Design of Digital Circuits (252-0028-00L), Spring 2021
Optional HW 5: Branch Prediction, VLIW, and Systolic Arrays

Instructor: Prof. Onur Mutlu
TAs: Juan Gomez-Luna, Jisung Park, Hasan Hassan, Mohammed Alser, Lois Orosa, Minesh Patel, Jawad Haj-Yahya, Haiyu Mao, Behzad Salami, Jeremie Kim, Giray Yaglikci, Can Firtina, Geraldo De Oliveira Junior, Rahul Bera, Konstantinos Kanellopoulos, Nika Mansouri, Gagandeep Singh

Released: Wednesday, May 19, 2021

1 Delayed Branching

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?

(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:

(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:

(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:

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

2 Delayed Branching II

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

(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

3 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 */
  }
}
```

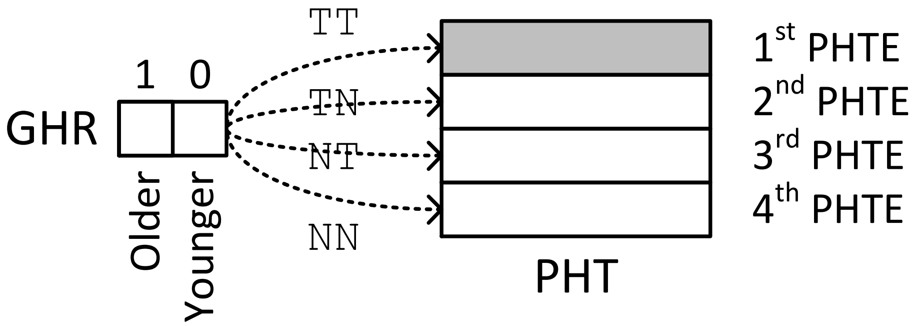
(a) Of the four branches, list all those that exhibit *local correlation*, if any.

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

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.

4 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	
TN	
NT	
NN	

Show your work here.

(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.

5 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.

5.1 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?

5.2 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:

(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:

(c) **Backward taken, Forward not taken (BTFN)**
Could this be the mystery branch predictor?

`YES` `NO`

Explain:

(d) **Forward taken, Backwards not taken (FTBN)**
Could this be the mystery branch predictor?

`YES` `NO`

Explain:

(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:

6 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:

Branch 2:

Branch 3:

Branch 4:

(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:

Branch 2:

Branch 3:

Branch 4:

(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):

