ECE 4750 Computer Architecture, Fall 2015

Problem Set #1

http://www.csl.cornell.edu/courses/ece4750
School of Electrical and Computer Engineering
Cornell University

revision: 2015-09-22-22-05

The problem sets are intended to provide you an opportunity to put the concepts you have been learning in lecture into practice. We encourage you to discuss the information and concepts covered in lecture with other students in the course; one of the best ways to learn new and challenging material is through substantive discussion with your peers. You can give "consulting" help to or receive "consulting" help from other students in the course to help clarify your understanding of the problems, but under no circumstances should you directly copy solutions from another student. If you give or receive "consulting" help you should explicitly include mention of this in your problem set. Each student must turn in his or her own solutions to the problems, and these solutions are expected to accurately reflect your understanding of the material. Please see the course syllabus if you have any questions concerning the collaboration policy for the course.

Problem sets must be submitted electronically in PDF format through the online CMS system. You will need to upload each problem individually as a separate PDF. *No other format or method of submission is allowed!* Problem sets are due at 11:59pm on the date listed on the course webpage. You can continue to resubmit your files as many times as you would like up until the deadline, so please feel free to upload early and often. No late submissions will be accepted and no extensions will be granted except for a family or medical emergency. *If you miss the deadline, then your assignment will not count as submitted, and you will receive a zero!*

As an exception to this rule, each student has three slip days that may be used to secure 24-hour extensions on problem sets. The maximum extension for any given assignment is 48 hours.

Students are strongly encouraged to type their solutions; if necessary students can scan handwritten diagrams and combine these diagrams with typed solutions. Please use a reasonably large font size, especially in tables. If students absolutely must write up their entire solutions by hand, then they must scan their final version into a legible PDF. Illegible scans will not be graded. A digital photograph will almost certainly not be legible. Please bold, circle, or otherwise highlight your specific answer where appropriate. All pages must be 8.5"$\times$11" using a portrait orientation. If you cannot print your problem set using standard printer settings, then we also will not be able to print your problem set. If we cannot print your problem set, then we will not be able to grade your hard work. *Please try and include your full name and NetID on every page. Number the pages. We do not want to loose any pages.*

List of Problems

Problem 1. Comparing CISC, RISC, and Stack ISAs

In this problem, your task is to compare three different ISAs: x86 is an extended accumulator, CISC architecture with variable-length instructions; MIPS32 is a load-store, RISC architecture with fixed-length instructions; and we will also look at a simple stack-based ISA with variable-length instructions.

Part 1.A x86 CISC ISA

Consider the following C code which takes an array pointer (`aptr`) and the number of elements (`size`) as inputs, and then adds one to each of the elements in the array.

```
1 void array_increment( int* aptr, int size )
2 {
3   int i;
4   for ( i = 0; i < size; i++ )
5     aptr[i] = aptr[i] + 1;
6 }
```

Using `gcc` and `objdump` on an x86 processor, we determine that the loop in the above C function compiles to the following assembly code. Feel free to try compiling this yourself. This assembly fragment is just for the loop; it excludes the code required for managing the stack and for returning from the function. On entry to this code, register `%edx` contains `aptr` and register `%ecx` contains `size`.

```
1       xor %eax, %eax
2       jmp L1
3 loop: inc (%edx, %eax, 4)
4       inc %eax
5 L1:   cmp %ecx, %eax
6       jl  loop
```

Spend some time understanding how the assembly code implements the C code. The meanings and instruction lengths of the instructions used above are given in Figure 1. A register specifier uses a `r` prefix, a memory address uses a `m` prefix, a register value is denoted as R[`specifier`] and a memory value is denoted as M[`address`].

Notice that there are two versions of the `inc` instruction: a register-register version and a memory-memory version with a more sophisticated addressing mode. The `jl` instruction implements a conditional jump by using the `SF` and `OF` condition flags. These flags are similar to the floating point condition flags in the MIPS32 ISA. In x86, the condition flags are set by the instruction preceding the jump based on the result of the computation. Some instructions, such as the `cmp` instruction, perform a computation and set the condition flags, but do not return any result. The `OF` condition flag indicates overflow and is set if the result exceeds the positive or negative limit of the number range. The `SF` condition flag indicates the sign of the result and is set if the result is negative (less than zero). Thus `jl` instruction implements a jump if less than operation by checking to see if `SF` $\neq$ `OF`.

How many bytes are in the x86 program? How many bytes of instructions need to be fetched if `size` is 10? 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? Show your work.

