ECEn 424
The Dynamic Scheduling Lab

1 Introduction

In this lab, you will write code that runs on a simulated Y86-64 processor that uses dynamic scheduling. Normally we want our code to run as fast as possible, but for this lab you are to write (pathological) code consisting of a fixed number of instructions that takes as *long* to execute as possible. Your code must also achieve certain conditions within the processor, such as causing the front-end of the pipeline to stall. You will find that you have to understand the processor fairly well to complete this assignment.

2 Logistics

You are to complete this lab on your own. Any clarifications or revisions to the assignment will be posted on the web page for the lab.

3 Handout Instructions

The handouts for this lab are this document and the initial lab5.ys file that can be downloaded from the lab webpage. Place your copy of lab5.ys in a new (protected) directory from which to work. The only handin for this lab will be your lab5.ys file after you complete your changes and verify that it works as expected.

4 The Target Architecture

For this lab, two instructions have been added to the Y86-64 instruction set: an integer multiply and an integer divide. Here are examples of their usage in a Y86-64 assembly language program.

```
mulq %rsi,%rax        # %rax <-- %rax * %rsi
divq %rcx,%rbx        # %rbx <-- %rbx / %rcx
```

The new instructions are consistent with the other Y86-64 arithmetic instructions: two program registers are specified, both of which contain input operands, and one of which will be overwritten with the numerical

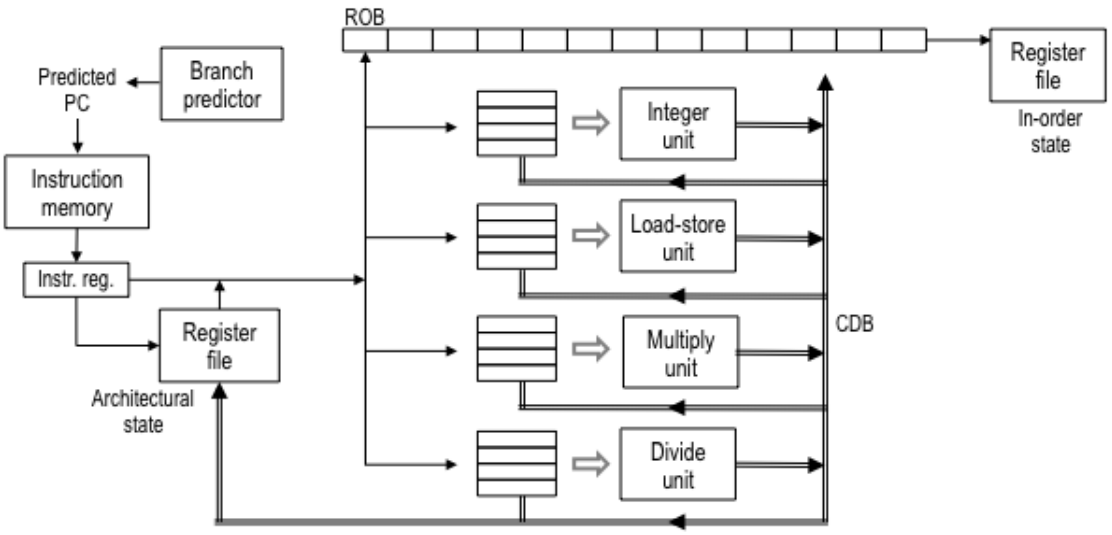

Figure 1: **The target Y86-64 hardware with dynamic scheduling.** The design is based on Tomasulo's algorithm, with four distinct functional units.

result. Consistent with x86-64 conventions, the first instruction above will multiply the 64-bit value in %rax by the 64-bit value in %rsi and put the 64-bit result in %rax. Similarly, the divq instruction will take the 64-bit value in %rbx, divide it by the 64-bit value in %rcx, and put the 64-bit result in %rbx.

The target hardware configuration is shown in Figure 1. This is similar to the dynamically scheduled CPU described in the class slides, but without functional units for floating-point operations.

