

DESIGN OF DIGITAL CIRCUITS (252-0028-00L), SPRING 2018
OPTIONAL HW 4: OUT-OF-ORDER EXECUTION, DATAFLOW,
BRANCH PREDICTION, VLIW, AND FINE-GRAINED MULTITHREADING

Instructor: Prof. Onur Mutlu

TAs: Juan Gomez Luna, Hasan Hassan, Arash Tavakkol, Minesh Patel, Jeremie Kim, Giray Yaglikci

Assigned: Tuesday, May 15, 2018

1 Tomasulo's Algorithm

Remember that Tomasulo's algorithm requires tag broadcast and comparison to enable wake-up of dependent instructions. In this question, we will calculate the number of tag comparators and size of tag storage required to implement Tomasulo's algorithm in a machine that has the following properties:

- 8 functional units where each functional unit has a dedicated separate tag and data broadcast bus
- 32 64-bit architectural registers
- 16 reservation station entries per functional unit
- Each reservation station entry can have two source registers

Answer the following questions. Show your work for credit.

- (a) What is the number of tag comparators per reservation station entry?

- (b) What is the total number of tag comparators in the entire machine?

- (c) What is the (minimum possible) size of the tag?

- (d) What is the (minimum possible) size of the register alias table (or, frontend register file) in bits?

- (e) What is the total (minimum possible) size of the tag storage in the entire machine in bits?

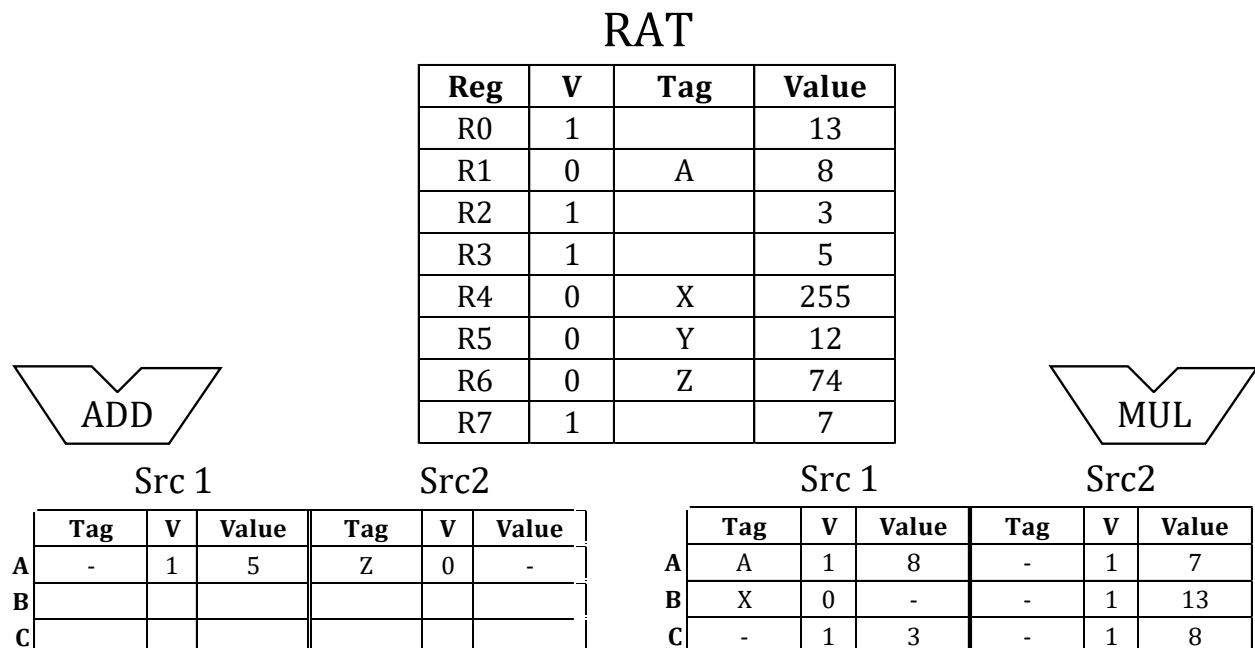
2 Out-of-Order Execution I

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 Tomasulos algorithm. Your job is to determine the original sequence of **five 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 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 a value is captured (at a reservation station) or written back (to a register) in this machine, the old tag that was previously at that location is *not cleared*; only the valid bit is set.
- When an instruction in a reservation station finishes executing, the reservation station is cleared.
- Both the adder and multiplier are fully pipelined. An add instruction takes 2 cycles. A multiply instruction takes 4 cycles.
- When an instruction completes execution, it broadcasts its result. A dependent instructions can begin execution in the next cycle if it has all its operands available.
- When multiple instructions are ready to execute at a functional unit, the oldest ready instruction is chosen.