Instruction	Operation	Length
`cmp rs0, rs1`	*temp* ← R[rs1] - R[rs0] SF ← sign bit of *temp* OF ← overflow of *temp*	2 bytes
`inc rt`	R[rt] ← R[rt] + 1	1 byte
`inc (ma,mb,imm)`	*temp* ← R[ma] + (R[mb] × imm) M[*temp*] ← M[*temp*] + 1	3 bytes
`jl label`	if (SF ≠ OF) jump to the address specified by label	2 bytes
`jmp label`	jump to address specified by label	2 bytes
`xor rt, rs`	R[rt] ← R[rt] ⊕ R[rs]	3 bytes

Figure 1: x86 ISA Subset

Part 1.B MIPS32 RISC ISA

Write an optimized MIPS32 assembly program that implements the `array_increment` function from the previous section. In lecture, we discussed how we will often assume that the length of arrays is always greater than one. To be enable a fair comparison for this problem, we will not make this assumption. Your assembly code will need to verify that the size is greater than zero before starting to work on the input array. On entry to the MIPS32 code, assume register `r4` contains `aptr` and register `r5` contains `size`. If necessary, use `r12`, `r13`, and `r14` for temporaries. You do not need to worry about the instructions required to return from the function (i.e., the `jr` instruction). You should not use pseudo-instructions. For example, do not use `li` or `la` since these pseudo-instructions can turn in a variable number of real instructions based on the context. We want to better understand the cycle-level execution of the code, and thus we need to focus on the real instructions that need to be executed. **Remember that unlike PARC, the MIPS32 instruction set includes a single-instruction branch delay slot.**

Note that there are more efficient ways than simply translating each individual x86 instruction directly into MIPS32 instructions. Try to optimize your code so that it minimizes register usage, static instructions, and/or the number of instructions fetched. You can assume the microarchitecture is fully bypassed. Obvious optimization approaches to consider are filling branch delay slots, using software scheduling to avoid load-use stalls, and more efficiently managing the loop control and array addressing. Remember that on entry to the MIPS32 code, register `r4` contains `aptr` and register `r5` contains `size`. The arguments to the `array_increment` function are passed by value, which means that the values in `r4` and `r5` do not need to be preserved; you are free to overwrite these values. **Your solution should contain commented assembly code and an explanation of your optimizations.**

How many bytes are in your MIPS32 program? How many bytes of instructions need to be fetched if `size` is 10? *You do not need to count instructions which are fetched but then later squashed!* **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? Show your work.**

Instruction	Operation	Length
`add`	pop *v0*; pop *v1*; push *v1* + *v0*	1 byte
`bgtz label`	pop v; if $v > 0$, jump to address specified by `label`; else, continue with next instruction	5 bytes
`dup`	pop v; push v; push v	1 byte
`goto label`	jump to address specified by `label`	5 bytes
`pushi imm`	push sext(`imm`)	2 byte
`pushm`	pop *addr*; push M[*addr*]	1 byte
`popm`	pop v; pop *addr*; M[*addr*] $\leftarrow v$	1 byte
`sub`	pop *v0*; pop *v1*; push *v1* - *v0*	1 byte
`swap`	pop *v0*; pop *v1*; push *v0*; push *v1*	1 byte

Figure 2: Simple Stack-Based ISA

Contents of Stack on First Iteration	Access Stack Memory?	Label	Instruction
aptr; size			`goto L1`
aptr; size		`loop:`	`...`
			`add rest of instrs here`
			`...`
aptr; size	Y	`L1:`	`dup`
aptr; size; size	Y		`bgtz loop`

Figure 3: `array_increment` **Loop Implemented with Stack-Based ISA**

Part 1.C Simple Stack-Based ISA

In a stack-based architecture, all instructions operate on an architecturally visible hardware stack. Some number of items at the top of the stack are kept in fast hardware registers, while the rest of the conceptually infinite stack is kept in a special portion of memory. Stack-based architectures were popular in the 1960's with the Burroughs B5000 computer serving as a classic example. This kind of architecture is convenient for expression evaluation, subroutine calls, recursion, and compiling for some important high-level languages such as ALGOL. Stack-based architectures fell out of favor in the late 1970's, but there has a been some recent interest owing to the stack-based nature of Java's intermediate bytecode representation. Figure 2 defines a simple stack-based ISA that we will use in this part. The semantics of each instruction are defined in terms of push and pop operations: pop v means pop the top of the stack and store it in variable v, and push v means push the variable v onto the stack. Note that v is not part of the architecture nor the microarchitecture – it is simply a notational construct to help define the instruction semantics. A memory value is denoted as M[*address*].

All values are 32-bit. In our simple stack-based ISA, only the `pushm` and `popm` instructions access memory; all other instructions remove zero, one, or two operands from the stack and replace them

with the result (if there is one). The `swap` instruction is special since it essentially reads two values from the top of the stack and writes two values onto the top of the stack in one instruction. The opcodes are encoded in a single byte. Notice that in this ISA, `pushi` only handles an 8-bit immediate that is sign extended to 32 bits when written to the top of the stack. The instruction and memory addresses are four bytes.

