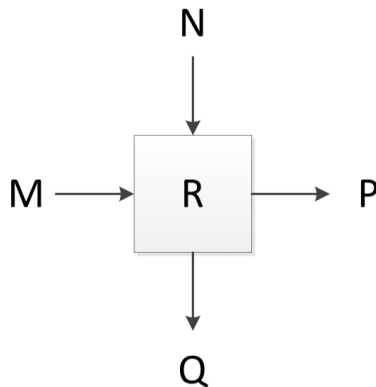
Figure 1: A systolic array processing element

$$
P = \boxed{M}
$$

$$
Q = \boxed{N}
$$

$$
R = \boxed{R + M \times N}
$$

(b) Please fill in all 30 input boxes in Figure **??** so that the systolic array computes the correct matrix multiplication result described on the previous page. (Hint: Use a_{ij} and b_{ij}.)
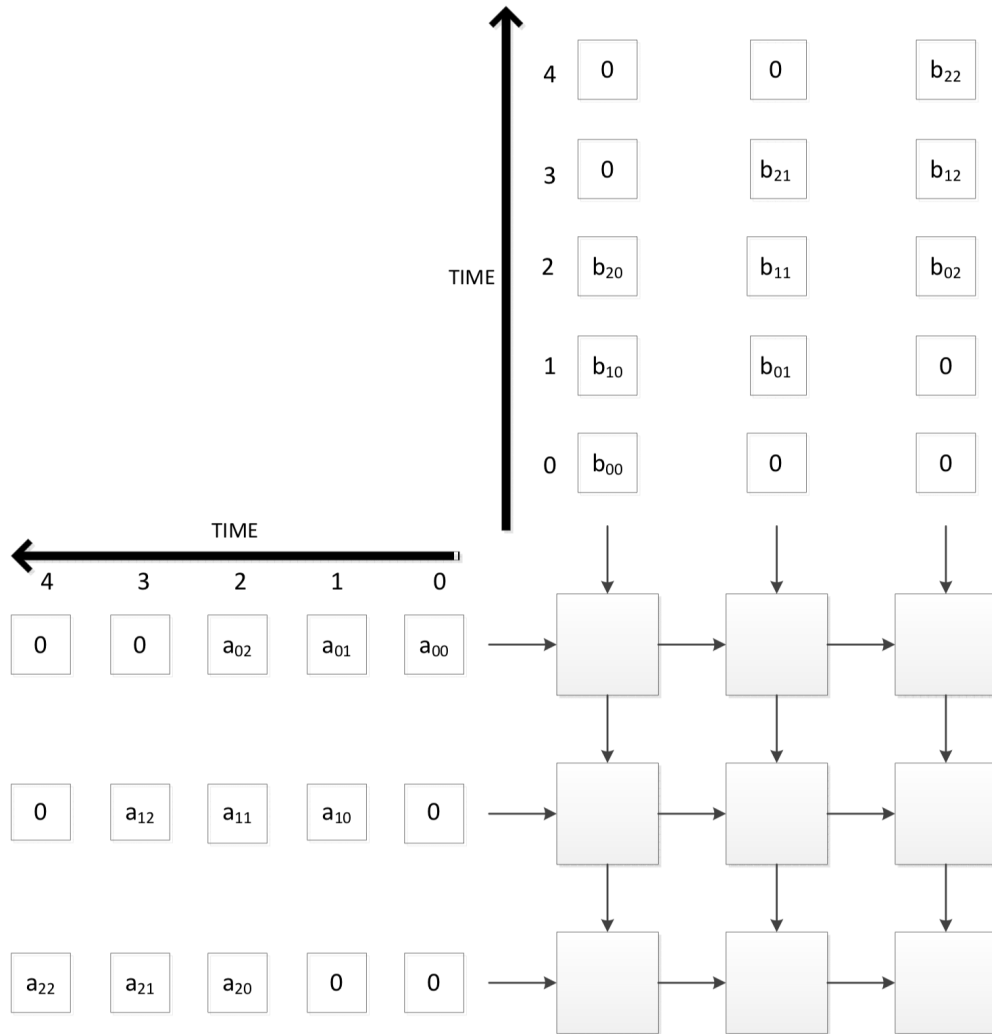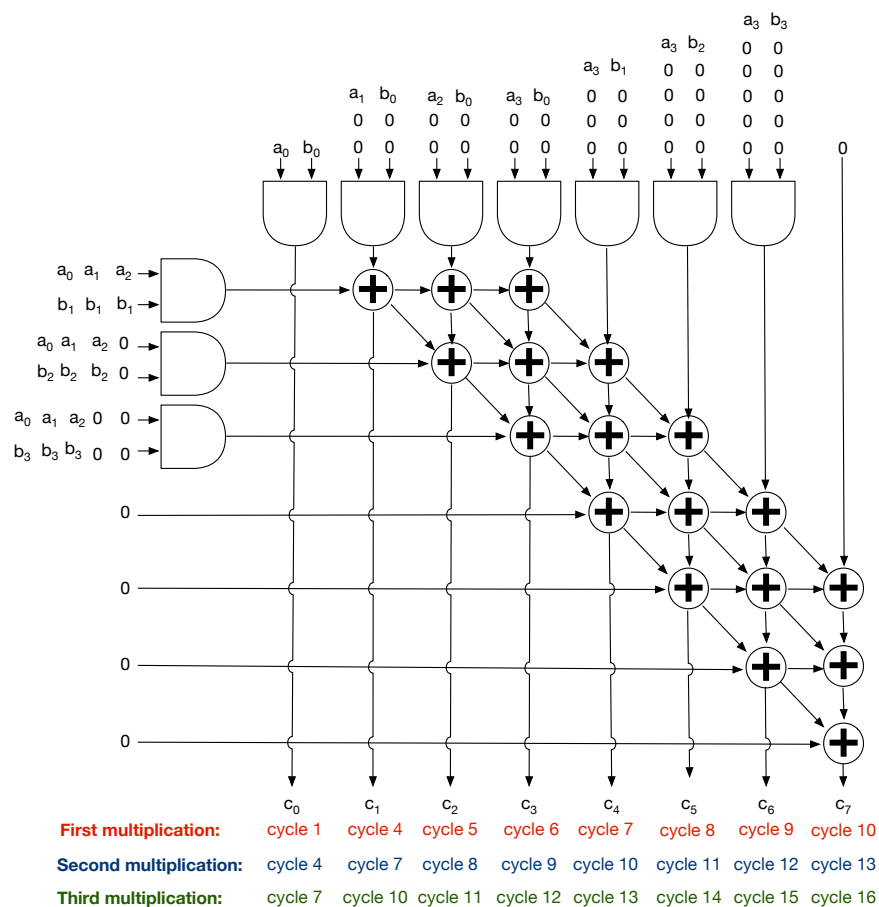