In this system, there are 4 reservation stations associated with each functional unit, and the reorder buffer (ROB) has 20 entries, not all of which are shown in the figure.

The two separate register files shown in the figure are useful in supporting recovery from mispredicted branches and precise exceptions, but the operational details are not important for this lab. (For the curious, the architectural state defines register values that subsequent instructions will use, and the in-order state effectively provides a means to reset the architectural state in the event of a mispredicted branch or exception.)

Operational details

In this processor instructions pass through six stages, as described below. Instructions that are past Issue but that have not yet reached Retire are said to be *in-flight*.

- **Fetch** An instruction is retrieved from memory at the predicted PC and placed into the instruction register. Normally instructions spend one cycle in Fetch; the stage is extended if the preceding instruction stalls in Issue. (For this lab, we're ignoring the details of both branch prediction and misprediction

recovery. You'll be writing straightline code with no branches, calls or returns.)

- **Issue** The instruction moves from the instruction register to a reservation station *and* to an entry in the ROB. The instruction stalls if a reservation station is not available, or if a ROB entry is not available. During this stage, the register file is accessed to obtain the operands for the instruction. If a required value is not in the register file (because it is being produced by an earlier instruction that has not finished execution), the tag associated with that value is copied. The instruction and its operands (or operand tags) are placed in the reservation station. Every instruction spends at least one full cycle in Issue.

- **Wait for Data**. An instruction remains in this stage until all required operands are available. (Each was obtained from the register file, or it will appear on the CDB and can be identified by the tag that accompanies it.) When all operands are available, the instruction can move to a functional unit on the next cycle. If two or more instructions become ready and try to start using the same functional unit on the same cycle, the oldest will succeed and the other will wait in its reservation station for at least one more cycle. Each instruction spends at least one full cycle in this stage.

- **Execute**. The four functional units are all fully pipelined. Instructions using the integer unit (halt, nop, rrmovq, irmovq, addq, subq, andq, xorq, jxx, cmovxx) require a single Execute cycle. Instructions using the load-store unit (mrmovq, rmmovq, popq, pushq, call, ret) require 2 Execute cycles. On the first cycle, the effective address for the memory reference is computed. On the second cycle, loads access memory. Stores are not allowed to modify memory until they retire from the ROB. (The mechanism used to accomplish this is described in the next section.) The mulq instruction requires 5 Execute cycles (and uses the multiply unit), and divq requires 11 cycles (and uses the divide unit).

- **CDB**. In this stage, the result of the completed instruction is placed on the common data bus (CDB) and distributed along with its unique tag. The result value is copied by any reservation station that matches the tag; similarly, the register file reads the value if it matches the tag. At most one instruction can use the CDB each cycle, so a functional unit will stall (delaying all instructions currently executing in that F-unit) if the CDB is not available on the next cycle after an instruction completes Execute. If multiple F-units try to use the CDB in the same cycle, the one with the longest latency will be granted the use of the CDB and the other will have to wait one cycle. For instructions that modify two registers (e.g. popq), we will assume that the CDB is actually wide enough to distribute both result values and their associated tags in the same cycle. We also assume that all instructions require a CDB cycle, even if they don't have a value to distribute (e.g., nop, store).

- **Retire**. In this stage, instructions are removed from the ROB. At most one instruction can retire each cycle, and only the oldest entry in the ROB is eligible to retire. An instruction can retire only after it has completed all Execute cycles and passed through the CDB stage. If the retiring instruction is a store, the write to memory takes place in the same cycle that the store retires. (The mechanism used to accomplish this is described in the next section.) The halt instruction causes execution to stop, but only when the instruction retires.

Load-store handling

Accesses to data memory are tricky in a CPU with dynamic scheduling. Loads return a value used by subsequent instructions, so we want them to finish as quickly as possible. Stores do not return a value, and they cannot be allowed to actually modify memory until they retire. (Only when an instruction reaches the end of the ROB is it known that it did not follow a mispredicted branch or an instruction with an exception.) That means that a load can attempt to read from a memory location before a previously issued store actually writes to that location. Our system needs to detect this situation and handle it correctly.