The microarchitecture assumed for this problem implements the top of the stack with two 32-bit registers and the remainder of the stack is stored in a special stack memory. We call these two 32-bit registers the top-of-stack (TOS) registers (the Burroughs B5000 also had two TOS registers). If there are two or more items on the stack and the program pushes another item on the stack, then the microarchitecture will take care of writing one item from the TOS registers to the special memory. If there are more than two items on the stack and the program pops one item from the stack, then the microarchitecture will take care of reading one item from the special memory into the TOS registers. To be more precise, assume an instruction pops n items ($0 < n$) at the beginning of its execution and pushes m items at the end of its execution. For our simple stack-based instruction set, $0 \leq n \leq 2$ and $0 \leq m \leq 2$ for all instructions. If $n < m$ and the final stack size is greater than two, the microarchitecture will need to write some number of items from the TOS registers to the special stack memory. If $n > m$ and the initial stack size is greater than two, the microarchitecture will need to read some number of items from the special stack memory into the TOS registers. If $n = m$, then there is no need to access the special stack memory regardless of the current stack size; in this case, the pushes and pops necessary to implement the instruction can simply be done by accessing the TOS registers. In other words, the `pushm`, `swap`, and `goto` instructions never need to access the special stack memory since $n = m$, while all other instructions *may* access the special stack memory depending on the current stack size.

Translate the `array_increment` loop from Part 1.A to our simple stack-based ISA. On entry to the code, assume that the top of the stack contains `size` and the second entry in the stack contains the `aptr`. To show your translation, **create a table like the one shown in Figure 3.** Explicitly track what is in the stack before the execution of each instruction, and note on which instructions the microarchitecture will need to perform some number of access to the stack memory. To get you started, a portion of the translation is already shown in Figure 3.

How many bytes are in the stack-based program? How many bytes of instructions need to be fetched if `size` is 10? 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? How many instructions will result in some kind of access to the 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? Show your work.

Part 1.D Comparison of ISAs

Given the results from the first three parts as a starting point, **make a compelling argument for which ISA will result in 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.*

B/C Mux Select Encoding	b	select mux input from bus
	s	select mux input from shifter
Immediate Adjustment Unit (IAU) Functions	si	sext(IR[15:0])
	ts	IR[25:0] << 2
	sis	sext(IR[15:0]) << 2
Arithmetic/Logic Unit (ALU) Functions	cpa	copy A
	cpb	copy B
	+1	A + 1
	+4	A + 4
	+	A + B
	+?	C[0] ? A + B : copy A
	cmp	A == B
	jt	{ A[31:28], B[27:0] }
Register File Select Encoding	r0	select `r0`
	r31	select `r31`
	rs	select register based on `rs` field in IR
	rt	select register based on `rt` field in IR
	rd	select register based on `rd` field in IR
Memory Request Op Encoding	r	read memory request
	w	write memory request
Next State Encoding	n	goto next state by incrementing µPC by one
	d	dispatch to instruction sequence based on opcode
	f	goto state F0
	b	goto state F0 if A == B

Table 1: Control Signal Encodings

Problem 2. Microcoded PARCv1 Processor

Consider the PARCv1 FSM processor with a microcoded control unit described in lecture. In this problem, we explore adding two new instructions by using the same datapath and just adding new microcode sequences to the control store.

Your solutions should be elegant and efficient; minimize the number of new states needed. Table 1 shows the encoding you should use for all of the control signal fields that are not just 0 or 1. Note that we have added three new operations to our ALU compared to lecture: copy the A input to the output of the ALU, copy the B input to the output of the ALU, and increment the A input by one. *You cannot add new datapath components or modify the datapath components beyond these three changes.* If you use any new pseudo-control-signal syntax please clearly explain what this syntax means. When filling in microcode, use don't cares (marked with an x or –) for fields where it is safe to use don't cares. Study the processor described in lecture well, and make sure all of your microinstructions are legal. Please comment your code clearly. Your code should exhibit "clean" behavior and not modify any architectural registers in the course of the execution. Finally, make sure that each new macroinstruction fetches the next macroinstruction with a microjump to F0.

		Bus Enables					Register Enables						Mux		Func		RF		mreq		
State	**Pseudo Control Sigs**	**pc**	**iau**	**alu**	**rf**	**rd**	**pc**	**ir**	**a**	**b**	**c**	**wd**	**b**	**c**	**iau**	**alu**	**sel**	**wen**	**val**	**op**	**next**
F0:	mreq_addr ← PC; A ← PC	1	0	0	0	0	0	0	1	0	0	0	–	–	–	–	–	0	1	r	n
F1:	IR ← RD	0	0	0	0	1	0	1	0	0	0	0	–	–	–	–	–	0	0	–	n
F2:	PC ← A+4; A ←A+4	0	0	1	0	0	1	0	1	0	0	0	–	–	–	+4	–	0	0	–	d
	...																				

