CSE 520 Computer Architecture II
Term: Spring 2026
Lead Instructor: **Prof. Michel A. Kinsy**

Review Problem Set 2

Posted Feb. 22nd, 2026

http://ascslab.org/courses/cse520/index.html

General guidelines: Always state your assumptions and clearly explain your answers.

Part 1

Figure

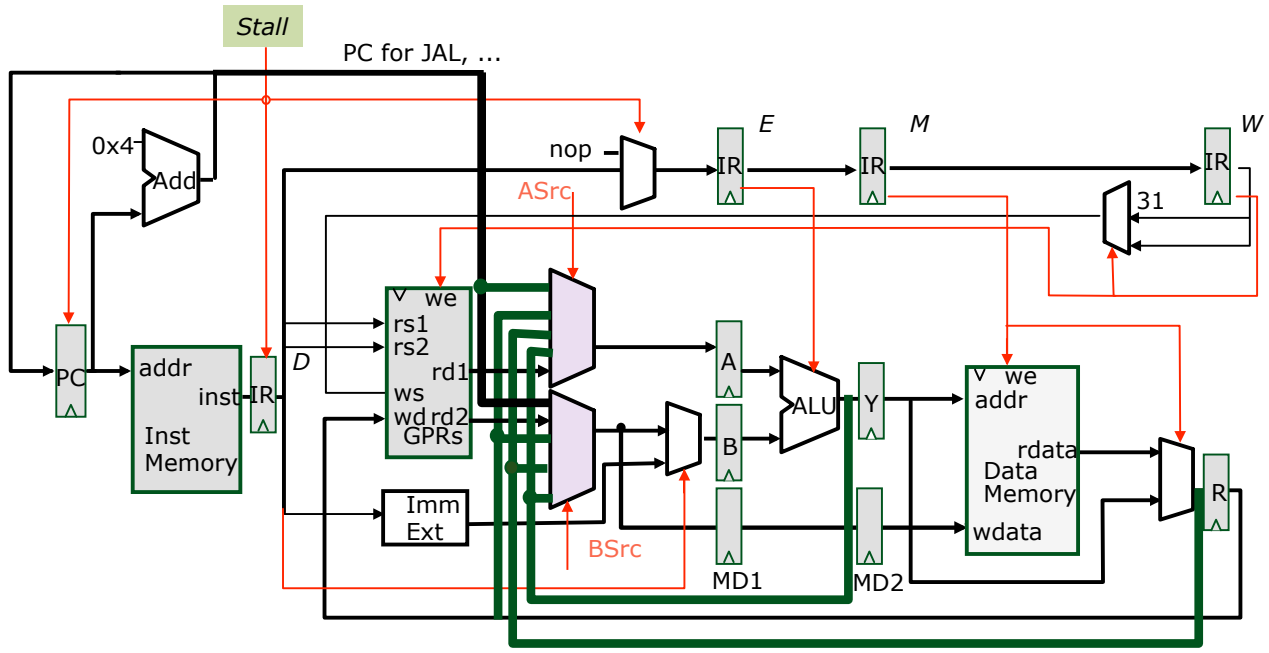


Figure 1: RISC-V 5-Stage Fully Bypassed Datapath

Problem 1

Figure 1 shows the 5-stage fully bypassed RISC-V processor that we saw in class. For this problem, only these RISC-V instructions are supported: ***ALU***, ***ALUi***, ***LW***, ***SW***, ***JAL***, ***JALR***, ***BEQ***, ***BGE***, ***BLT***, and ***BNE***.

Albert needs to compute the power function for small numbers. Realizing there is no multiply instruction in this RISC-V machine, he uses the following code to calculate the result when an unsigned number ***m*** is raised to the ***n***th power, where ***n*** is another unsigned number.

```
if (m == 0) {
   result = 0;
}else {
   result = 1;
   i = 0;

   while (i < n) {
      temp = result;
      j = 1;
      while (j < m) {
         result += temp;
         j++;
      }
      i++;
   }
}
```

The variables ***i***, ***j***, ***m***, ***n***, ***temp***, and ***result*** are unsigned 32-bit values.

Write the RISC-V assembly that implements Albert's code. This RISC-V machine does not have branch delay slots. Use ***x1*** for ***m***, ***x2*** for ***n***, and ***x3*** for result. At the end of your code, only ***x3*** must have the correct value. The values of all other registers do not have to be preserved.