Initially, the machine is empty. Five instructions then are fetched, decoded, and dispatched into reservation stations. When the final instruction has been fetched and decoded, one instruction has already been written back. Pictured below is the state of the machine at this point, after the fifth instruction has been fetched and decoded:



- (a) Give the five 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 \leftarrow source1, source2.”

		\leftarrow		,	
		\leftarrow		,	
		\leftarrow		,	
		\leftarrow		,	
		\leftarrow		,	

- (b) 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 five 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 fifth instruction.

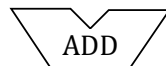
As we saw in class, use “F” for fetch, “D” for decode, “En” to signify the nth cycle of execution for an instruction, and “W” to signify writeback. You may or may not need all columns shown.

Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction:														
Instruction:														
Instruction:														
Instruction:														
Instruction:														

Finally, show the state of the RAT and reservation stations after **10 cycles** in the blank figures below.

RAT

Reg	V	Tag	Value
R0			
R1			
R2			
R3			
R4			
R5			
R6			
R7			



Src 1

Src2

	Tag	V	Value	Tag	V	Value
A						
B						
C						



Src 1

Src2

	Tag	V	Value	Tag	V	Value
A						
B						
C						

3 Out-of-Order Execution II

In this problem, we consider an in-order fetch, out-of-order dispatch, and in-order retirement execution engine that employs Tomasulo's algorithm. This engine behaves as follows:

- The engine has four main pipeline stages: Fetch (F), Decode (D), Execute (E), and Write-back (W).
- The engine can fetch one instruction per cycle, decode one instruction per cycle, and write back the result of one instruction per cycle.
- The engine has two execution units: 1) an adder for executing ADD instructions and 2) a multiplier for executing MUL instructions.
- The execution units are fully pipelined. The adder has two stages (E1-E2) and the multiplier has four stages (E1-E2-E3-E4). Execution of each stage takes one cycle.
- The adder has a two-entry reservation station and the multiplier has a four-entry reservation station.
- An instruction always allocates the first available entry of the reservation station (in top-to-bottom order) of the corresponding execution unit.
- Full data forwarding is available, i.e., during the last cycle of the E stage, the tags and data are broadcast to the reservation station and the Register Alias Table (RAT). For example, an ADD instruction updates the reservation station entries of the dependent instructions in E2 stage. So, the updated value can be read from the reservation station entry in the next cycle. Therefore, a dependent instruction can potentially begin its execution in the next cycle (after E2).
- The multiplier and adder have separate output data buses, which allow both the adder and the multiplier to update the reservation station and the RAT in the same cycle.
- An instruction continues to occupy a reservation station slot until it finishes the Write-back (W) stage. The reservation station entry is deallocated after the Write-back (W) stage.

3.1 Problem Definition

The processor is about to fetch and execute *six* instructions. Assume the *reservation stations (RS)* are all initially empty and the initial state of the *register alias table (RAT)* is given below in Figure (a). Instructions are fetched, decoded and executed as discussed in class. At some point during the execution of the six instructions, a snapshot of the state of the RS and the RAT is taken. Figures (b) and (c) show the state of the RS and the RAT at the snapshot time. A dash (–) indicates that a value has been cleared. A question mark (?) indicates that a value is unknown.

Reg	Valid	Tag	Value
R0	1	–	1900
R1	1	–	82
R2	1	–	1
R3	1	–	3
R4	1	–	10
R5	1	–	5
R6	1	–	23
R7	1	–	35
R8	1	–	61
R9	1	–	4

(a) Initial state of the RAT

Reg	Valid	Tag	Value
R0	1	?	1900
R1	0	Z	?
R2	1	?	12
R3	1	?	3
R4	1	?	10
R5	0	B	?
R6	1	?	23
R7	0	H	?
R8	1	?	350
R9	0	A	?