Figure 4: Microcode Table

Part 2.A Implementing Conditional Move

For this part, you are to add the PARCv3 conditional move instruction `movn`. This instruction only copies the source register to the destination register if a second source register is not zero. The assembly format and semantics for this instruction are as follows:

```
movn rd, rs, rt
```

if (R[rt] != 0) R[rd] ← R[rs]

See the PARC ISA document for more information. **Create a table like the one shown in Figure 4 to represent the contents of the control store, and fill in the state, pseudo-control-signal syntax, actual control signals, and next state fields for a microinstruction fragment that implements the** `movn` **instruction.** The fetch fragment has already been provided for you.

Part 2.B Implementing Memory-Memory Increment Instruction

For this part, you are to implement a new PARC instruction that is similar in spirit to the x86 memory-memory `inc` instruction discussed in Part 1.A. This instruction will use a relatively complicated addressing mode to read a value from memory, increment that value by one, and then write the value back to the same location in memory. The assembly format and semantics for the new PARC instruction are as follows:

```
inc rt, rs, imm
```

addr ← R[rt] + (R[rs] × imm[3:0]); M[*addr*] ← M[*addr*] + 1

Note that *addr* is simply a temporary to simplify the instruction semantics. It is not architectural state. Note that this instruction only uses the least significant four bits of the immediate when calculating the effective address. **Create a table like the one in Figure 4 to represent the contents of the control store, and fill in the state, pseudo-control-signal syntax, actual control signals, and next state fields for a microinstruction fragment that implements the** `inc` **instruction.**

Problem 3. Multiplier Microarchitecture

In this problem, we consider several different implementations of an unsigned two-input integer multiplier capable of multiplying a 32-bit operand by a 4-bit operand to produce a truncated 32-bit result. Figure 5 abstractly illustrates the datapaths for five microarchitectures: a single-cycle microarchitecture; a four-cycle iterative microarchitecture; a two-cycle microarchitecture that can be either unpipelined or pipelined; a four-cycle microarchitecture that again can be either unpipelined or pipelined; and a variable-latency pipelined microarchitecture. The "iron-law" of processor performance is applicable to far more than just processors. We will be using the following generalized form to examine the performance of each of these microarchitectures:

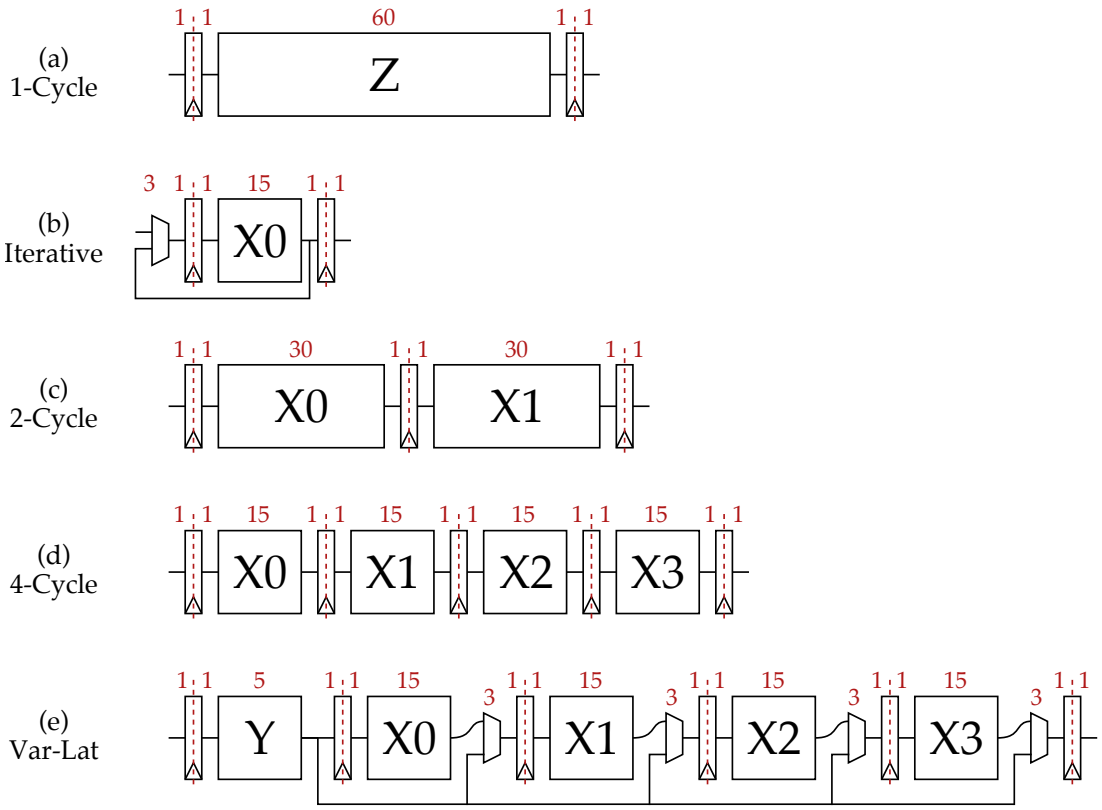

Figure 5: Various Multiplier Microarchitectures

$$\frac{\text{Time}}{\text{Transaction Sequence}} = \frac{\text{Transactions}}{\text{Transaction Sequence}} \times \frac{\text{Cycles}}{\text{Transaction}} \times \frac{\text{Time}}{\text{Cycle}}$$

In our multiplier, a *transaction* is simply a multiplication request. **Create a table similar to the one shown in Figure 6.** Feel free to use a spreadsheet and copy the final table into your submission. In each part, we will study one of these microarchitectures, and our goal is to gradually fill in this table. The table already includes the results for the single-cycle multiplier microarchitecture.