How many RISC-V instructions are executed to calculate the power function? How many cycles does it take to calculate the power function?

m, n	***Instructions***	***Cycles***
0, 1		
1, 0		
2, 2		
3, 4		
M, N		

Problem 2

Using Figure 1, please answer the following questions:

2.a.

Does Albert still need to stall this pipeline? If so, explain why.

(1) Write down the correct equation for the stall condition:

(2) Give an example instruction sequence, which causes a stall:

2.b.

In lecture, we saw the bypass signal (ASrc) from EX stage to ID stage.

Write down the rest of bypass conditions:

Bypass $_{EX->ID}$ $\quad ASrc = (rs_D = ws_E).we\text{-}bypass_E.rel_D$ ***(Seen in lecture)***

Bypass $_{MEM->ID}$ $\quad ASrc =$

Bypass $_{WB->ID}$ $\quad ASrc =$

Please, indicate the priority of the signals; that is, if all bypass conditions are met, indicate which one has the highest and the lowest priorities.

Priority:

2.c.

While bypassing gives us a performance benefit, it may introduce extra logic in critical paths and may force us to lower the clock frequency. Suppose we can afford to have only one bypass in the datapath. How would you justify your choice? Argue in favor of one bypass path over another.

Problem 3 and 4 Figure

IF	ID	EX1	EX2/MEM	WB
Instruction fetch	Instruction decode and register reads	ALU1 execution and address calculation	ALU2 execution and memory access	Writeback to register file

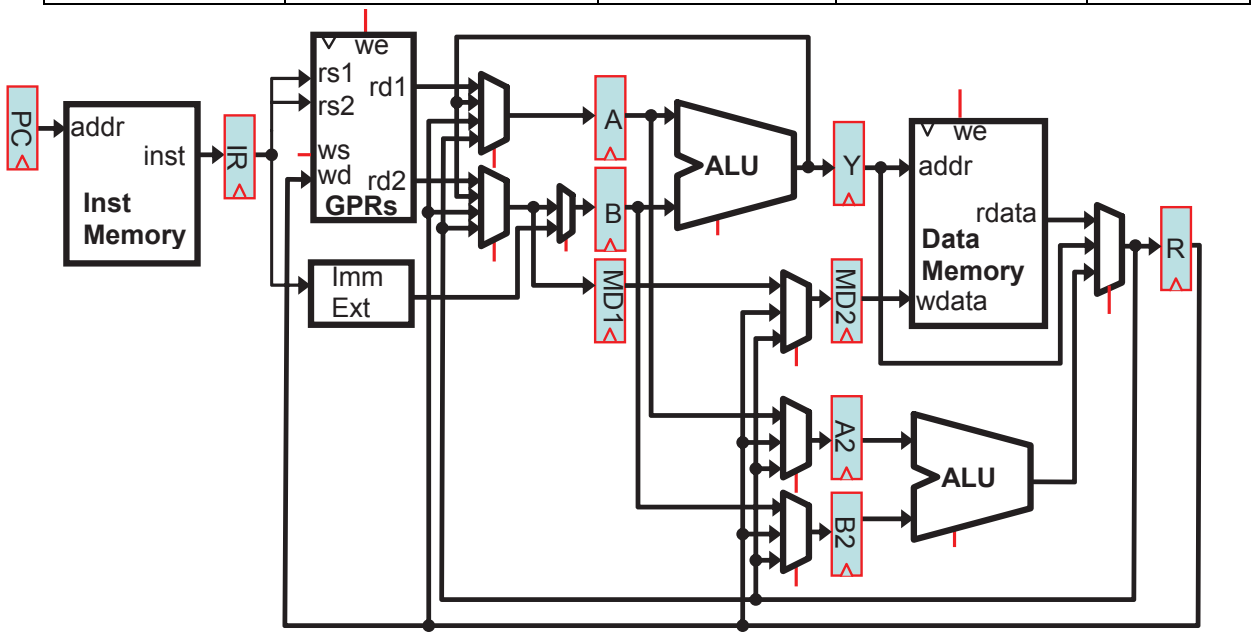


Figure 2: Dual ALU Pipelined

Albert decides to improve the 5-stage RISC-V architecture in Figure 1 by having a dual ALU Pipelined architecture. The Dual ALU Pipeline has two ALUs: ALU1 is in the 3rd pipeline stage (EX1) and ALU2 is in the 4th pipeline stage (EX2/MEM). A memory instruction always uses ALU1 to compute its address. An ALU instruction uses either ALU1 or ALU2, but never both. If an ALU instruction's operands are available (either from the register file or the bypass network) by the end of the ID stage, the instruction uses ALU1; otherwise, the instruction uses ALU2.

