Homework # 1

1. Comparing CISC, RISC, and Stack ISAs

a) ***How many bytes in program?***

`xor eax, eax`	`jmp L1`	`inc [edx+eax*4]`	`inc eax`	`cmp eax, ecx`	`jl loop`	**TOTAL**
2 bytes	2 bytes	4 bytes	1 byte	2 bytes	2 bytes	**13 bytes**

How many bytes of instructions need to be fetched if size is 16?
2 bytes + 2 bytes + 16*(9 bytes) + 2 bytes + 2 bytes = **152 bytes**

Assuming 32-bit data values, how many bytes of data need to be loaded from the data memory?
Only `inc [edx+eax*4]` loads from the data memory and we load 16 times. The total number of bytes loaded is **64 bytes** since each load is 4 bytes long.

How many bytes need to be stored to the data memory?
Only `inc [edx+eax*4]` stores into the data memory and we store 16 times. The total number of bytes stored is **64 bytes** since each store is 4 bytes long.

b) Translate each of the x86 instructions into one or more TinyRV2 instructions.

Label	**x86 Instruction**	**Equivalent TinyRV2 Instruction Sequence**
	`xor eax, eax`	`xor x5, x5, x5`
	`jmp L1`	`auipc pc, #32 //bc that's how many instructions`

`loop:`	`inc [edx+eax*4]`	`lui x6, #4`
		`mul x6, x6, x5`
		`add x6, x6, x10`
		`lw x7, x6, #0`
		`addi x7, x7, #1`
		`sw x6, x7, #0`

	`inc eax`	`addi x5, x5, #1`
`L1:`	`cmp eax, ecx`	`slt x6, x5, x11`
	`jl loop`	`blt x5,x11, #-32 //bc that's how many instrs`

How many bytes are in the TinyRV2 program using your direct translation?
11 instructions * 4 bytes/instruction = **44 bytes**

How many bytes of instructions need to be fetched if size is 16?
4 bytes + 4 bytes + 16(9*4 bytes) + 4 bytes + 4 bytes = **592 bytes**

Assuming 32-bit data values, how many bytes of data need to be loaded from the data memory?
How many bytes need to be stored to the data memory?
Load and store happens once every iteration of the loop, so 16 times 4 bytes, **64 bytes** of memory data is loaded and stored.

c) Write an optimized TinyRV2 assembly program that implements the array_increment function.

```
void array_increment( int* aptr, int size ){
      int i;
      for ( i = 0; i < size; i++ )
            aptr[i] = aptr[i] + 1;
}
```

```
            lui x5, #0              // i = 0
loop :      slli x6, x5, #2         // x4, using logical left shift by 2
            add x6, x6, x10         // add the offset to the aptr
            lw x7, x6, #0           // load the val from aptr+4i
            addi x7, x7, #1         // increment val by 1
            sw x6, x7, #0           // store the val
            addi x5, x5, #1         // i++
            bne x5, x11, #-24        // if i != size, repeat
```

How many bytes are in your TinRV2 program?

8 instructions * 4 bytes/instruction = **32 bytes**

How many bytes of instructions need to be fetched if size is 16?

4 bytes + 16 loops (7 instructions/loop * 4 bytes/instruction) = **452 bytes**

Assuming 32-bit data values, how many bytes of data need to be loaded from the data memory? How many bytes need to be stored to the data memory?

Load and store happens once every iteration of the loop, so 16 times 4 bytes, **64 bytes** of memory data is loaded and stored.

d) Create a table like the one shown.

Contents of Stack On first Iteration	Stack Memory?	Label	Instruction	Length of Instruction
aptr; size	n		goto L1	5 bytes
aptr; size	n	loop:	swap	1 byte
size; aptr	y		pushi #4	2 bytes
size; aptr; #4	y		add	1 byte
size; aptr	y		dup	1 byte
size; aptr; aptr	y		dup	1 byte
size; aptr; aptr; aptr	y		pushm	1 byte
size; aptr; aptr; M[aptr]	y		pushi #1	2 bytes
size; aptr; aptr; M[aptr]; #1	y		add	1 byte
size; aptr; aptr; M[aptr]+1	y		popm	1 byte
size; aptr	n		swap	1 byte
aptr; size	y		pushi #1	2 bytes
aptr; size; #1	y		sub	1 byte
aptr; size	y	L1:	dup	1 byte
aptr; size; size	y		bgtz loop	5 bytes

How many bytes are in the stack-based program? How many bytes of instructions need to be fetched if size is 16?