The *cycle time* (i.e., clock period) is measured in normalized gate delays, where $1\,\tau$ is the delay of a single inverter driving four identical inverters. Rough approximations of the delay of each component are shown on the datapath diagrams in Figure 5. Note that a register has a clock-to-data delay of $1\,\tau$ (i.e., the combinational delay from the rising clock edge to when the output data is stable), and a setup time of $1\,\tau$ (i.e., how much time before the clock edge the input data must be stable).

The *transaction latency* is the number of cycles we need to execute a single transaction going through the multiplier in isolation. For the variable latency multiplier, the transaction latency depends on the data so you should include a range in the table that captures the best and worst case transaction latency.

The *transaction throughput* can be measured as either the average number of transactions we process per cycle or the average number of cycles a transaction occupies the microarchitecture. One is just

		Num Trans	Cycle Time	Transaction Latency		Transaction Throughput		Total Execution Time	
	Microarchitecture	**(#)**	**(τ)**	**(cyc)**	**(τ)**	**(trans/cyc)**	**(cyc/trans)**	**(cycles)**	**(τ)**
(a)	1-Cycle	60	62	1	62	1	1	60	3720
(b)	Iterative	60							
(c)	2-Cycle Unpiplined	60							
(c)	2-Cycle Pipelined	60							
(d)	4-Cycle Unpiplined	60							
(d)	4-Cycle Pipelined	60							
(e)	Var-Lat Pipelined	60							

Figure 6: Evaluation of Various Multiplier Microarchitectures

the inverse of the other. Latency and throughput are two very different (although related) concepts; please make sure you clearly understand these two concepts. When calculating the transaction throughput for this problem, we will assume that our multiplier is processing a sequence of 60 transactions. The 60 transactions are formed by repeating the following three transactions 20 times.

```
1 mul 0xdeadbeef, 0xf
2 mul 0xf5fe4fbc, 0x7
3 mul 0x0a01b044, 0x3
```

So the 60 transaction sequence will look like this:

```
1 mul 0xdeadbeef, 0xf
2 mul 0xf5fe4fbc, 0x7
3 mul 0x0a01b044, 0x3
4 mul 0xdeadbeef, 0xf
5 mul 0xf5fe4fbc, 0x7
6 mul 0x0a01b044, 0x3
7 mul 0xdeadbeef, 0xf
8 mul 0xf5fe4fbc, 0x7
9 ...
```

The *total execution time* is the total time (in units of τ) to execute the sequence of 60 transactions.

Part 3.A Iterative Microarchitecture

Consider the iterative multiplier microarchitecture shown in Figure 5(b). This microarchitecture is very similar to the one you implemented in the first lab assignment, except that we only need to iterate for four cycles. This is because we are multiplying a 32-bit operand by a just a 4-bit operand. Assume that we have optimized the implementation so that we do not need any additional cycles to handle the val/rdy interface. The multiplier only handles unsigned numbers so we don't need to worry about sign/unsign logic. **In other words, we can complete each transaction in exactly four cycles, and we are ready to start the next transaction after exactly four cycles.**

Draw a transaction vs. time diagram illustrating the execution of the first four multiplication transactions on this microarchitecture. A transaction vs. time diagram is like a pipeline diagram,

but of course this microarchitecture is not pipelined. There should be one column per cycle and one row per transaction. Use the symbol Z to indicate on which cycle each transaction is using the iterative multiplier. Use your transaction vs. time diagram to fill in the appropriate row of the table in Figure 6.

Part 3.B Two-Cycle Microarchitecture

Consider the two-cycle multiplier microarchitecture shown in Figure 5(c). In this microarchitecture, we use the same basic approach as the iterative multiplier, but we do the computation in space instead of time by unrolling the shift and add operations. We do two of the shift and addition operations in a single cycle. We will consider an unpipelined variant where only a single transaction can be using any part of the multiplier at once, and a pipelined variant where there can be two different transactions using the multiplier at the same time (i.e., one in the X0 stage and one in the X1 stage).

Draw two transaction vs. time diagrams illustrating the execution of the first four multiplication transactions on both the unpipelined and pipelined variants. Use the symbols X0 and X1 to indicate on which cycle each transaction is using that part of the multiplier. Use your transaction vs. time diagram to fill in the appropriate rows of the table in Figure 6.