Figure 2: A systolic array

9 Systolic Arrays II

The following diagram is a systolic array that performs the multiplication of two 4-bit binary numbers (**a** and **b**). For example, if **a**=1110 and **b**=1011, the result of the multiplication is **c**=10011010:

$$
\begin{array}{r}
1011 \\
\times\ 1110 \\
\hline
0000 \\
1011 \\
1011 \\
+\ 1011 \\
\hline
10011010
\end{array}
\tag{1}
$$

The input to the systolic arrays is through the AND gates. The figure shows which bits of the two numbers **a** and **b** are inserted into each AND gate. However, the figure does *not* indicate in which cycle each input is issued. Make the folowing assumptions:

- The latency of each adder is one cycle.

- Vertical arrows propagate the sum to the next adder.

- Diagonal arrows propagate the carry to the next adder.

- Horizontal arrows propagate the output of the AND gates in each row.

- An adder adds the value of its three inputs (vertical, diagonal and horizontal inputs)

- An adder can hold a value for only one cycle.

	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7
First multiplication:	cycle 1	cycle 4	cycle 5	cycle 6	cycle 7	cycle 8	cycle 9	cycle 10
Second multiplication:	cycle 4	cycle 7	cycle 8	cycle 9	cycle 10	cycle 11	cycle 12	cycle 13
Third multiplication:	cycle 7	cycle 10	cycle 11	cycle 12	cycle 13	cycle 14	cycle 15	cycle 16

(a) How many cycles does it take to perform *one* multiplication of two 4-bit binary numbers in this systolic array? Indicate 1) in which cycle each bit is inputted in the systolic array and 2) in which cycle each bit of the result is produced.

> 10 cycles. The cycle each input is inserted into the array is indicated in the figure.
> c_0 can be produced any time during the 10 cycles that the multiplication takes. In the example solution of the figure:
> c_0 is produced in cycle 1
> c_1 is produced in cycle 4
> c_2 is produced in cycle 5
> c_3 is produced in cycle 6
> c_4 is produced in cycle 7
> c_5 is produced in cycle 8
> c_6 is produced in cycle 9
> c_7 is produced in cycle 10

(b) How many cycles does it take to perform N consecutive multiplications of two 4-bit binary numbers in this systolic array?

> 10+3*(N-1) cycles.
> Performing one multiplication takes 10 cycles, but subsequent multiplications can be pipelined. We can issue one new multiplication operation every 3 cycles. We show in the solution the cycles in which the results of the first 3 multiplications of two 4-bit binary numbers are produced (the inputs of the second and third multiplication are not shown in the figure). Performing N multiplications will take 10+3*(N-1) cycles.

We did not cover delayed branching in detail in lecture, so these two questions are for your benefit and learning solely.

10 Delayed Branching I (Extra)

A machine has a five-stage pipeline consisting of fetch, decode, execute, mem and write-back stages. The machine uses delay slots to handle control dependences. Jump targets, branch targets and destinations are resolved in the execute stage.

(a) What is the number of delay slots needed to ensure correct operation?

2

(b) Which instruction(s) in the assembly sequences below would you place in the delay slot(s), assuming the number of delay slots you answered for part(a)? Clearly rewrite the code with the appropriate instruction(s) in the delay slot(s).

(I)
```
ADD R5 <- R4, R3
OR  R3 <- R1, R2
SUB R7 <- R5, R6
J   X
```
Delay Slots
```
LW  R10 <- (R7)
ADD R6 <- R1, R2
X:
```

Solution:
```
ADD R5 <- R4, R3
J X
OR R3 <- R1, R2
SUB R7 <- R5, R6
LW R10 <- (R7)
ADD R6 <- R1, R2
X:
```

(II) `ADD R5 <- R4, R3`
      `OR  R3 <- R1, R2`
      `SUB R7 <- R5, R6`
      `BEQ R5 <- R7, X`
      ┌───────────┐
      │ Delay Slots │
      └───────────┘
      `LW  R10 <- (R7)`
      `ADD R6 <- R1, R2`
      `X:`

Solution:

ADD R5 <- R4, R3
SUB R7 <- R5, R6
BEQ R5 <- R7, X
OR R3 <- R1, R2
NOP
LW R10 <- (R7)
ADD R6 <- R1, R2
X:

(III) `ADD R2 <- R4, R3`
      `OR  R5 <- R1, R2`
      `SUB R7 <- R5, R6`
      `BEQ R5 <- R7, X`
      ┌───────────┐
      │ Delay Slots │
      └───────────┘
      `LW  R10 <- (R7)`
      `ADD R6 <- R1, R2`
      `X:`

Solution:

ADD R2 <- R4, R3
OR R5 <- R1, R2
SUB R7 <- R5, R6
BEQ R5 <- R7, X
NOP
NOP
LW R10 <- (R7)
ADD R6 <- R1, R2
X:

(c) Can you modify the pipeline to reduce the number of delay slots (without introducing branch prediction)? Clearly state your solution and explain why.

Move the resolution of jump targets and branch targets and destinations to the decode stage.
Jumps and branches would get resolved one cycle earlier and hence one delay slot would be enough to ensure correct operation.