Program is **26 bytes** long. If the size is 16, 5 bytes + 16*(21 bytes) **= 341 bytes** of instructions need to be fetched.

Assuming 32-bit data values, how many bytes of data need to be loaded from the data memory? How many bytes need to be stored to the data memory?

Load and store happens once every iteration of the loop, so 16 times 4 bytes, **64 bytes** of memory data is loaded and stored.

How many instructions will result in some kind of access to the special stack memory?

There are a total of 15 instructions on this program currently, and **12 instructions** require access to special stack memory.

If the microarchitecture used eight TOS registers, how many instructions would result in some kind of access to the special stack memory?

Currently the longest the stack length reaches is 5. If the microarchitecture used eight TOS registers, **none of the instructions** would require special stack access.

e) Comparison of ISAs

Given the results from the first four parts as a starting point, make a compelling argument for which ISA will result in the smallest static code size and fewest dynamically fetched instruction bytes on a broader selection of common programs. While you certainly should summarize the results from the first three parts, your analysis should not be purely based on these results. Consider how these initial results can be extrapolated to other programs. Do not factor in memory traffic or performance; your argument should be purely based on static code size and number of bytes of instructions that need to be fetched.

I believe x86 ISA will result in the smallest static size and also the fewest dynamically fetched instruction bytes given a broader range of programs. Relatively speaking x86 is a much richer ISA in terms of types and flexibility of instructions. TinyRV2 has 34 general purpose instructions, and they must be tailored to handle more complex actions. The x86 can have a single instruction to do something TinyRV2 uses 5 instructions to complete. Thus, since the instruction set is much more versatile in x86, it is much easier to optimize the code byte-size. Stack based ISA is probably the least suited for a compactness of instructions fetched. Although the example provided above may suggest that stack based ISA may be more compact than TinyRV2, but in the grand scheme of programs, we are not always incrementing an array by one. The reason for TinyRV2 unpromising numbers in the example above is the bytes used to express each instruction, which is always 4 bytes. This will always play against the TinyRV2 when the program is iterative/loop-y.

2. Microcoded TinyRV1 Processor

a) Implementing Conditional Move `movz rd, rs1, rs2` **`if (R[rs2] == 0) R[rd] ← R[rs1]`**

State	Pseudo Control Signals	Bus Enables					Register Enables						Mux		Func		RF		mreq		
		pc	ig	alu	rf	rd	pc	ir	a	b	c	wd	b	c	ig	alu	sel	wen	val	op	next
MZ0	A ← RF[x0]	0	0	0	1	0	0	0	1	0	0	0	x	x	x	x	x0	0	0	x	n
MZ1	B ← RF[rs2]	0	0	0	1	0	0	0	0	1	0	0	b	x	x	x	rs2	0	0	x	n
MZ2	C ← A == B	0	0	1	0	0	0	0	0	0	1	0	x	b	x	cmp	x	0	0	x	n
MZ3	B ← RF[rs1]	0	0	0	1	0	0	0	0	1	0	0	b	x	x	x	rs1	0	0	x	n
MZ4	RF[rd] ← A +? B; goto F0	0	0	1	0	0	0	0	0	0	0	0	x	x	x	+?	rd	0	0	x	f

b) Implementing Memory-Memory Increment Instruction

`inc rd, rs1, imm` — `addr ← R[rd] + (R[rs1] × imm[3:0]); M[addr] ← M[addr] + 1`

State	Pseudo Control Signals	Bus Enables					Register Enables						Mux		Func		RF		mreq		
		pc	ig	alu	rf	rd	pc	ir	a	b	c	wd	b	c	ig	alu	sel	wen	val	op	next
MM0	A ← RF[x0]	0	0	0	1	0	0	0	1	0	0	0	x	x	x	x	x0	0	0	x	n
MM1	B ← RF[rs1]	0	0	0	1	0	0	0	0	1	0	0	b	x	x	x	rs1	0	0	x	n
MM2	C ← sext(imm_s)	0	1	0	0	0	0	0	0	0	1	0	x	b	s	x	x	0	0	x	n
MM3	C ← C>>8	0	0	0	0	0	0	0	0	0	1	0	x	s	x	x	x	0	0	x	n
MM4	A ← A +? B; B ← B<<1; C ← C>>1;	0	0	1	0	0	0	0	1	1	1	0	s	s	x	+?	x	0	0	x	n
MM5	A ← A +? B; B ← B<<1; C ← C>>1;	0	0	1	0	0	0	0	1	1	1	0	s	s	x	+?	x	0	0	x	n
MM6	A ← A +? B; B ← B<<1; C ← C>>1;	0	0	1	0	0	0	0	1	1	1	0	s	s	x	+?	x	0	0	x	n
MM7	A ← A +? B; B ← B<<1; C ← C>>1;	0	0	1	0	0	0	0	1	1	1	0	s	s	x	+?	x	0	0	x	n
MM8	A ← A +? B	0	0	1	0	0	0	0	0	1	0	0	x	x	x	+?	x	0	0	x	n
MM9	B ← RF[rd]	0	0	0	1	0	0	0	0	1	0	0	x	x	x	x	rd	0	0	x	n
MM10	memreq.addr ← A + B; RF[rd] ← A + B	0	0	1	1	0	0	0	0	0	0	0	x	x	x	+	rd	1	1	r	n
MM11	A ← RD	0	0	0	0	1	0	0	1	0	0	0	x	x	x	x	x	0	0	x	n
MM13	WB ← A + 1	0	0	1	0	0	0	0	0	0	0	1	x	x	x	+1	x	0	0	w	n
MM14	memreq.addr ← RF[rd]; goto F0	0	0	0	1	0	0	0	0	0	0	0	x	x	x	x	rd	0	0	x	f