(b) State of the RAT at the snapshot time

ID	V	Tag	Value	V	Tag	Value
A	1	?	350	1	?	12
B	0	A	?	0	Z	?

+

ID	V	Tag	Value	V	Tag	Value
–	–	–	–	–	–	–
T	1	?	10	1	?	35
H	1	?	35	0	A	?
Z	1	?	82	0	H	?

×

(c) State of the RS at the snapshot time

3.2 (a) Data Flow Graph

Based on the information provided above, identify the instructions and complete the dataflow graph below for the six instructions that have been fetched. Please appropriately connect the nodes using edges and specify the direction of each edge. Label each edge with the destination architectural register and the corresponding Tag. *Note that you may **not** need to use all registers and/or nodes provided below.*

Register IDs:



Output

3.3 (b) Program Instructions

Fill in the blanks below with the six-instruction sequence in program order. When referring to registers, please use their architectural names (R0 through R9). Place the register with the smaller architectural name on the left source register box. For example, `ADD R8 \leftarrow R1, R5`.

<input type="text"/>	<input type="text"/>	\leftarrow	<input type="text"/>	,	<input type="text"/>
<input type="text"/>	<input type="text"/>	\leftarrow	<input type="text"/>	,	<input type="text"/>
<input type="text"/>	<input type="text"/>	\leftarrow	<input type="text"/>	,	<input type="text"/>
<input type="text"/>	<input type="text"/>	\leftarrow	<input type="text"/>	,	<input type="text"/>
<input type="text"/>	<input type="text"/>	\leftarrow	<input type="text"/>	,	<input type="text"/>
<input type="text"/>	<input type="text"/>	\leftarrow	<input type="text"/>	,	<input type="text"/>

4 Out-of-Order Execution III

A five instruction sequence executes according to Tomasulo's algorithm. Each instruction is of the form ADD DR,SR1,SR2 or MUL DR,SR1,SR2. ADDs are pipelined and take 9 cycles (F-D-E1-E2-E3-E4-E5-E6-WB). MULs are also pipelined and take 11 cycles (two extra execute stages). An instruction must wait until a result is in a register before it sources it (reads it as a source operand). For instance, if instruction 2 has a read-after-write dependence on instruction 1, instruction 2 can start executing in the next cycle after instruction 1 writes back (shown below).

```
instruction 1      |F|D|E1|E2|E3|.....|WB|
instruction 2      |F|D|-|-|.....|-|E1|
```

The machine can fetch one instruction per cycle, and can decode one instruction per cycle.

The register file before and after the sequence are shown below.

	Valid	Tag	Value		Valid	Tag	Value
R0	1		4	R0	1		310
R1	1		5	R1	1		5
R2	1		6	R2	1		410
R3	1		7	R3	1		31
R4	1		8	R4	1		8
R5	1		9	R5	1		9
R6	1		10	R6	1		10
R7	1		11	R7	1		21

(a) Before

(b) After

- (a) Complete the five instruction sequence in program order in the space below. Note that we have helped you by giving you the opcode and two source operand addresses for the fourth instruction. (The program sequence is unique.)

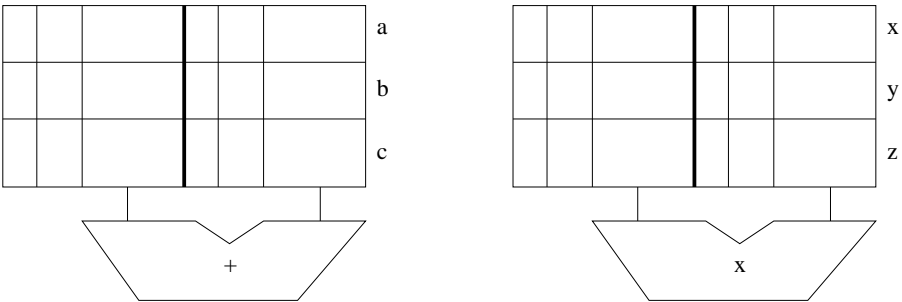
Give instructions in the following format: "opcode destination \leftarrow source1, source2."