Part 3.C Four-Cycle Microarchitecture

Consider the four-cycle multiplier microarchitecture shown in Figure 5(d). In this microarchitecture, we use the same basic approach as the two-cycle multiplier, but with a single shift and addition operation per cycle. We will consider an unpipelined variant where only a single transaction can be using any part of the multiplier at once, and a pipelined variant where there can be four different transactions using the multiplier at the same time (i.e., different transactions in X0, X1, X2, and X3).

Draw two transaction vs. time diagrams illustrating the execution of the first four multiplication transactions on both the unpipelined and pipelined variants. Use the symbols X0, X1, X2, and X3 to indicate on which cycle each transaction is using that part of the multiplier. Use your transaction vs. time diagram to fill in the appropriate rows of the table in Figure 6.

Part 3.D Variable-Latency Microarchitecture

Consider the variable-latency multiplier microarchitecture shown in Figure 5(e). This microarchitecture exploits the fact that when some of the bits in the four-bit operand are zero, we don't actually have to do any work. We add a new stage at the beginning of the pipeline (denoted with the Y symbol) that is responsible for determining the bit position of the most significant one in the four-bit operand. This control information is used to set the mux select in a later pipeline stage so as to skip over some of the early stages in the pipeline. For example, if the four-bit operand is two (0b0010), then the transaction would go through stage Y, skip over stages X0 and X1, and go through stages X2 and X3. Note that this requires an extra mux at the end of each X stage, and we will need to carefully handle the structural hazard caused by multiple stages writing the same register. Again, if the four-bit operand is two, then we need to avoid two transactions writing the pipeline register at the end of the X1 stage at the same time. **Assume that the multiplier stalls in the Y stage whenever it detects that letting the current transaction go down the pipeline would cause a structural hazard.**

Draw a transaction vs. time diagram illustrating the execution of the first four multiplication transactions on the pipelined variable-latency microarchitecture. Use the symbols Y, X0, X1, X2, and X3 to indicate on which cycle each transaction is using that part of the multiplier. Look at the four-bit operand in each of the four multiplication transactions to determine how many stages of computation are actually required. Ensure that two transactions are never in the same stage at the same time. Use your transaction vs. time diagram to fill in the appropriate row of the table in Figure 6.

Part 3.E Comparison of Microarchitectures

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 compare to the variable-latency pipelined multiplier? 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.*

Problem 4. Two-Cycle Pipelined Integer ALU and Multiplier

Assume in a given emerging technology, the logic delay is significantly slower than the memory delay as compared to the standard CMOS technology used in modern processors. In this situation, the execute stage of the standard five-stage pipeline might be significantly longer than the other stages, and as a consequence, we might want to split this stage into two pipeline stages. In this problem we will be investigating the implications of using a two-cycle pipelined integer ALU and multiplier. Our new pipelined PARCv1 processor will have the following six stages:

- F – instruction fetch
- D – decode and read registers
- X0 – first half of the ALU and multiply operation
- X1 – second half of the ALU and multiply operation
- M – data memory read/write
- W – write registers

Figure 7 illustrates the new six-stage datapath. Spend some time studying this datapath to understand how the two-cycle pipelined integer ALU and multiplier afffect the structural, data, and control hazards in the pipeline. Assume that only those bypass paths shown in the diagram are present. More specifically, notice that this datapath does not allow bypassing between back-to-back dependent integer ALU operations without requiring some kind of stall. Also notice the store data can only bypassed into the end of the D stage, and that conditional branches are resolved by the end of the X1 stage.

Figure 8 shows a simple code sequence and illustrates the read-after-write (RAW) data dependencies present given the current pipeline assuming the branch is not taken. An arrow indicates a microarchitectural RAW dependency, and since some of these arrows point backwards they create data hazards. Please note the backwards arrow representing the RAW hazard between the X1 stage of the `addiu` instruction and the D stage of the `sw` instruction. The result of the `addiu` is not ready until the *end* of the X1 stage, but the `sw` instruction needs the store data at the *end* of the D stage so