11 Delayed Branching II (Extra)

You are designing an ISA that uses delayed branch instructions. You are trying to decide how many instructions to place into the branch delay slot. How many branch delay slots would you need for the following different implementations? Explain your reasoning briefly.

(a) An in-order processor where conditional branches resolve during the 4th stage

> 3

(b) An out-of-order processor with 64 unified reservation station entries where conditional branches resolve during the 2nd cycle of branch execution. The processor has 15 pipeline stages until the start of the execution stages

> We don't know.

The following exercises are old exam questions that are conceptually similar to the ones above, but with slight alterations. We do not expect or recommend you to solve all of them, unless you think you are struggling with a particular concept, or would like to do practice runs on these old exam questions.

12 Branch Prediction (Extra)

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 to reduce the overhead of the branches. The ISA implements the instructions given in the following table:

Instructions	Description
la R_i, Address	load the effective $Address$ into R_i
move R_i, R_j	$R_i \leftarrow R_j$
move R_i, (R_j)	$R_i \leftarrow \text{Memory}[R_j]$
move (R_i), R_j	$\text{Memory}[R_i] \leftarrow R_j$
li R_i, Imm	$R_i \leftarrow \text{Imm}$
add R_i, R_j, R_k	$R_i \leftarrow R_j + R_k$
addi R_i, R_j, Imm	$R_i \leftarrow R_j + \text{Imm}$
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 $R_i < \text{Memory}[R_j]$; set zero flag, if $R_i = \text{Memory}[R_j]$
cmpi R_i, Imm	Compare: Set sign flag, if $R_i < \text{Imm}$; set zero flag, if $R_i = \text{Imm}$.
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.

```
 1          la R0, Array
 2          move R6, R0
 3          li R1, 4
 4          move R5, R1
 5          move R7, R1
 6          move R2, R0
 7          addi R2, R2, 4
 8  Loop:
 9          move R3, (R2)
10          cmp R3, (R0)
11          jg Next_Iteration
12          move R4, (R0)
13          move (R0), R3
14          move (R2), R4
15  Next_Iteration:
16          addi R0, R0, 4
17          addi R2, R2, 4
18          addi R1, R1, -1
19          cmpi R1, 0
20          jnz Loop
21          move R1, R7
22          addi R5, R5, -1
23          move R0, R6
24          move R2, R0
25          addi R2, R2, 4
26          cmpi R5, 0
27          jnz Loop
28          halt
29  .data
30  Array: word 5, 20, 1, -5, 34
```

(a) What would be the prediction accuracy using a shared one-bit-history (last time) branch predictor for *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) What would be the prediction accuracy using a shared two-bit-history (two-bit counter) branch predictor for *all* the branches? Assume that the initial state of the two-bit-history branch predictor is "weakly taken". The "weakly taken" state transitions to "weakly not-taken" state on misprediction. Similarly, the "weakly not-taken" taken state transitions to "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) What would be the prediction accuracy using two-bit-history (two-bit counter) branch predictor 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

13 Systolic Arrays (Extra)

A systolic array consist of 3x4 Processing Elements (PEs), interconnected as shown in Figure 3. The inputs of the systolic array are labeled as H0, H1, H2 and V0,V1,V2,V3. Figure 4 shows the PE logic, which performs a multiply and accumulate operation (MAC), and it saves the result in an internal register (reg). Figure 4 also shows how each PE propagates its inputs. We make the following assumptions:

- The latency of each MAC is one cycle.

- The propagation of the values from i_0 to o_0, and from i_1 to o_1, takes one cycle.

- The initial value of all registers is zero.