<input type="text"/>	<input type="text"/>	\leftarrow	<input type="text"/>	,	<input type="text"/>
<input type="text"/>	<input type="text"/>	\leftarrow	<input type="text"/>	,	<input type="text"/>
<input type="text"/>	<input type="text"/>	\leftarrow	<input type="text"/>	,	<input type="text"/>
MUL	<input type="text"/>	\leftarrow	R6	,	R6
<input type="text"/>	<input type="text"/>	\leftarrow	<input type="text"/>	,	<input type="text"/>

- (b) In each cycle, a single instruction is fetched and a single instruction is decoded.

Assume the reservation stations are all initially empty. Put each instruction into the next available reservation station. For example, the first ADD goes into "a". The first MUL goes into "x". Instructions remain in the reservation stations until they are completed. Show the state of the reservation stations at the end of cycle 8.

Note: to make it easier for the grader, when allocating source registers to reservation stations, please always have the higher numbered register be assigned to source2.

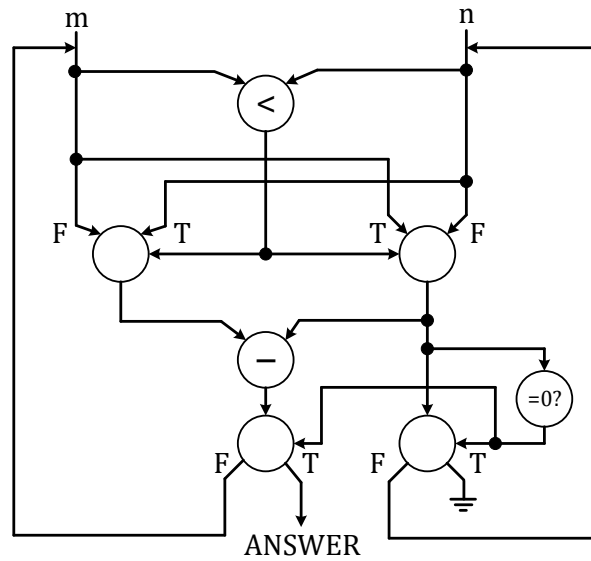


(c) Show the state of the Register Alias Table (Valid, Tag, Value) at the end of cycle 8.

	Valid	Tag	Value
R0			
R1			
R2			
R3			
R4			
R5			
R6			
R7			

5 Dataflow I

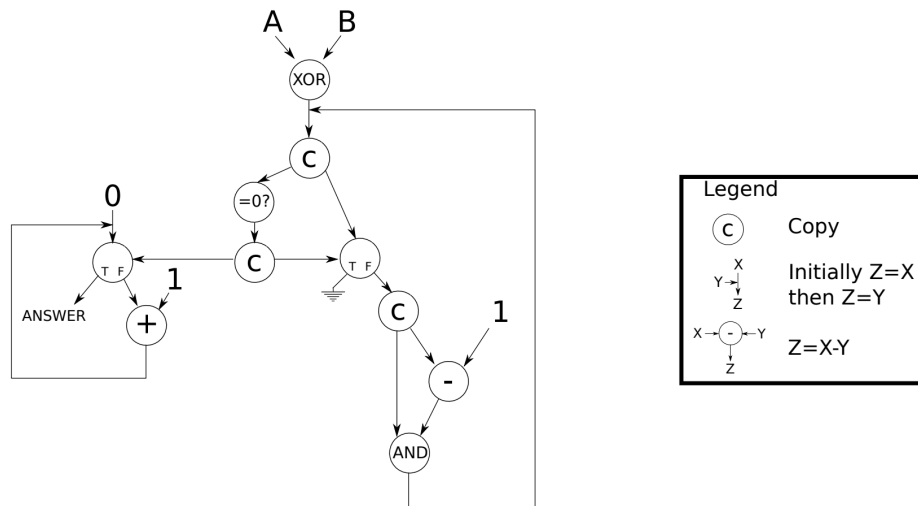
- (a) What does the following dataflow program do? Specify clearly in less than 10 words (one could specify this function in three words).





6 Dataflow II

Here is a dataflow graph representing a dataflow program:



Note that the inputs, A and B, are non-negative integers.