In our CPU, correct execution of loads and stores depends on a *store buffer*. It is not shown in Figure 1, but it is essentially a FIFO that operates in parallel with the ROB. For a store to issue, there must also be an unused entry in the store buffer (in addition to a ROB entry and a reservation station). Thus, the store buffer contains an entry for every instruction that writes to memory fom the time it enters Issue until it reaches the Retire stage. Each store buffer entry contains an address field and a value field: the pending store will write the given value to the specified memory location.

A load cannot be allowed to read from memory until it is confirmed that no pending stores will write to the same location. This is enforced by comparing the address the load will read from against the address fields of all entries in the store buffer. We will assume that these comparisons can all take place in parallel in a single cycle. Once it is known that the load is independent of all store buffer entries, the load can safely read from memory. If the address in one or more entries matched (multiple stores to the same location can be pending), the load can obtain its value from the corresponding value field in the store buffer rather than wait for the store to write to memory.

Hopefully, this is intuitive and logical so far: our design must ensure that every load returns the value most recently written to that location. The store buffer is essential because the CPU can't speculatively execute a store; there is no good way to undo modifications to memory in the same way we can undo modifications to registers within the CPU.

Unfortunately, this isn't quite the full story yet. There are two additional complications that can arise in our load-store handling. First, recall that every Y86-64 instruction that accesses memory uses a register value (sometimes with a constant displacement) to determine the memory address. That register value may not be available at Issue, so the address field of some store buffer entries may have the value "unknown" when the store buffer entries are checked against a load address. If this happens, the load simply has to wait until the store address is determined and can be compared against the load address.

The second complication is that the register value the store is writing to memory may itself not be available at Issue. It is therefore possible for the address of a load to match the address in a store buffer entry that has an "unknown" store value. In this case, the load simply has to wait until the value appears in the store buffer entry, which will take place when the instruction producing the value puts its result on the CDB.

In our CPU, we'll assume the existence of a special *load buffer* (associated with the load-store unit and connected to the store buffer) where loads will wait when either of the above situations occurs. The load buffer keeps the load-store unit from locking up completely (and not servicing any new loads) when a load must wait for a store-buffer address or a store-buffer value.

In our simulated system, the store buffer has 8 entries and the load buffer has 4 entries.

5 Coding Rules and Objectives

Your assignment is to create a Y86-64 assembly file that consists of exactly 20 instructions. For convenience, you are given a file that serves as a starting point. Do not modify the first or last instructions in this file. You must use the remaining 18 instructions to achieve an unusual set of objectives, as discussed below. Your code may not include any branch, call or return instructions.

Normally our focus in writing code is to produce the correct result. In this lab, we are concerned almost exclusively with the timing of events within the CPU. Thus, the computations your code performs don't have to make a lot of sense, so long as the execution does not generate any exceptions. For example, you are free to use the result of a divide or multiply as the register used to compute a memory address, provided the result value is in range. You will discover that it is very easy to generate divide-by-zero exceptions (added to this CPU because it has a divide instruction) and bad address exceptions (on loads and stores). Exceptions of all types must be avoided.

The first goal you are to achieve with your code is to **maximize the cycle count from the time the first instruction is fetched until the last instruction (halt) retires**. To achieve this you'll want to create data dependencies between instructions with large latencies. Your score will depend on the cycle count you achieve.

Your code should also cause each of the following events to occur at least once:

- Pipeline stall. The instruction in the instruction register cannot finish the Issue stage.
- WAW hazard. Two instructions write to the same register, and the second finishes execution first.
- RAW hazard. The value of a source operand is not available during Issue, and the instruction must wait in a reservation station until the value is distributed via the CDB.
- Functional unit conflict. Two or more entries in the reservation stations of a given F-unit become ready on the same cycle, creating a conflict for that F-unit on the next cycle. (The oldest instruction will go first.)
- CDB conflict. Two or more functional units finish execution on the same cycle, creating a conflict for the CDB. (The instruction from the F-unit with the longest latency will go first.)
- Load forced to wait for an unknown store buffer address. A load must wait at least one cycle *because the store buffer includes a entry for which the address has not yet been determined.* (Note that this is quite different from waiting for the *value* in a store buffer entry with a matching address.)
- Load gets value from store buffer. A load matches on the address in the store buffer and get its return value from the associated value field of that entry.
- Loads executing out-of-order. A second load moves past a previously issued load and executes before the first.