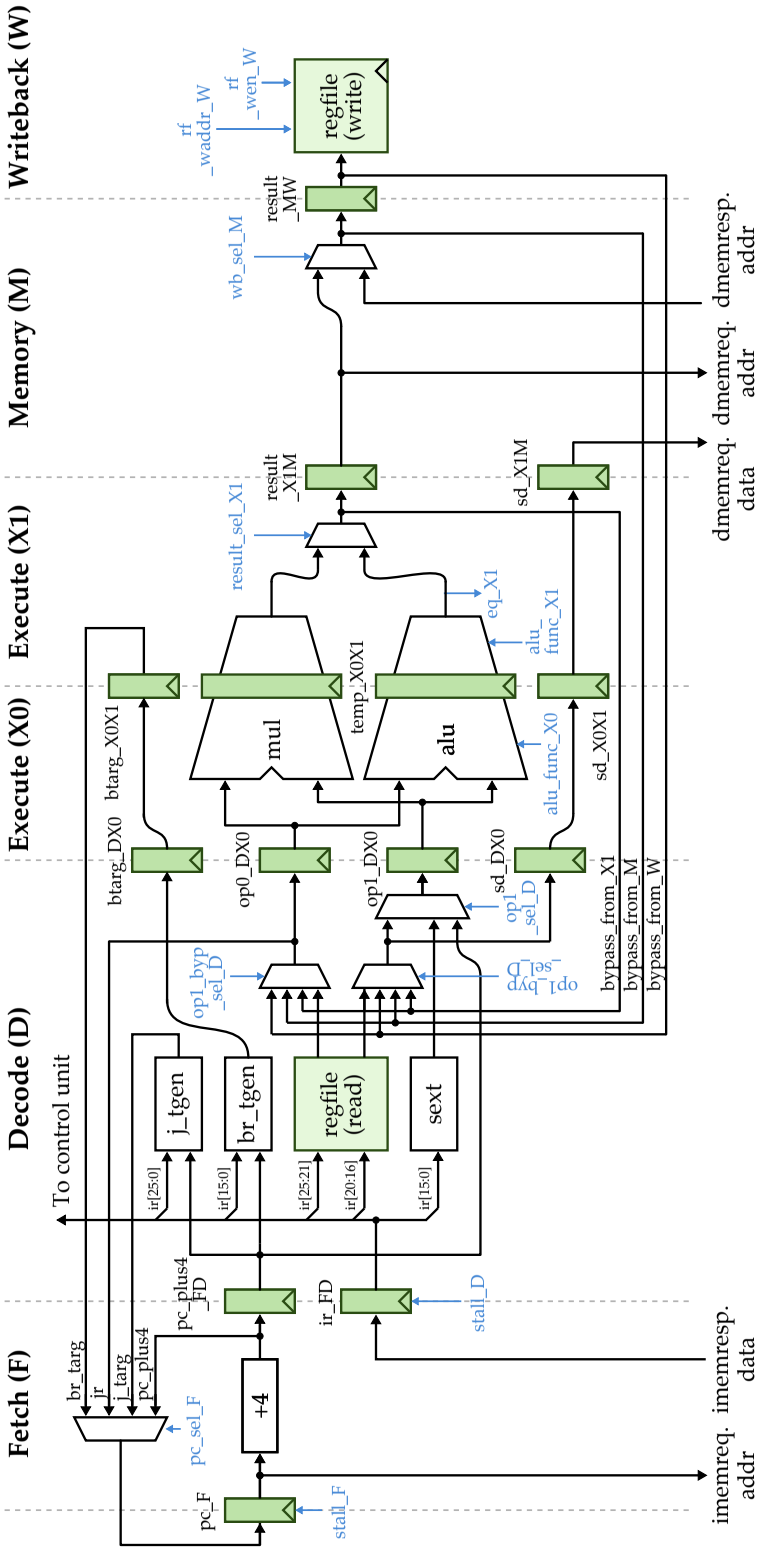

Figure 7: Datapath for Six-Stage Pipelined PARCv1 Processor

Static Instr Sequence

```
1   bne   r1, r0, done
2   lw    r5, 0(r2)
3   lw    r6, 0(r3)
4   addu  r7, r5, r6
5   addiu r8, r4, 4
6   sw    r7, 0(r8)
7   ...
8 done:
9   addiu r10, r9, 1
```

Dynamic Transaction		Cycle 0	1	2	3	4	5	6	7	8	9	10
1	`bne r1, r0, done`	F	D	X0	X1	M	W					
2	`lw r5, 0(r2)`		F	D	X0	X1	M	W				
3	`lw r6, 0(r3)`			F	D	X0	X1	M	W			
4	`addu r7, r5, r6`				F	D	X0	X1	M	W		
5	`addiu r8, r4, 4`					F	D	X0	X1	M	W	
6	`sw r7, 0(r8)`						F	D	X0	X1	M	W

Figure 8: Execution of Six-Stage Pipelined MIPS32 Processor

that it can bypass it into the operand pipeline register between the D and X0 stages – spend some time studying what bypass paths are available in Figure 7. Technically, we could add a special bypass path just for the store data from the end of the X1 stage to the end of the X0 stage (and also from the end of the M stage into the end X0 stage and the end of the M stage into the end of the X1 stage). Since this bypass path can only be used in very specific cases, it is not included in our design (and thus not shown in Figure 7). In this microarchitecture, the store base addresss *and* store data must both be ready at the the end of the D stage so they can be muxed into the operand pipeline reigster between the D and X0 stages. Spend some more time understanding the example in Figure 8.