Assume that the control logic is optimized to stall only when necessary. You may ignore branch and jump instructions in this problem.

Problem 3

3.a.

Give an example sequence of RISC-V instructions (five or fewer instructions) that would cause a pipeline bubble in the original datapath, but not in the new datapath.

3.b.

Give an example sequence of RISC-V instructions (five or fewer instructions) that would cause a pipeline bubble in the new datapath, but not in the original datapath.

3.c.

Compare the advantages and disadvantages of the new datapath. Which one would you recommend? Justify your choice.

Problem 4

4.a.

For the following instruction sequence, indicate which ALU each add instruction uses. Assume that the pipeline is initially idle (for example, it has been executing nothing but ***nop*** instructions). Registers involved in inter-instruction dependencies are highlighted in bold for your convenience.

	ALU1 or ALU2
`add **x1**, x2, x3`	
`lw **x4**, 0(**x1**)`	
`add **x5**, **x4**, x6`	
`add x7, x5, x8`	
`add **x1**, x2, x3`	
`lw x4, 0(**x1**)`	
`add x5, **x1**, x6`	

4.b.

Fill in the equation for the control logic signal $alu2_{ID}$. This signal is computed during the ID stage. It should be true if the instruction will use ALU2, or false otherwise. Like other control logic signals, alu2 travels down the pipeline with an instruction as $alu2_{EX1}$ and $alu2_{EX2/MEM}$, you may use these signals in your equation if needed. In the equation, "+" means logical or, and "·" means logical and.

$$alu2_{ID} = (((OP_{ID} = ALU) + (OP_{ID} = ALUi)) \cdot ((rs_{ID} = ws_{EX1}) + (rt_{ID} = ws_{EX1}) \cdot re2_{ID}) \cdot (ws_{EX1} \neq 0) \cdot (\_\_\_\_\_))$$

4.c.

Indicate whether each of the following instruction sequences causes a stall in the pipeline. Consider each sequence separately and assume that the pipeline is initially idle (for example, it has been executing nothing but ***nop*** instructions). Registers involved in inter-instruction dependencies are highlighted in bold for your convenience.

	Stall? (Yes/No)
`add  **x1**, x2, x3` `lw  x4, 0(**x1**)`	

`lw  **x1**, 0(x2)`
`add  x3, **x1**, x4`
`lw  x5, 0(**x1**)`	

`lw  **x1**, 0(x2)` `lw  x3, 0(**x1**)`	
`lw  **x1**, 0(x2)` `sw  **x1**, 0(x3)`	

`lw  **x1**, 0(x2)`
`add  **x3**, **x1**, x4`
`sw  x5, 0(**x3**)`	

`lw  **x1**, 0(x2)` `add  x3, **x1**, x4`	

4.d.

Give the stall equation for the new pipeline. It should be optimized so that the pipeline only stalls when necessary to resolve data hazards. You may use the alu2 logic signals from 3.b. if needed.

$stall_{ID}$ =

Part 2

Problem 1

			Latencies beyond single cycle	
Loop:	LD	F2,0(RX)	Memory LD	+4
I0:	DIVD	F8,F2,F0	Memory SD	+1
I1:	MULTD	F2,F6,F2	Integer ADD, SUB	+0
I2:	LD	F4,0(Ry)	Branches	+1
I3:	ADDD	F4,F0,F4	ADDD	+1
I4:	ADDD	F10,F8,F2	MULTD	+5
I5:	ADDI	Rx,Rx,#8	DIVD	+12
I6:	ADDI	Ry,Ry,#8		
I7:	SD	F4,0(Ry)		
I8:	SUB	R20,R4,Rx		
I9:	BNZ	R20,Loop		

Figure 3: Instructions and Latencies

Question a: What would be the baseline performance (in cycles, per loop iteration) of the code sequence in Figure 1 if no new instruction's execution could be initiated until the previous instruction's execution had completed?

Note: Ignore front-end fetch and decode. Assume for now that execution does not stall for lack of the next instruction, but only one instruction/cycle can be issued. Assume the branch is taken, and that there is a one-cycle branch delay slot.