3. Multiplier Microarchitecture

	Microarchitecture	Num Trans	Cycle Time	Transaction Latency		Transaction Throughput		Total Execution Time	
		(#)	(τ)	(cyc)	(τ)	(trans/cyc)	(cyc/trans)	(cycles)	(τ)
	1-cycle	60	62	1	62	1	1	60	3720
Part 3.a	Iterative	60	20	4	80	0.25	4	240	4800
Part 3.b	2-cycle unpipelined	60	32	2	64	0.5	2	120	3840
Part 3.b	2-cycle pipelined	60	32	2	64	~1	~1	61	1952
Part 3.c	4-cycle unpipelined	60	17	4	68	0.25	4	240	4080
Part 3.c	4-cycle pipelined	60	17	4	68	~1	~1	63	1071
Part 3.d	Var-Lat pipelined	60	20	1-5	7-87	0.5	2	120	2400

a) Iterative Multiplier

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
mul 0xdeadbeef, 0xf	x0	x0	x0	x0													
mul 0xf5fe4fbc, 0x7					x0	x0	x0	x0									
mul 0x0a01b044, 0x3									x0	x0	x0	x0					
mul 0xdeadbeef, 0xf													x0	x0	x0	x0	

b) Two Cycle Microarchitecture

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
mul 0xdeadbeef, 0xf	x0	x1															
mul 0xf5fe4fbc, 0x7			x0	x1													
mul 0x0a01b044, 0x3					x0	x1											
mul 0xdeadbeef, 0xf							x0	x1									

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
mul 0xdeadbeef, 0xf	x0	x1															
mul 0xf5fe4fbc, 0x7		x0	x1														
mul 0x0a01b044, 0x3			x0	x1													
mul 0xdeadbeef, 0xf				x0	x1												

c) Four Cycle Microarchitecture

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
mul 0xdeadbeef, 0xf	x0	x1	x2	x3													
mul 0xf5fe4fbc, 0x7					x0	x1	x2	x3									
mul 0x0a01b044, 0x3									x0	x1	x2	x3					
mul 0xdeadbeef, 0xf													x0	x1	x2	x3	

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
mul 0xdeadbeef, 0xf	x0	x1	x2	x3													
mul 0xf5fe4fbc, 0x7		x0	x1	x2	x3												
mul 0x0a01b044, 0x3			x0	x1	x2	x3											
mul 0xdeadbeef, 0xf				x0	x1	x2	x3										

d) Variable Latency Microarchitecture

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
mul 0xdeadbeef, 0xf	y	x0	x1	x2	x3												
mul 0xf5fe4fbc, 0x7		y	y	x1	x2	x3											
mul 0x0a01b044, 0x3				y	y	x2	x3										
mul 0xdeadbeef, 0xf					y	x0	x1	x2	x3								

e) Which microarchitecture has the highest performance? In a few paragraphs, explain some of the trade-offs in terms of area and performance between these microarchitectures.Would we ever want to consider a multiplier with many more stages (e.g., a 20-cycle pipelined microarchitecture)? How does the fixed-latency pipelined multiplier compared to the variable-latency pipelined multi-plier? Discuss when we would want to choose fixed-latency over variable-latency, and when we would want to choose variable-latency over fixed-latency. How would this trade-off change if one multiply transaction needs to wait for the result of an earlier transaction before starting? Make sure you generalize your conclusions so that they are valid over many different transaction sequences, not just the specific sequence studied in this problem.

