Description of our Tomasulo Based Machine (A Joint Effort Between SatTech and MichaelTronics)

The major components of the machine are as follows:

1. *Register File*: Contains the data produced by the functional units. The register file is read by instructions as they pass from the dispatch stage to the Issue stage. The register file is potentially written by instructions when they are in the Writeback stage. In this model, the same register set is used for both floating point and integer instructions.

2. *Register Status Table (RST)*: The register status table (RST) implements renaming. The RST contains two different halves: one speculative and one non-speculative. The speculative half contains the mapping of register names to RSV's assuming that the current unresolved branch is correctly predicted. The non-speculative half contains the mapping assuming that the current unresolved branch is incorrectly predicted. When an instruction proceeds from Dispatch to Issue, it checks to see if there is an unresolved branch. If there is, it reads and writes the corresponding RST entries from the speculative half. If there are no outstanding branches, it reads and writes the RST entries from the non-speculative half. Each input register of the instruction is looked up in the RST. If an RST entry is not empty, it indicates that value is currently being computed by another instruction. The instruction also updates the RST entry corresponding to the instruction's destination register with the tag indicating the Reservation Station (RSV) that the instruction is mapped to. When the instruction writes its results to the CDB, if the non-speculative half of the RST still contains the RSV tag for that instruction, the instruction's result will be written to the register file and the RST entry will be cleared. Similarly, the speculative half's RST entry will also be cleared if it contains the instruction's RSV tag.

3. *Reservation Stations (RSV's)*: These units hold instructions that have been dispatched but have not begun execution. Each reservation station entry contains the instruction, its operands, and a speculative dispatch bit (SDB). If an operand is not available, the entry will contain a tag (read from the RST), indicating the reservation station that will produce that operand. When the operand becomes available (via the CDB), the value will be put into the RSV (reservation station) entry. The SDB indicates whether an instruction has been speculatively dispatched (i.e. dispatched when there is an unresolved branch). A value of 1 indicated speculative dispatch. If our branch prediction was incorrect, all instructions that have been speculatively dispatched will be deleted from the reservation stations.

4. *Functional Units (FU's)*: There are several different types, depending on the operations they compute: integer, floating point add, floating point multiply/divide, branch. Effective address calculation for loads and stores is handled by a dedicated Address Resolution Unit. Branches are handled by the branch functional unit.

5. *Common Data Buses (CDB's)*: The CDB's, one per functional unit, transmit a tagged data word. The tag indicates the reservation station whose instruction generated the tag. Results from the execution units are written to the CDBs in the Writeback stage. Values from the CDBs are written to the register file at writeback, if the corresponding entry in the non-speculative half of the RST contains the RSV tag broadcasted on the CDB. Reservation stations snoop the CDB to see if any of the operands they need have just been written.

6. *Load/Store Address Queue*: An in-order queue that ensures that loads and stores resolve their addresses in order.

7. *Instruction Queue*: After fetch and decode (which are not shown as part of the system), an instruction is placed in the instruction queue. Instructions dispatch in-order when they reach the front of the queue.

Stages of Tomasulo:

There are 4 basic stages to Tomasulo's Algorithm:

1. *Dispatch (D)*: An instruction proceeds from dispatch to issue when it reaches the front of the instruction queue and there is a free reservation station (RSV) for the functional unit (FU) it needs. In the case of loads and stores, there must also be a slot in the Load/Store Address Queue. If the instruction at the beginning of the queue is blocked from issuing, all instructions behind it in the queue will also be blocked from issuing. When an instruction goes from dispatch to issue, it reads from the RF and bypasses in any results from instructions that are writing to the CDB in the same cycle. The instruction checks to see if there is an unresolved branch and sets the SDB in its RSV accordingly. The instruction also reads the RST (speculative or non-speculative depending on whether there is an unresolved branch) to see if the corresponding inputs are pending from another instruction. Finally, it updates the RST to map the instruction's destination register to its RSV number.