Question b: Think about what latency numbers really mean—they indicate the number of cycles a given function requires to produce its output, nothing more. If the overall pipeline stalls for the latency cycles of each functional unit, then you are at least guaranteed that any pair of back-to-back instructions (a "*producer*" followed by a "*consumer*") will execute correctly. But not all instruction pairs have a producer/consumer relationship. Sometimes two adjacent instructions have nothing to do with each other.

How many cycles would the loop body in the code sequence in Figure 1 require if the pipeline detected true data dependences and only stalled on those, rather than blindly stalling everything just because one functional unit is busy?
Show the code with <stall> inserted where necessary to accommodate stated latencies.

Hint: An instruction with latency +2 requires two <stall> cycles to be inserted into the code sequence. Think of it this way: A one-cycle instruction has latency 1 + 0, meaning zero extra wait states. So, latency 1 + 1 implies one stall cycle; latency ***1 + N*** has ***N*** extra stall cycles.

Question c: Consider a multiple-issue design. Suppose you have two execution pipelines, each capable of beginning execution of one instruction per cycle, and enough fetch/decode bandwidth in the front end so that it will not stall your execution. Assume results can be immediately forwarded from one execution unit to another, or to itself. Further assume that the only reason an execution pipeline would stall is to observe a true data dependency.

Now how many cycles does the loop require?

Question d: In the multiple-issue design of **Question c**, you may have recognized some subtle issues. Even though the two pipelines have the exact same instruction repertoire, they are neither identical nor inter-changeable, because there is an implicit ordering between them that must reflect the ordering of the instructions in the original program. If instruction ***N + 1*** begins execution in Execution Pipe 1 at the same time that instruction N begins in Pipe 0, and ***N + 1*** happens to require a shorter execution latency than ***N***, then ***N*** + 1 will complete before ***N*** (even though program ordering would have implied otherwise).

Give at least two reasons why that could be hazardous and will require special considerations in the microarchitecture. Give an example of two instructions from the code in Figure 1 that demonstrate this hazard.

Question e: Reorder the instructions to improve performance of the code in Figure 1. Assume the two-pipe machine in **Question c** and that the out-of-order completion issues of **Question d** have been dealt with successfully. Just worry about observing true data dependences and functional unit latencies for now.

How many cycles does your reordered code take?

Question f: Every cycle that does not initiate a new operation in a pipe is a lost opportunity, in the sense that your hardware is not living up to its potential.

In your reordered code from **Question e**, what fraction of all cycles, counting both pipes, were wasted (did not initiate a new op)?

Loop unrolling is one standard compiler technique for finding more parallelism in code, in order to minimize the lost opportunities for performance. Hand-unroll two iterations of the loop in your reordered code from **Question e**.

What speedup did you obtain?
For this question, just color the ***N + 1*** iteration's instructions green to distinguish them from the **Nth** iteration's instructions; if you were actually unrolling the loop, you would have to reassign registers to prevent collisions between the iterations.)

Problem 2

```
Loop: LD      F4,0(Rx)
I0:   MULTD   F2,F0,F2
I1:   DIVD    F8,F4,F2
I2:   LD      F4,0(Ry)
I3:   ADDD    F6,F0,F4
I4:   SUBD    F8,F8,F6
I5:   SD      F8,0(Ry)
```

a. Instructions

```
I0:   LD      T9,0(Rx)
I1:   MULTD   T10,F0,T9
...
```

b. Renaming Example

Figure 4: Renaming code

Question a: Computers spend most of their time in loops, so multiple loop iterations are great places to speculatively find more work to keep CPU resources busy. Nothing is ever easy, though; the compiler emitted only one copy of that loop's code, so even though multiple iterations are handling distinct data, they will appear to use the same registers. To keep multiple iterations' register usages from colliding, we rename their registers. Figure 2 shows example code that we would like our hardware to rename. A compiler could have simply unrolled the loop and used different registers to avoid conflicts, but if we expect our hard- ware to unroll the loop, it must also do the register renaming.

How? Assume your hardware has a pool of temporary registers (call them T registers, and assume that there are 64 of them, T0 through T63) that it can substitute for those registers designated by the compiler. This rename hardware is indexed by the *src* (source) register designation, and the value in the table is the T register of the last destination that targeted that register.

Think of these table values as producers, and the *src* registers are the consumers; it doesn't much matter where the producer puts its result as long as its consumers can find it. Consider the code sequence in Figure 2. Every time you see a destination register in the code, substitute the next available T, beginning with T9. Then update all the *src* registers accordingly, so that true data dependences are maintained. Show the resulting code.