- You can input a value more than once in the systolic array.
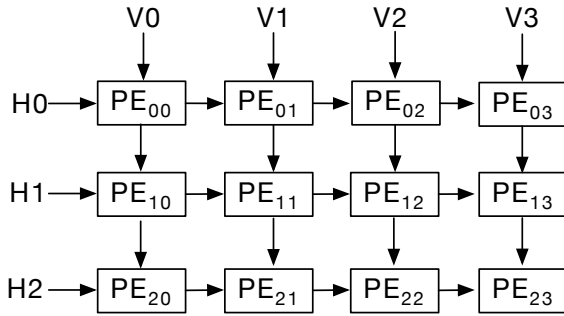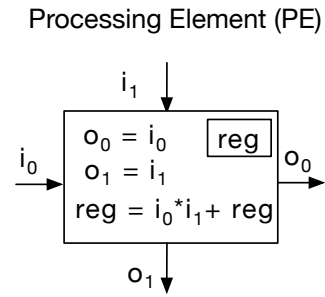
Figure 3: PE array

Figure 4: Processing Element (PE)

Your goal is to use this systolic array to perform the convolution of a 3x3 image (matrix I) with three 2x2 filters (matrices F, G, and H), to obtain three outputs (matrices O, U, and E):

$$
\begin{matrix} I_{00} & I_{01} & I_{02} \\ I_{10} & I_{11} & I_{12} \\ I_{20} & I_{21} & I_{22} \end{matrix}
\circledast
\begin{matrix} F_{00} & F_{01} \\ F_{10} & F_{11} \end{matrix}
=
\begin{matrix} O_{00} & O_{01} \\ O_{10} & O_{11} \end{matrix}
$$

$$
\begin{matrix} I_{00} & I_{01} & I_{02} \\ I_{10} & I_{11} & I_{12} \\ I_{20} & I_{21} & I_{22} \end{matrix}
\circledast
\begin{matrix} G_{00} & G_{01} \\ G_{10} & G_{11} \end{matrix}
=
\begin{matrix} U_{00} & U_{01} \\ U_{10} & U_{11} \end{matrix}
$$

$$
\begin{matrix} I_{00} & I_{01} & I_{02} \\ I_{10} & I_{11} & I_{12} \\ I_{20} & I_{21} & I_{22} \end{matrix}
\circledast
\begin{matrix} H_{00} & H_{01} \\ H_{10} & H_{11} \end{matrix}
=
\begin{matrix} E_{00} & E_{01} \\ E_{10} & E_{11} \end{matrix}
$$

As an example, the convolution of the matrix I with the filter F is computed as follows:

- $O_{00} = I_{00} * F_{00} + I_{01} * F_{01} + I_{10} * F_{10} + I_{11} * F_{11}$

- $O_{01} = I_{01} * F_{00} + I_{02} * F_{01} + I_{11} * F_{10} + I_{12} * F_{11}$

- $O_{10} = I_{10} * F_{00} + I_{11} * F_{01} + I_{20} * F_{10} + I_{21} * F_{11}$

- $O_{11} = I_{11} * F_{00} + I_{12} * F_{01} + I_{21} * F_{10} + I_{22} * F_{11}$

You should compute the three convolutions in the minimum possible amount of cycles. Fill the following table with:

1. The input values (matrices I, F, G, and H) in the correct input ports of the systolic array (the values can be repeated).

2. The output values and the corresponding PE where the outputs (matrices O, U, and E) are generated.

Fill the gaps only with relevant information.

cycle	H0	H1	H2	V0	V1	V2	V3	PE_{00}	PE_{01}	PE_{02}	PE_{03}	PE_{10}	PE_{11}	PE_{12}	PE_{13}	PE_{20}	PE_{21}	PE_{22}	PE_{23}
0	F_{00}			I_{00}															
1	F_{01}	G_{00}		I_{01}	I_{01}														
2	F_{10}	G_{01}	H_{00}	I_{10}	I_{02}	I_{10}													
3	F_{11}	G_{10}	H_{01}	I_{11}	I_{11}	I_{11}	I_{11}	O_{00}											
4		G_{11}	H_{10}		I_{12}	I_{20}	I_{12}		O_{01}			U_{00}							
5			H_{11}			I_{21}	I_{21}			O_{10}			U_{01}			E_{00}			
6							I_{22}				O_{11}			U_{10}			E_{01}		
7															U_{11}			E_{10}	
8																			E_{11}
9																			
10																			
11																			
12																			
13																			
14																			
15																			