Discussion, Hints, and Recommendations

Clearly, you'll want to chain together as many divq instructions as possible, where each is dependent on the previous. To avoid a divide-by-zero exception, you need one non-zero value in a register for each divide, but recognize that you already have one in %rsp. There is no need to modify %rsp after the first instruction in your code (and it is a bad idea on general principle), but note that you can divide registers that are uninitialized (and hence contain 0) by %rsp without causing any exceptions.

It may not be obvious, but it is possible to meet all lab requirements with just one store and two loads. As with all other instructions, you'll need to think carefully about the order of these instructions and all dependences that they are a part of. Any store to memory should write a memory location that is at or near the top of the stack – on general principle, you should NOT overwrite the instructions in your program. We can be more flexible with loads – you can access stack locations and also instruction byte codes. It is highly recommended that you make sure that all memory accesses are quad-word aligned. The simulator can handle unaligned accesses, but dependencies between loads and stores will be less obvious to you if the referenced locations overlap but are not identical.

It can be challenging to cause some of the required scenarios. Consider, for example, what must be involved in a WAW hazard. Obviously, you need two instructions that write the same result register, but there are other considerations. If there are *any* data dependencies between the two instructions such that the second must wait for the first to complete (either directly or indirectly through intermediate values and instructions), then there is no WAW hazard.

Another potentially tricky scenario is the load forced to wait because of an unknown address in the store buffer. As previously discussed, this is NOT the same as waiting for the store value to show up in a store buffer entry, and you will need to think carefully about how to set this up.

It can be difficult to get an instruction sequence that results in a CDB conflict. To get two functional units to finish an operation on the same cycle (and hence try to use the CDB on the next), consider using back-to-back instructions that differ by just one cycle in latency.

6 Evaluation

The lab is worth 100 points. If your code causes at least one instance of each of the following events, you will get the associated points. (Causing additional instances earns no additional points.)

- Pipeline stall: 10 points
- WAW hazard: 10 points
- RAW hazard: 10 points
- Functional unit conflict: 5 points
- CDB conflict: 5 points
- Load forced to wait for an unknown address in the store buffer: 5 points

- Load gets value from store buffer: 10 points
- Loads executing out-of-order: 5 points

The above components sum to 60 points. The remaining 40 points depend on the total cycle count (c) that your code requires to execute. Points will be allocated as follows:

- 40 points: $c \geq 170$
- 35 points: $160 \leq c < 170$
- 30 points: $150 \leq c < 160$
- 20 points: $140 \leq c < 150$
- 10 points: $130 \leq c < 140$
- 5 points: $120 \leq c < 130$
- 0 points: $c < 120$

To ensure that you don't just ignore one or more of the required scenarios and use the instructions you saved to increase the cycle count, *you can earn the full 40 points from your cycle count only if your code generates all required events*. If your submission does not cause each of the events to occur, you can earn at most 36 points for your cycle count. (Points for cycle counts in the top 4 categories are similarly reduced.)

Your submission will incur a 20 point penalty if more than 20 instructions are used. No points will be given to a submission that does not assemble and run as submitted, or to a submission that causes any sort of runtime exception in the simulator.

Since so much of your score depends on generating the required events, you will want to study output from the simulators (described in an appendix) to make sure your code functions as expected. Of particular interest is the dssim simulator which models the target Y86-64 CPU with dynamic scheduling on a cycle-accurate basis. That simulator will report the precise score you will receive for a given version of your code. Moreover, its detailed cycle-by-cycle output is very useful in understanding exactly what the CPU does and hand-crafting instruction sequences that cause the required events to occur.