What does the dataflow program do? Specify clearly in less than 15 words.

7 Branch Prediction

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 % 5 == 0) { /* B3 */
        sum += val; /* TAKEN PATH for B3 */
    }
    if (val % 10 == 0) { /* B4 */
        sum += val; /* TAKEN PATH for B4 */
    }
}
```

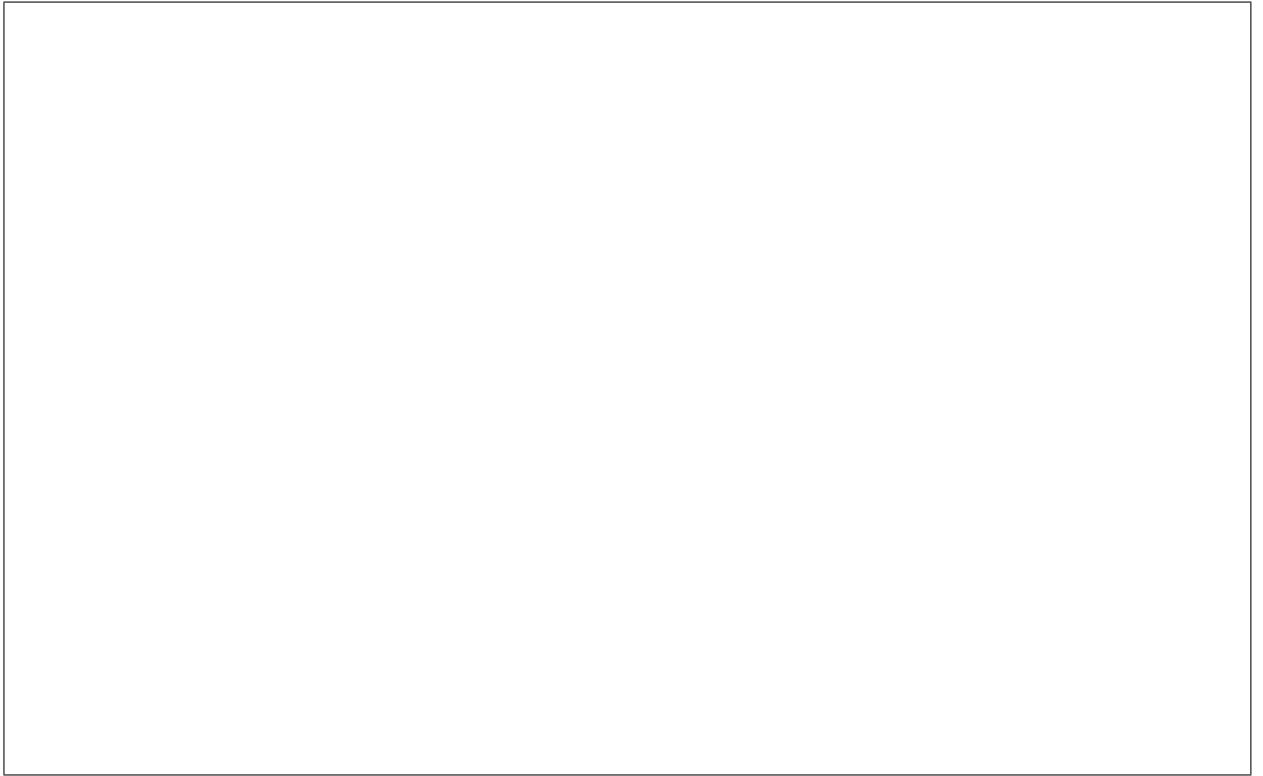
- (a) Of the four branches, list all those that exhibit local correlation, if any

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

- (c) 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: -10241023)
- 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.

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 ones 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...*



8 VLIW

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:										

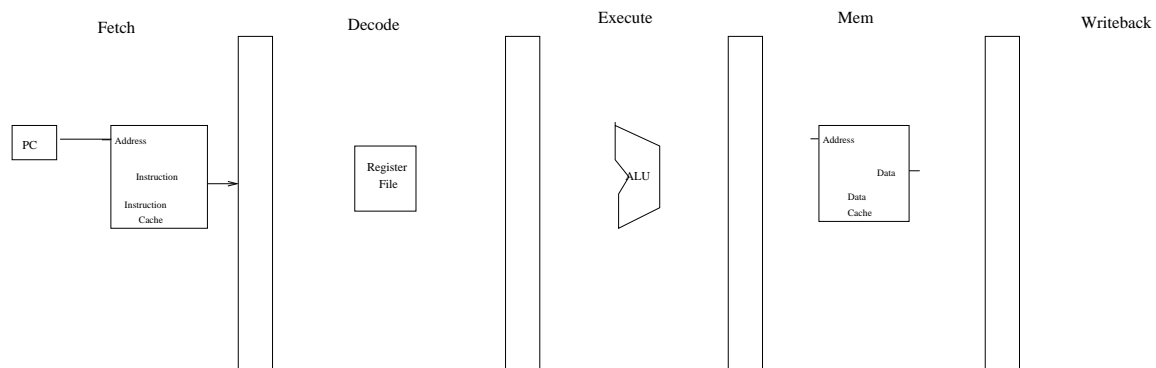
- (f) A competing processor design company builds an in-order superscalar processor with the same machine-width N as the width you found above in part(b). The machine has the same clock frequency as the VLIW processor. When you run the original MIPS program on this machine, you find that it executes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

Consider a design “Machine I” with five pipeline stages: fetch, decode, execute, memory, and writeback. Each stage takes 1 cycle. The instruction and data caches have 100% hit rates (i.e., there is never a stall for a cache miss). Branch directions and targets are resolved in the execute stage. The pipeline stalls when a branch is fetched, until the branch is resolved. Dependency check logic is implemented in the decode stage to detect flow dependences. The pipeline must stall on detection of a flow dependence, as it does not have any forwarding paths. In order to avoid stalls, we will modify Machine I to use fine-grained multithreading.

- (a) In the five stage pipeline of Machine I shown below, clearly show what blocks you would need to add in each stage of the pipeline, to implement fine-grained multithreading. You can replicate any of the blocks and add muxes. You don’t need to implement the mux control logic (although provide an intuitive name for the mux control signal, when applicable).



- (b) The machine’s designer first focuses on the branch stalls, and decides to use fine-grained multithreading to keep the pipeline busy no matter how many branch stalls occur. What is the minimum number of threads required to achieve this? Why?

- (c) The machine’s designer now decides to eliminate dependency-check logic and remove the need for flow-dependence stalls (while still avoiding branch stalls). How many threads are needed to ensure that no flow dependence ever occurs in the pipeline? Why?

A rival designer is impressed by the throughput improvements and the reduction in complexity that fine-grained multithreading (FGMT) brought to Machine I. This designer decides to implement FGMT on another machine, Machine II. Machine II is a pipelined machine with the following stages.