2. *Issue (I)*: In the issue stage, the instruction collects any values that it needs from the CDB (including branch outcomes, see the section on branches), based on the tag that is broadcast. If at the end of the cycle, the instruction would have both operands available, it will proceed the execute stage. However, the instruction must stay in the issue stage if there is an older instruction (i.e., dispatched earlier) assigned to the same FU that is also ready to proceed to the execute stage (i.e., it fails arbitration.) Thus, if the instruction does not fail arbitration, in the best case, there is a one cycle delay between dependent instructions.

3. *Execute (INT, A1-A2, M1-M3, BR, MEM)*: In this model, we assume that all functional units are fully pipelined. Note that an instruction will reserve its reservation station until the end of the writeback stage. An instruction waiting for this reservation station will occupy it (and be in the Issue stage) immediately afterwards.

4. *Writeback (WB)*: Once an instruction has finished executing, it will write its result to the CDB. We assume every functional unit has a dedicated CDB. During writeback, the appropriate register will be updated in the register file, if the non-speculative RST entry matches the tag. Furthermore, the non-speculative *and* speculative RST entries that match the tag will be cleared. Also during this stage, any instruction in a RSV waiting for that value will update its value field in the RSV. Branches will broadcast their tag and branch outcome on the dedicated branch CDB. Stores do not do anything in the writeback stage.

Please note that this differs from the description given in H&P. In H&P, *Dispatch* is referred to as "Issue" and *Issue* is a part of the "Execution" stage. The terminology (as noted in H&P) here is more standard.

Loads:

Upon dispatch, load instructions (of the register-immediate variety) are assigned a load reservation station and inserted into the Load/Store address queue. The address queue entry contains the load's RSV tag, the immediate field, and the value or tag for the register input. The address queue entries snoop the CDB waiting for any tag values that are necessary to resolve the address. If the *head* item in the address queue has valid inputs (from snooping the CDB, if necessary) at the end of the cycle, it continues to the address resolution logic (ARL). The ARL calculates the effective address (EA), and simultaneously compares it to all addresses currently sitting in the Store (but not load) reservation stations. The comparisons form a bit-vector of the RSV's that that particular load must wait for before proceeding. At the end of the cycle, the EA and the bit vector are slotted in the load's reservation station. When the load's bit vector has cleared, it will proceed to the execute stage, clearing the corresponding bit in the bit vector of the store reservation stations (including the corresponding bit of a store currently in the ARL stage). When the load has finished, it will send its result over the CDB, like any normal instruction. Note however, that the load must wait before entering the execute stage, if an older load or store instruction is also ready to enter the execute stage.

For notational and timing purposes, we say that a load goes from the D (dispatch) stage to the AQ (address queue) stage to the AR (address resolution) stage to the I (issue) stage, to the MEM (memory) stages, to the WB (writeback) stage.

Stores:

Upon exiting dispatch, store instructions (of the register-immediate variety) are assigned a *store* reservation station and inserted into the Load/Store address queue. The address queue entry contains the store's RSV tag, the immediate field, and the register value that will produce the value. The store reservation station contains the data value to write (or the RSV tag of the RSV that will produce the value to write). This RSV will be constantly snooping the CDB, waiting for the corresponding input to arrive, if needed. The address queue entries snoop the CDB waiting for any tag values that are necessary to resolve the address. If the *head* item in the address queue has valid inputs (from snooping the CDB, if necessary) at the end of the cycle, it continues to the address resolution logic (ARL). The ARL calculates the effective address (EA) and simultaneously compares it to all addresses currently sitting in the Load and Store RSV. The comparisons form a bit-vector of the RSV's that that particular store must wait for before proceeding. At the end of the cycle, the EA and the bit vector are slotted in the store's RSV. The store sits in the RSV (corresponding to the Issue stage) until its bit vector has been cleared and the data input is available. When the store's bit vector has cleared, it will proceed to the execute stage at the clock edge, clearing the corresponding bit in the bit vector of the other load and store reservation stations (including the corresponding bit of an instruction currently in the ARL stage). A store does not do anything in the writeback stage. Note that the store must wait before entering the execute stage if an older load or store instruction is also ready to enter the execute stage (i.e., it loses out on arbitration). (Note that even though it is pipelined over multiple cycles, the load/store functional unit has logic that can handle back-to-back loads and stores that access the same address.)