Question b: The question above (**Question a**) explored simple register renaming: when the hardware register *renamer* sees a source register, it substitutes the destination T register of the last instruction to have targeted that source register. When the rename table sees a destination register, it substitutes the next available T for it, but superscalar designs need to handle multiple instructions per clock cycle at every stage in the machine, including the register renaming. A simple scalar processor would there- fore look up both *src* register mappings for each instruction and allocate a new *dest* mapping per clock cycle. Superscalar processors must be able to do that as well, but they must also ensure that any *dest-to-src* relationships between the two concurrent instructions are handled correctly. Consider the following sample code sequence:

```
I0:   SUBD    F1,F2,F3
I1:   ADDD    F4,F1,F2
I2:   MULTD   F6,F4,F1
I3:   DIVD    F0,F2,F6
```

Assume that we would like to simultaneously rename the first two instructions. Further assume that the next two available T registers to be used are known at the beginning of the clock cycle in which these two instructions are being renamed. Conceptually, what we want is for the first instruction to do its rename table lookups and then update the table per its destination's T register. Then the second instruction would do exactly the same thing, and any inter-instruction dependency would thereby be handled correctly. But there's not enough time to write that T register designation into the renaming table and then look it up again for the second instruction, all in the same clock cycle. That register substitution must instead be done live (in parallel with the register rename table update). Figure 3 shows a circuit diagram, using multiplexers and comparators, that will accomplish the necessary on-the-fly register renaming. Your task is to show the cycle-by-cycle state of the rename table for every instruction of the code above. Assume the table starts out with every entry equal to its index (T0 = 0; T1 = 1, …).

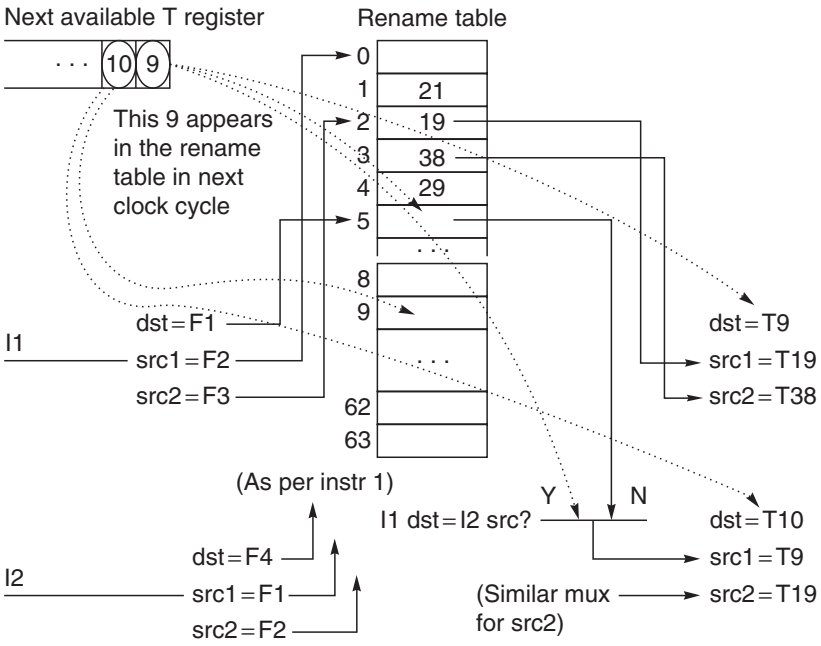


Figure 5: Rename table and on-the-fly renaming logic

Question c: If you ever get confused about what a register renamer has to do, go back to the assembly code you're executing, and ask yourself what has to happen for the right result to be obtained. For example, consider a three-way superscalar machine renaming these three instructions concurrently:

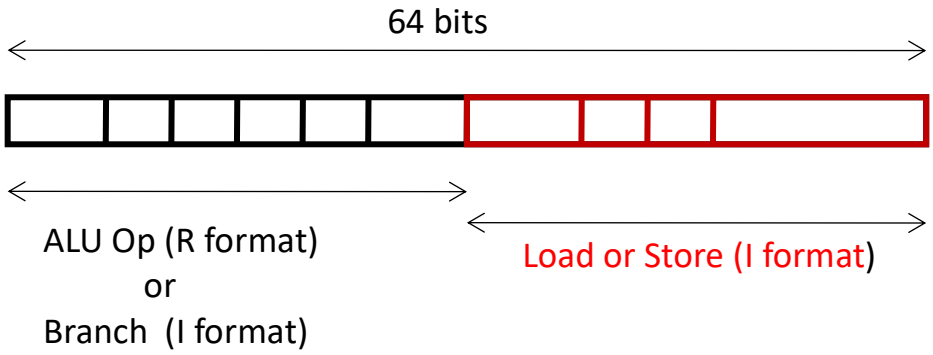
```
ADD  x1,  x1,  x1
ADD  x1,  x1,  x1
ADD  x1,  x1,  x1
```

If the value of `x1` starts out as 5, what should its value be when this sequence has executed?

Part 3

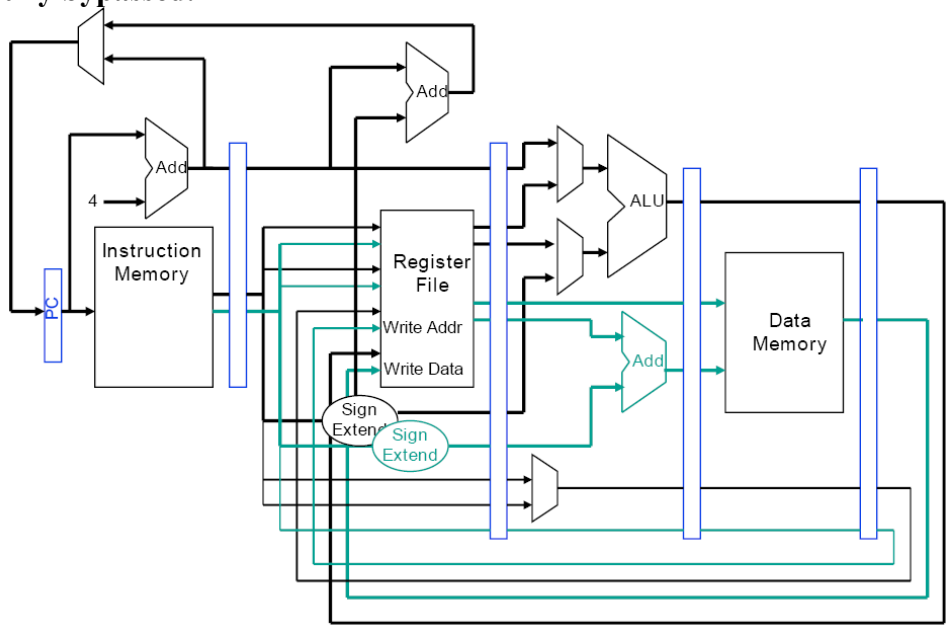
Problem 1

Albert Bitdiddle is a serious man. His latest processor is a single-issue RISC-V in-order VLIW processor with 2 operations per instruction:



So operations are always fetched, decoded, and issued in pairs, if one operation of the pair cannot be used for an instruction, it is replaced with a NOP.

Below is a diagram of Albert's VLIW machine. **Please note that no load-use allowed**. **But it is fully bypassed.**



To test his architecture, Albert runs small program:

C Code

```
int k = 513;
for(int i = 32, i > 0 ; i= i - 1)
{
    array [i] =  array [i] + k ;
}
```

RISC-V Assembly

```
# The array is x1 and k is stored in x2
loop: lw   x3, 0(x1);   # x3 is loaded with the array element
      add  x3, x3, x2;  # add k to the element
      sw   x3, 0(x1);   # store result
      addi x1, x1, -4;  # decrement pointer
      bne  x1, x0, loop;# loop if x1 != 0
```

Question a: Per Albert's architecture, what are the minimum latencies (cycles) for these operations?
Load (lw):
Add (add):

Question b: Albert wants to schedule his instructions so that each loop is executed in the least amount of cycles. Please help Ben by filling in the table below (show only one iteration of the loop):

Cycle	ALU or Branch	Memory Operation (Adder)
1	NOP	lw x3, 0(x1)
2		
3		
4		
5		

Question c: What is Albert's CPI (cycle per instruction) with this schedule?