Branch 2 (ii):

Branch 3 (i):

Branch 3 (ii):

7 VLIW I

Explain the motivation for VLIW in one sentence.

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			
2			
3			
4			
5			
6			
7			
8			
9			
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.

(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			
2			
3			
4			
5			
6			
7			
8			
9			
10			
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?

(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)

8 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?

(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:										
VLIW Instr.2:										
VLIW Instr.3:										
VLIW Instr.4:										
VLIW Instr.5:										
VLIW Instr.6:										
VLIW Instr.7:										
VLIW Instr.8:										
VLIW Instr.9:										

(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?

(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:										
VLIW Instr.2:										
VLIW Instr.3:										
VLIW Instr.4:										
VLIW Instr.5:										
VLIW Instr.6:										
VLIW Instr.7:										
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?

(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?

9 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				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19				
20				

Hint: you should not require more than 20 cycles.

(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.

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

10 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.

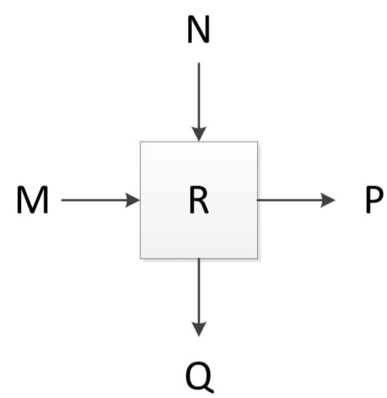


Figure 1: A systolic array processing element

P =

Q =

R =

(b) Please fill in all 30 input boxes in Figure 2 so that the systolic array computes the correct matrix multiplication result described on the previous page. (Hint: Use a_{ij} and b_{ij}.)

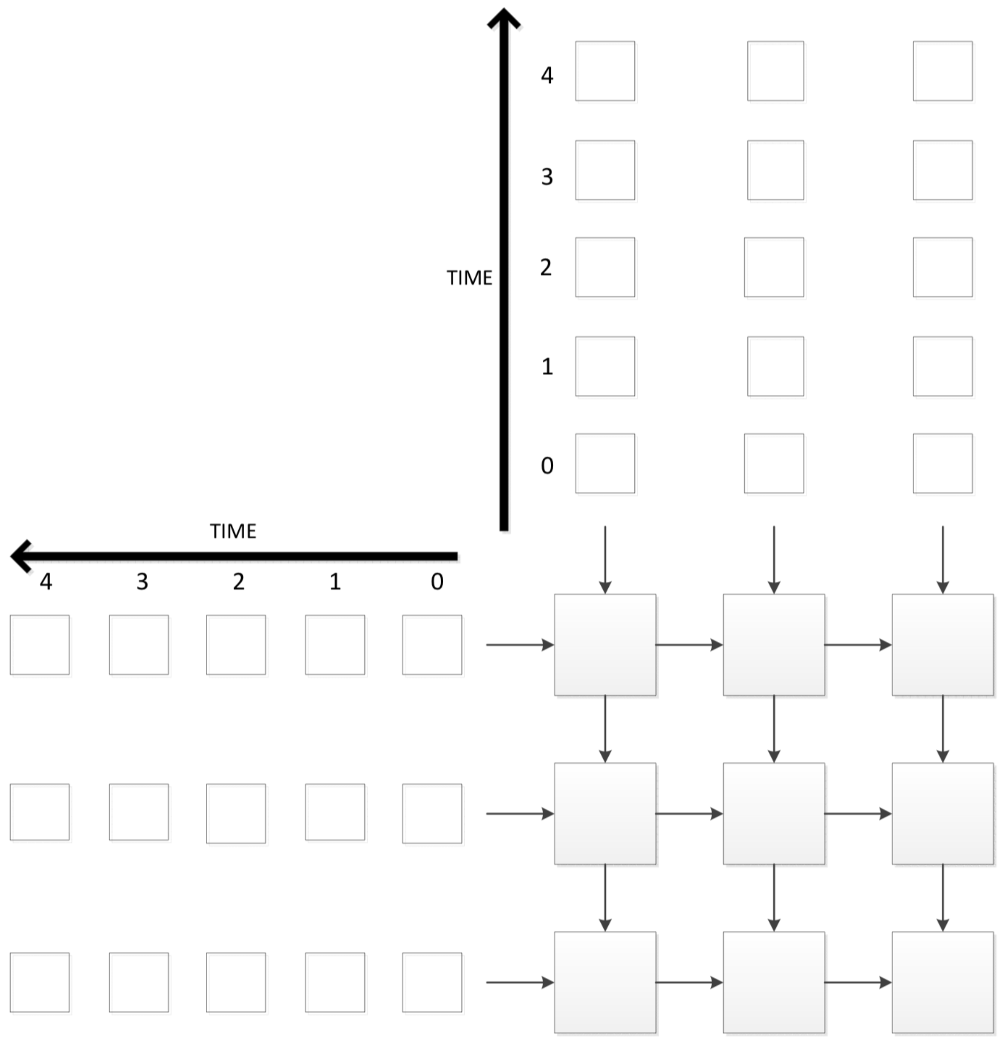


Figure 2: A systolic array

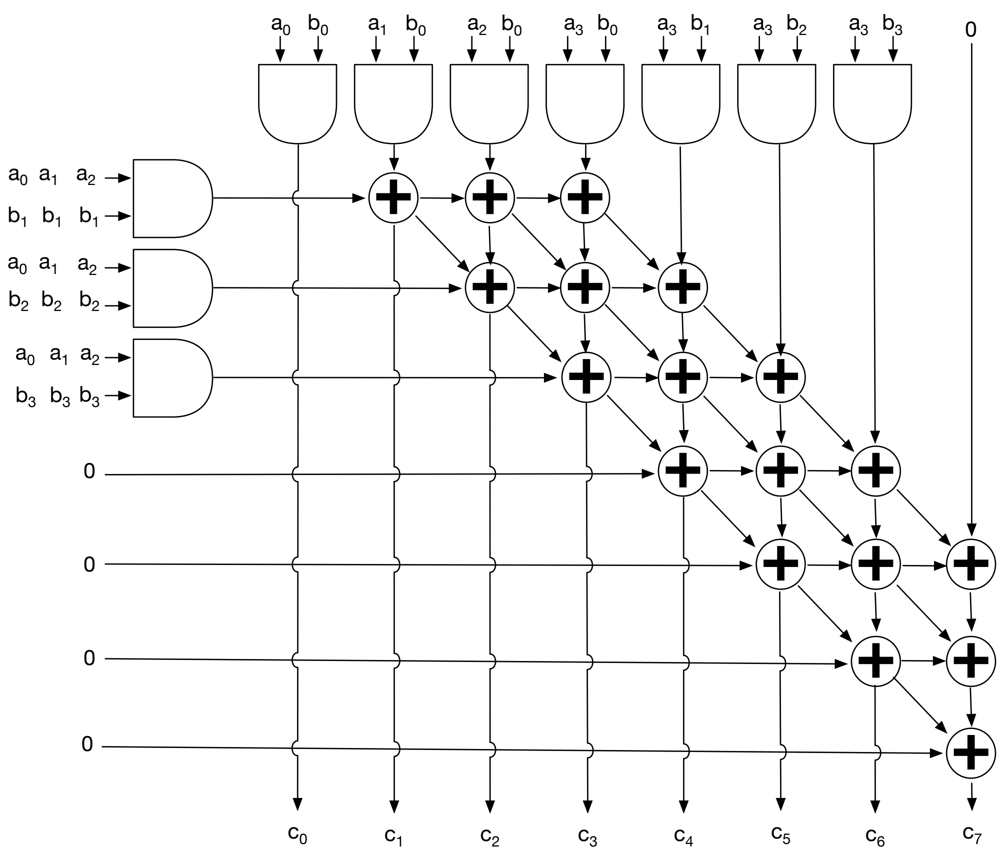
11 Systolic Arrays II

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.



(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.

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