7 Submission

When you are convinced that your solution meets all the requirements (or you are satisfied with the score you will receive), submit your lab5.ys file via Learning Suite. Remember to place your name in a comment before submitting.

Appendix: The Tools

On any of the spice machines, you should be able to find the programs described below in `/ee2/ee424/bin`.

- dsyas. This is a version of the Y86-64 assembler that has been extended to recognize the divq and mulq instructions.
- dsyis. This is a version of the sequential Y86-64 simulator that has been extended to recognize the divq and mulq instructions.
- dssim. This is a cycle-accurate simulator of the dynamically scheduled Y86-64 processor described in this document. It expects a .yo file as input.

The dsyas and dsyis programs work exactly like yas and yis that you've used in previous assignments. The dssim program generates detailed output about the execution timing, as well as the score that your Y86-64 program would receive if submitted in its current form. The simulator can be run with different *verbosity* levels that generate different levels of output. (Type `dssim` with no arguments to see all available options.) The simulator output at each verbosity level is described in the following subsections.

dssim: Verbosity Level 0

An example of the output at level 0 is given below. Each instruction executed appears on a separate line of the output that describes the cycle(s) spent in each of the stages. Detailed scoring for this lab assignment is also included in the output. In cases where your code didn't produce the events that you expected, you should turn to the detailed timing output of the higher verbosity levels.

```
Each line of output gives instruction, address, and the cycle it completes each stage
irmovq (PC=  0) IF:   1,  IS:   2,  WD:   3,  EX:   4,  CDB:   5,  RET:   6
call   (PC=  a) IF:   2,  IS:   3,  WD:   5,  EX:   7,  CDB:   8,  RET:   9
irmovq (PC= 38) IF:   3,  IS:   4,  WD:   5,  EX:   6,  CDB:   7,  RET:  10
irmovq (PC= 42) IF:   4,  IS:   5,  WD:   6,  EX:   8,  CDB:   9,  RET:  11
call   (PC= 4c) IF:   5,  IS:   6,  WD:   8,  EX:  10,  CDB:  11,  RET:  12
irmovq (PC= 56) IF:   6,  IS:   7,  WD:   8,  EX:   9,  CDB:  10,  RET:  13
irmovq (PC= 60) IF:   7,  IS:   8,  WD:   9,  EX:  11,  CDB:  12,  RET:  14
xorq   (PC= 6a) IF:   8,  IS:   9,  WD:  11,  EX:  12,  CDB:  13,  RET:  15
andq   (PC= 6c) IF:   9,  IS:  10,  WD:  12,  EX:  13,  CDB:  14,  RET:  16
jmp    (PC= 6e) IF:  10,  IS:  11,  WD:  14,  EX:  15,  CDB:  16,  RET:  17
...
halt   (PC= 13) IF:  34,  IS:  35,  WD:  37,  EX:  38,  CDB:  39,  RET:  45
Halt instruction retired: execution terminated
34 instructions executed in 45 cycles

Dynamic Scheduling Lab: Evaluation
   Pipeline stall: 0/10 pts
   WAW hazard: 10/10 pts
   RAW hazard: 10/10 pts
   F-unit conflict: 5/5 pts
   CDB conflict: 5/5 pts
   Load wait: 0/5 pts
   Load from SB: 0/10 pts
   Load out-of-order: 0/5 pts
   Total cycles: 45    Points for cycle count: 0
  Penalty! (-20 pts) Used more than 20 (34) instructions
Total points: 10
```