Problem 2

Very long instruction word (VLIW) designers have a few basic choices to make regarding architectural rules for register use. Suppose a VLIW is designed with self-draining execution pipelines: once an operation is initiated, its results will appear in the destination register at most L cycles later (where L is the latency of the operation). There are never enough registers, so there is a temptation to wring maximum use out of the registers that exist. Consider the following code:

```
Loop: LW x4,0(x0)   | ADDI x11,x3,2
      LW x5,8(x1)   | ADDI x11,x0,2
      <stall>
      ADDI x10,x4,1|
      SW x7,0(x6)   | SW x9,8(x8)
      ADDI x2,x2,16
      SUB x4,x3,x2
      BNZ x4, Loop
```

If loads have a 1 + 2 cycle latency, unroll this loop once, and show how a VLIW capable of two loads and two adds per cycle can use the minimum number of registers, in the absence of any pipeline interruptions or stalls. Give an example of an event that, in the presence of self-draining pipelines, could disrupt this pipelining and yield wrong results.

Part 4

Problem 1

Assume a five-stage single-pipeline microarchitecture (fetch, decode, execute, memory, write-back) and the following code:

```
Loop: LW   x3,0(x0)
      LW   x1,0(x3)
      ADDI x1,x1,1
      SUB  x4,x3,x2
      SW   x1,0(x3)
      BNZ  x4, Loop
```

All ops are one cycle except LW and SW, which are 1 + 2 cycles, and branches, which are 1 + 1 cycles. There is no forwarding. Show the phases of each instruction per clock cycle for one iteration of the loop.

How many clock cycles per loop iteration are lost to branch overhead?

Assume a static branch predictor, capable of recognizing a backwards branch in the Decode stage. Now how many clock cycles are wasted on branch overhead?

Assume a dynamic branch predictor. How many cycles are lost on a correct prediction?

Problem 2

Question a: An (m,n) correlating branch predictor uses the behavior of the most recent m executed branches to choose from 2m predictors, each of which is an n- bit predictor. A two-level local predictor works in a similar fashion, but only keeps track of the past behavior of each individual branch to predict future behavior.

There is a design trade-off involved with such predictors: Correlating predictors require little memory for history which allows them to maintain 2-bit predictors for a large number of individual branches (reducing the probability of branch instructions reusing the same predictor), while local predictors require substantially more memory to keep history and are thus limited to tracking a relatively small number of branch instructions.

For this problem, consider a (1,2) correlating predictor that can track four branches (requiring 16 bits) versus a (1,2) local predictor that can track two branches using the same amount of memory. For the following branch outcomes, provide each prediction, the table entry used to make the prediction, any updates to the table as a result of the prediction, and the final misprediction rate of

each predictor. Assume that all branches up to this point have been taken. Initialize each predictor to the following:

Correlating predictor			
Entry	**Branch**	**Last outcome**	**Prediction**
0	0	T	T with one misprediction
1	0	NT	NT
2	1	T	NT
3	1	NT	T
4	2	T	T
5	2	NT	T
6	3	T	NT with one misprediction
7	3	NT	NT

Local predictor			
Entry	**Branch**	**Last 2 outcomes (right is most recent)**	**Prediction**
0	0	T,T	T with one misprediction
1	0	T,NT	NT
2	0	NT,T	NT
3	0	NT	T
4	1	T,T	T
5	1	T,NT	T with one misprediction
6	1	NT,T	NT
7	1	NT,NT	NT

Branch PC (word address)	Outcome
454	T
543	NT
777	NT
543	NT
777	NT
454	T
777	NT
454	T
543	T

Question b: Suppose we have a deeply pipelined processor, for which we implement a branch-target buffer for the conditional branches only. Assume that the misprediction penalty is always four cycles and the buffer miss penalty is always three cycles. Assume a 90% hit rate, 90% accuracy, and 15% branch frequency. How much faster is the processor with the branch-target buffer versus a processor that has a fixed two-cycle branch penalty? Assume a base clock cycle per instruction (CPI) without branch stalls of one.

Question c: Consider a branch-target buffer that has penalties of zero, two, and two clock cycles for correct conditional branch prediction, incorrect prediction, and a buffer miss, respectively. Consider a branch-target buffer design that distinguishes conditional and unconditional branches, storing the target address for a conditional branch and the target instruction for an unconditional branch. What is the penalty in clock cycles when an unconditional branch is found in the buffer?

Question d: Determine the improvement from branch folding for unconditional branches. Assume a 90% hit rate, an unconditional branch frequency of 5%, and a two-cycle penalty for a buffer miss. How much improvement is gained by this enhancement? How high must the hit rate be for this enhancement to provide a performance gain?