Fetch	1 stage
Decode	1 stage
Execute	8 stages (branch direction/target are resolved in the first execute stage)
Memory	2 stages
Writeback	1 stage

Assume everything else in Machine II is the same as in Machine I.

- (d) Is the number of threads required to eliminate branch-related stalls in Machine II the same as in Machine I? If yes, why? If no, how many threads are required?

- (e) What is the minimum CPI (i.e., maximum performance) of each thread in Machine II when this minimum number of threads is used?

- (f) Now consider flow-dependence stalls. Does Machine II require the same minimum number of threads as Machine I to avoid the need for flow-dependence stalls? If yes, why? If no, how many threads are required?

- (g) What is the minimum CPI of each thread when this number of threads (to cover flow-dependence stalls) is used?

- (h) After implementing fine grained multithreading, the designer of Machine II optimizes the design and compares the pipeline throughput of the original Machine II (without FGMT) and the modified Machine II (with FGMT) both machines operating at their maximum possible frequency, for several code sequences. On a particular sequence that has no flow dependences, the designer is surprised to see that the new Machine II (with FGMT) has lower overall throughput (number of instructions retired by the pipeline per second) than the old Machine II (with no FGMT). Why could this be? Explain concretely.

10 Hardware-Software Interlocking

Consider two pipelined machines, I and II:

Machine I implements interlocking in hardware. On detection of a flow dependence, it stalls the instruction in the decode stage of the pipeline (blocking fetch/decode of subsequent instructions) until all of the instruction's sources are available. Assume internal register file forwarding (an instruction writes into a register in the first half of a cycle and another instruction can access the same register in the next half of the cycle). No other data forwarding is implemented. However, there are three execute units with adders, and independent instructions can be executed in separate execution units and written back out-of-order. There is one write-back stage per execute unit, so an instruction can write-back as soon as it finishes execution.

Machine II 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 NOPs. Assume internal register file forwarding (an instruction writes into a register in the first half of a cycle and another instruction can access the same register in the next half of the cycle).

Both machines have the following *four pipeline stages* and *three adders*:

1. Fetch (one clock cycle)
2. Decode (one clock cycle)
3. Execute (ADD takes 3 clock cycles. Each ADD unit is not pipelined, but an instruction can be executed if an unused execute (ADD) unit is available.)
4. Write-back (one clock cycle). There is one write-back stage per execute (ADD) unit.

Consider the following 2 code segments:

Code segment A