dssim: Verbosity Level 1

At level 1, the output takes the form shown in the example below, with one line of output per cycle of execution. Note the header row that helps to explain the various output fields in the lines that follow. Each line of the output shows the positions of the instructions in the various stages during the indicated cycle. We will consider each stage in turn.

```
cycle   addr    op           # RS        RS         finishing EX            #          #
  1: IF:   0 (irmovq)  <IS:   - >  <WD:    >  [EX:   ,   ,   ,   ]  <CDB:    > <RET:  - >
  2: IF:   a (jmp   )  <IS:   1 a>  <WD:    >  [EX:   ,   ,   ,   ]  <CDB:    > <RET:  - >
  3: IF:  38 (irmovq)  <IS:   2 a>  <WD:a   >  [EX:   ,   ,   ,   ]  <CDB:    > <RET:    >
  4: IF:  42 (irmovq)  <IS:   3 a>  <WD:a   >  [EX:  1,   ,   ,   ]  <CDB:    > <RET:    >
  5: IF:  4c (call  )  <IS:   4 a>  <WD:a   >  [EX:  2,   ,   ,   ]  <CDB:   1> <RET:    >
  6: IF:  d4 (irmovq)  <IS:   5 a>  <WD:a   >  [EX:  3,   ,   ,   ]  <CDB:   2> <RET:   1>
  7: IF:  de (irmovq)  <IS:   6 a>  <WD: a  >  [EX:  4,   ,   ,   ]  <CDB:   3> <RET:   2>
  8: IF:  e8 (irmovq)  <IS:   7 a>  <WD:a   >  [EX:   ,   ,   ,   ]  <CDB:   4> <RET:   3>
  9: IF:  f2 (andq  )  <IS:   8 b>  <WD:.   >  [EX:  6,  5,   ,   ]  <CDB:    > <RET:   4>
 10: IF:  f4 (je    )  <IS:   9 a>  <WD:a   >  [EX:  6,   ,   ,   ]  <CDB:   5> <RET:    >
```

- IF. The output gives a hex address and the op code of the instruction fetched during that cycle. To allow for a succinct representation in subsequent stages, the fetch cycle of each instruction is used thereafter as the ID for that instruction.
- IS. This states which instruction (if any) is in Issue in the current cycle. The fetch cycle is used as a shorthand ID to represent each instruction, so you can refer back to the fetch cycle to see what the instruction is and which functional unit it will issue to. Each F-unit has 4 reservation stations; the actual station number used is identified here using a letter from a to d. (Letters are used rather than numbers to make the output a little easier to parse.)
- WD. This field shows which instructions in reservation stations will proceed to Execute on the next cycle. (That would mean that each has all the operands it needs.) A positional representation is used: the field is 4 characters wide, and each slot corresponds to a specific F-unit. (The order left-to-right matches the top-to-bottom order of the F-units in Figure 1.) The letter indicates which reservation station holds the instruction that is moving on.
- EX. This field gives IDs of instructions that are completing Execute in the current cycle, and hence likely to be contenders for the use of the CDB on the next cycle. (In the case of a load, there is a possibility that the instruction will need to wait for an entry in the store buffer and hence move to the load unit rather than use the CDB on the next cycle.) Again, a positional representation is used, with the instruction IDs separated by commas.
- CDB. The ID of the instruction using the CDB on this cycle is given.
- RET. The ID of the instruction retiring during the current cycle is given.

This sumary output makes it quite easy to track the progress of any specific instruction through the processor. In the example above, the irmovq instruction at address 0 was fetched on cycle 1, it was issued on cycle 2, it spent cycle 3 in WD, it executed in cycle 4, it used the CDB in cycle 5, and it retired in cycle 6.

This output helps to identify some of the events or scenarios you are trying to cause in this lab. For example, during cycle 9, two instructions (5 and 6) are both finishing their last execution cycle, but only one (5 in this case because it is the oldest) will be granted the use of the CDB on the next cycle. Since our processor has no special storage buffers for instructions that have finished execution and are waiting to use the CDB, we assume the entire F-unit will stall if an instruction finishes executing but then must wait for the CDB. This is why instruction 6 finds itself in the same position in cycle 10 that it was in during cycle 9.