Part 4.A Control and Data Hazard Latencies

The *jump resolution latency* and *branch resolution latency* are the number of cycles we need to delay the fetch of the next instruction in order to avoid any kind of control hazard (assuming we do not use speculation). Note that with a single-issue processor, we always delay the fetch of the next instruction by one cycle anyways. The *ALU-use delay latency* is the number of cycles we need to delay the execution of an instruction that uses the result of an integer ALU instruction to avoid a data hazard. The *load-use delay latency* is the number of cycles we need to delay an instruction that uses the result of a load to avoid a data hazard. For the standard five-stage pipeline, the jump resolution latency is two cycles, the branch resolution latency is three cycles if the branch condition is checked in the execute stage, the branch resolution latency is two cycles if the branch condition is checked in the decode stage, the ALU-use delay latency is one cycle, and the load-use delay latency is two cycles. Since a single-issue processor always delays the fetch of the next instruction by one cycle, we do not need to stall even though the ALU-use delay latency is one cycle.

What is the jump resolution latency, branch resolution latency, the ALU-use delay latency, and the load-use delay latency for the new six-stage pipeline shown in Figure 7?

Part 4.B Resolving Data Hazards with Software Scheduling

Assume we have a fully bypassed datapath, but we expose those stalls that are unavoidable (even with bypassing) in the instruction set architecture. Reschedule the code shown in Figure 8 by moving instructions and/or adding `nop` instructions to avoid any data hazards that are not handled by bypassing. Try to minimize the execution time of the instruction sequence. Your rescheduled code should be functionally equivalent to the original code, but should have no stalls! **Show the new static code sequence and describe your optimizations. Draw a pipeline diagram similar to**

the one shown in Figure 8. Either draw the microarchitectural RAW dependencies as in Figure 8 or list them in the form *Instruction X's D stage depends on Instruction Y's M stage*. Verify that none of the RAW dependency arrows point backward in time. Assume the branch is not taken, and that the microarchitecture always speculatively executes the not taken path.

Part 4.C Resolving Data Hazards with Stalling

Assume we do not reschedule the code in software, but instead we simply use hardware stalling to correctly prevent RAW hazards that we cannot avoid with bypassing. Note that we are using a combination of stalling and bypassing in this problem. **Draw a pipeline diagram similar to the one shown in Figure 8 that shows which instructions have to stall in which stages.** Show stalls by simply repeating the pipeline stage character (e.g., D) for multiple consecutive cycles. Your pipeline diagram should be for the original unscheduled code in Figure 8. Either draw the microarchitectural RAW dependencies as in Figure 8 or list them in the form *Instruction X's D stage depends on Instruction Y's M stage*. Verify that none of the RAW dependency arrows point backward in time. Assume the branch is not taken, and that the microarchitecture always speculatively executes the not taken path.

Part 4.D New Stall Signal

The stall signal for a fully bypassed five-stage pipeline was discussed in class and is included below for you reference:

```
ostall_load_use_X_rs_D
  =    val_D && rs_en_D && val_X && rf_wen_X
    && (inst_rs_D == rf_waddr_X) && (rf_waddr_X != 0)
    && (op_X == LW)

ostall_load_use_X_rt_D
  =    val_D && rt_en_D && val_X && rf_wen_X
    && (inst_rt_D == rf_waddr_X) && (rf_waddr_X != 0)
    && (op_X == LW)

stall_D
  =    val_D && !squash_D
    && ( ostall_load_use_X_rs_D || ostall_load_use_X_rt_D )
```

Each signal has a suffix indicating which pipeline stage the signal originates from. `rs_en` and `rt_en` are true for instructions that read from either the `rs` or `rt` registers respectively; `rs` and `rt` are the actual read register specifiers for both read ports; `rf_waddr` is the write destination register; and `op` is the opcode. Understand this stall signal thoroughly before attempting to complete this part.

Derive the new stall signal for the six-stage pipeline with the datapath and associated bypassing shown in Figure 7. This stall signal essentially implements the stalls that you identified in the previous part. You should use a similar syntax as the original stall signal above. Define new `ostall` hazard signals as necessary.

Part 4.E Resolving Control Hazards with Scheduling and Speculation

Assume the branch is taken. **Draw a pipeline diagram similar to the one shown in Figure 8 that shows which instructions have to be squashed in which stages.** Use a dash symbol (–) to indicate pipeline bubbles caused by squashing instructions.

Now assume that we modify the PARCv1 ISA to include a single-instruction branch delay slot after every jump and conditional branch. Remember that a single-instruction branch delay slot specifies that the instruction after a branch is always executed regardless of whether or not the branch is taken or not taken. Assume that the static code sequence remains the same as in in Figure 8. This means the first `lw` instruction is now in the branch delay slot, and it will always be executed regardless of whether or no thte branch is taken or not taken. Assume the programmer has already ensured that scheduling the `lw` in the branch delay slot perserves correct execution of the program. **Draw a pipeline diagram similar to the one shown in Figure 8 assuming we now have a single-instruction branch delay slot. Show which instructions have to be squashed in which stages.**