```
ADD R5 ← R6, R7
ADD R3 ← R5, R4
ADD R6 ← R3, R8
ADD R9 ← R6, R3
```

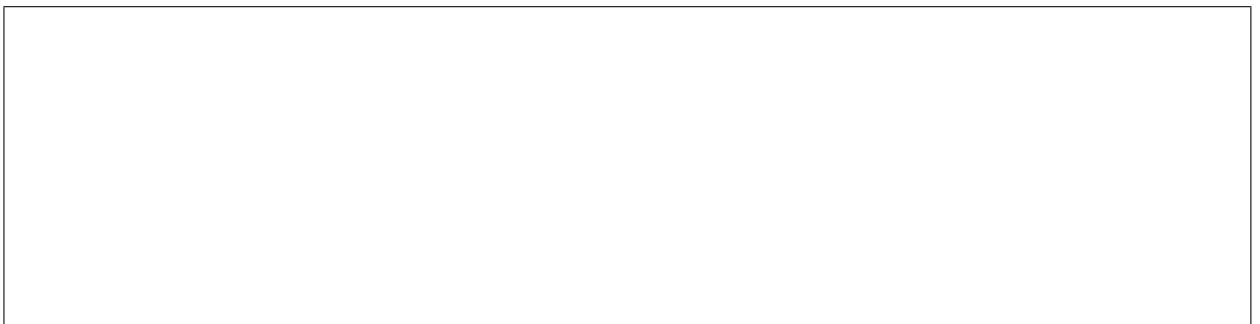
Code segment B

```
ADD R3 ← R1, R2
ADD R8 ← R9, R10
ADD R4 ← R5, R6
ADD R7 ← R1, R4
ADD R12 ← R8, R2
```

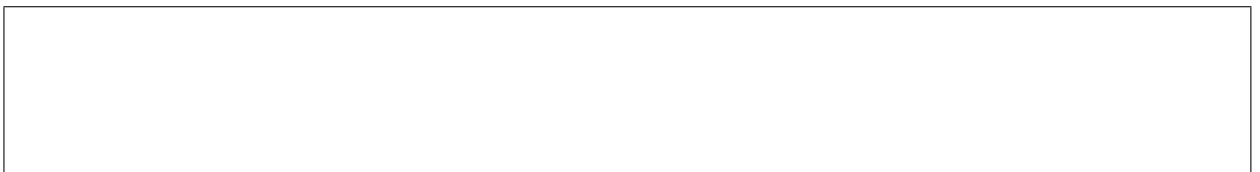
- (a) Calculate the number of cycles it takes to execute each of these two code segments on Machines I and Machine II.



- (b) Calculate the machine code size of each of these two code segments on machines I and II, assuming fixed-length ISA, where each instruction is encoded as 4 bytes.



- (c) Which machine takes a smaller number of cycles to execute each code segment I and II?



- (d) Does the machine that takes the smaller number of cycles for code segment A also take the smaller number of cycles than the other machine for code segment B ? Why or why not?

- (e) Would you say that the machine that provides a smaller number of cycles as compared to the other machine has higher performance (taking into account all components of the Iron Law of Performance)?

- (f) Which machine incurs lower code size for each code segment A and B ?

- (g) Does the same machine incur lower code sizes for both code segments A and B ? Why or why not?