dsissm: Verbosity Level 2

At level 2, the simulator dumps virtually all of its internal state at each cycle of execution. For each cycle, the output begins with a summary line with the same information as at verbosity level 1. As shown in the example below, this is followed by a listing of much additional state information. We will consider each line of the detailed state output in turn.

```
38: IF:  89 (addq  )  <IS:  37 a>  <WD:aa .>  [EX: 30,   ,   ,   ]  <CDB:  32> <RET:  25>
     Reg: %rax:27.0 %rcx:33.0 %rbx:36.0 %rsi:34.0 CC:36.0
     RS 0.3: (irmovq) ID= 34, flaga=1 (taga=  0.0), flagb=1 (tagb=  0.0), flagcc=1 (tagcci=  0.0)
     RS 1.0: (rmmovq) ID= 37, flaga=0 (taga= 36.0), flagb=1 (tagb=  0.0), flagcc=1 (tagcci=  0.0)
     RS 3.0: (divq  ) ID= 36, flaga=0 (taga= 33.0), flagb=0 (tagb= 35.1), flagcc=1 (tagcci=  0.0)
     F-UNIT 0:  33
     F-UNIT 1:   -  35
     F-UNIT 2:  27   -   -   -   -
     F-UNIT 3:   -   -   -   -   -   -   -   -   -   -   -
     CDB: (jne   ) ID=30, tage=0.0, tagm=0.0, tagcco=0.0
     SB: (index:ID,addr,flag,data) (0: 37,0xffffffff,0,0)
     LB:
     ROB: 26,27,28,29,30,31,32,33,34,35,36,37
```

- Reg. This lists all program registers that have a pending write and gives the tag value associated with each. The tag corresponds to the ID of the instruction (its fetch cycle) that will produce the result. The decimal point on the tag is a notational convenience: some instructions generate two result values that are written to registers. Thus, instruction 35 may generate values associated with tags 35.0 and 35.1.

- RS. These lines of output list the contents of all non-empty reservation stations. Each line begins with a RS number $x.y$, where x is the number of the functional unit and y is the RS number within that functional unit. (F-units are numbered 0 to 3, matching the order of F-units in Figure 1 top-to-bottom.) The output gives the op code and the ID of the instruction in each RS slot. The remaining fields show the availability of all required operands. For each of valA, valB, and CC, there is a flag to show if the value is already present or if the instruction is waiting with a tag. (If the flag is 1, the RS has the corresponding value. If the flag is 0, the RS does not yet have the operand, and it will get it from the CDB when it matches on the indicated tag.) An instruction cannot leave a reservation station until all flags are 1.

- F-Unit. Each line of output shows instructions making progress through the indicated F-unit. Each F-unit is fully pipelined, so there is a separate internal stage for each cycle of Execute latency, and the output includes a slot for each stage. Instructions begin in the rightmost slot and work their way to the left. Thus, all instructions that could potentially use the CDB on the next cycle will appear in

the leftmost column in the current cycle. If the same instruction appears in the leftmost column on back-to-back cycles, then the functional unit has stalled (waiting for the CDB).

- CDB. The output states which, if any, instruction is using the CDB on that cycle. For convenience, both the op code and the instruction ID are given. The output line also gives the tag values associated with valE, valM, and CC. Any tag value that shows up here as non-zero will be matched against all tag values in reservation stations and the register file; if any of these match, they will grab the value and clear the tag (and set the flag in an RS). This is how the processor satisfies data dependencies.

- SB. This output lists all entries in the store buffer, with each entry enclosed in parentheses. The output specifies the SB index used (of the 8 total SB entries), the instruction ID of the store, the address of the memory location that the store will write to, the data value that will be written to that address, and a flag that indicates (1) if the data is already available or (0) still coming from an in-flight instruction. If the address is unknown (because it depends on an in-flight instruction), the address field will be set to -1 (as in the example shown above).

- LB. Next, the contents of all entries in the load buffer are listed. Each entry includes the load buffer index (of the 4 total LB entries), the ID of the load instruction, and the address that the load will access in memory. Remember, loads are placed in the load buffer only if they are waiting for an address or a value (or both) in a store buffer entry.

- ROB. This last line shows all entries in the reorder buffer. Instruction IDs are listed in order, with the oldest (and hence the next to retire) on the left. In our processor, at most one instruction can retire each cycle, at which point it is removed from the ROB. On the same cycle that stores retire, they modify memory and are deleted from the store buffer.