Based on the table of entries of performance specification of all microarchitectures, the highest performing was the 4 stage pipelined microarchitecture. This is because it had the least execution time for 60 transactions. Pipelining the 4 stages allowed us to achieve approximately 1 transaction per cycle for large transaction counts. For each microarchitecture having multi-stage architecture helped with the performance, but each came at a cost of increasing active components like registers, multiplexers, and logic units. Since we only multiplied by a 4-bit number having a 4 stage pipelined architecture proved to be super beneficial. This would not hold for larger bit numbers. The performance would decrease as we added stages and increased cycles per transaction. For large numbers having a variable latency microarchitecture would probably be best suited, since it would allow for shorter cycles whenever possible. This would optimize the transaction execution time. Fixed latency is very useful for scenarios where one of the numbers are limited to a certain number of bits. In that scenario we can design a multiplier that has optimal number of stages and datapath. In scenarios of one transaction needing the calculated value of an earlier transaction, all architectures would suffer in performance. Variable latency however would still allow the flexibility of starting the next transaction earliest time possible since transaction does not have to take all of the stages to complete. Overall, there are always trade-offs when designing anything, but for our case we can be sure that pipelining can be generally considered a major performance optimizer.

4. Two-Cycle Pipelined Integer ALU and Multiplier

a) Draw a pipeline diagram similar to the one shown in Figure 8 that shows which instructions have to stall in which stages.

Dynamic Cycle Transaction	1	2	3	4	5	6	7	8	9	10	11	12	13	14
bne x1, x0, done	F	D	X0	X1	M	W								
lw x5, 0(x2)		F	D	X0	X1	M	W							
lw x6, 0(x3)			F	D	X0	X1	M	W						
add x7, x5, x6				F	D	D	D	X0	X1	M	W			
addi x8, x4, 4					F	F	F	D	X0	X1	M	W		
sw x7, 0(x8)								F	D	D	X0	X1	M	W

b) Derive the new stall signal for the six-stage pipeline with the datapath and associated by passing shown in Figure 9.

```
ostall_load_use_X0_rs1_D =
      val_D && rs1_en_D && val_X0 && rf_wen_X0 &&
      (inst_rs1_D == rf_waddr_X0) && (rf_waddr_X0 != 0)
      && (op_X0 == LW)

ostall_load_use_X1_rs1_D =
      val_D && rs1_en_D && val_X1 && rf_wen_X1 &&
      (inst_rs1_D == rf_waddr_X1) && (rf_waddr_X1 != 0)
      && (op_X1 == LW)

ostall_load_use_X0_rs2_D =
      val_D && rs2_en_D && val_X0 && rf_wen_X0 &&
      (inst_rs2_D == rf_waddr_X0) && (rf_waddr_X0 != 0)
      && (op_X0 == LW)

ostall_load_use_X1_rs2_D =
      val_D && rs2_en_D && val_X1 && rf_wen_X1 &&
      (inst_rs2_D == rf_waddr_X1) && (rf_waddr_X1 != 0)
      && (op_X1 == LW)

ostall_D =
      val_D &&
      ( ostall_load_use_X0_rs1_D || ostall_load_use_X1_rs2_D ||
        ostall_load_use_X0_rs1_D || ostall_load_use_X1_rs2_D )
```

c) Draw a pipeline diagram similar to the one shown in Figure 8 that shows which instructions have to be squashed in which stages.

Dynamic Cycle Transaction	1	2	3	4	5	6	7	8	9	10	11	12
`bne x1, x0, done`	F	D	X0	X1	M	W						
`lw x5, 0(x2)`		F	D	X0	-	-	-					
`lw x6, 0(x3)`			F	D	-	-	-	-				
`add x7, x5, x6`				F	-	-	-	-	-			
`addi x8, x4, 4`												
`sw x7, 0(x8)`												
`...`												
`addi x10, x9, 1`					F	D	X0	X1	M	W		

Draw a pipeline diagram similar to the one shown in Figure 8 except with a `jal` ***instruction in place of the*** `bne` ***instruction. Your diagram should clearly show which instructions have to be squashed in which stages.***

Dynamic Cycle Transaction	1	2	3	4	5	6	7	8	9	10	11	12
`jal x1, done`	F	D	X0	X1	M	W						
`lw x5, 0(x2)`		F	-	-	-	-	-					
`lw x6, 0(x3)`												
`add x7, x5, x6`												
`addi x8, x4, 4`												
`sw x7, 0(x8)`												
`...`												
`addi x10, x9, 1`			F	D	X0	X1	M	W				