For notational and timing purposes, we say that a store goes from the D (dispatch) stage to the AQ (address queue) stage to the AR (address resolution) stage to the I (issue) stage, to the MEM (memory) stages, to the WB (writeback) stage. It will spend exactly one cycle in the AR and one cycle in the WB stage, and at least one cycle in each of the other stages.

Branches:

Branches post an interesting problem. In the simplest Tomasulo scheme, after a branch is dispatched, no other instructions would dispatch until the branch is resolved and the target is known. In this scheme, new instructions may not enter a reservation station until the outcome of the last branch is known. At the cost of some complexity, we can eliminate this limitation. We allow instructions to enter the reservation stations before the branch they depend upon is resolved. However, the instructions must wait for the branch to resolve before entering execution. We will refer to the latter scheme as a "speculative dispatch" machine (not to be confused with a speculative execution machine). This document describes a speculative dispatch machine.

A branch reads the "unresolved branch" bit to see if there is currently an unresolved branch. If there is an unresolved branch, we stall the branch from dispatching. If there is no unresolved branch, the branch instruction will set the unresolved branch bit to 1 and dispatch to the branch reservation station. When a branch dispatches, it also copies the non-speculative RST entries to the speculative RST entries. Instructions that are dispatched while the branch is still unresolved are said to be "speculatively dispatched" (SDB = 1) and will read and modify the speculative RST entries only. Branches wait in the branch reservation station, snooping the CDB for when their inputs are available. When its operands are ready, it proceeds to the branch resolution (execute) stage, where the outcome of the branch (predicted-correctly or -incorrectly) is determined. This result is broadcasted on the branch CDB, in the Writeback stage and the "unresolved branch" bit is set to 0. If the outcome is "incorrect", then the IQ is cleared and the PC is set to the correct fetch location. We assume that the correct instruction is available at the head of the IQ (i.e. it is in the dispatch stage) on the cycle after the branch outcome is written back. Furthermore, the speculative RST entries will be cleared and any instruction in a RSV that has a SDB set to 1 will mark itself as invalid. This includes instructions sitting in the address queue. If the outcome of the branch is "predicted correctly", then all reservation stations will set their SDB to 0 to indicate that they are no longer speculative. Also, we copy the speculative RST entries to the non-speculative entries. *An instruction may not proceed to the execute stage until the SDB is 0.* The semantics of this

system ensure that instructions that are in the RSVs but whose branches are already resolved as correctly predicted execute even when a later branch is incorrectly predicted. It also ensures that the instructions after an incorrectly predicted branch are squashed.

Finally, when an entry exits the ARL, it verifies that the instruction is still in the Load or Store RSV (and has not been killed.) If it is not, it has been killed and the address is dropped.

Pipeline Diagram for Tomasulo:

Unfortunately, H&P shuns the use of pipeline diagrams when describing the operation of Tomasulo's algorithm. While one can only speculate (pun intended) as to why they did this, it is actually much clearer to use pipeline diagrams than the tables they use.

In the lecture slides, an example is given of using pipeline diagrams to show execution with Tomasulo's. In addition to studying those slides, the following problems are meant to give you more practice for the exam. Here, we will assume that instructions have the following "cycles in execution:"

Integer: 1
FP Add: 2
FP Mult: 3
Memory Accesses: 1 cycle

Please attempt to do the pipeline diagrams before looking at the answer…

Example #1:

Draw a pipeline diagram for the following set of instructions:

daddui R1, R2, R3
daddui R2, R2, R5
daddui R1, R2, R4

		1	2	3	4	5	6	7
daddui	R1, R2, R3	D	I	INT	WB			
daddui	R2, R2, R5		D	I	INT	WB		
daddui	R1, R2, R4			D	I	I	INT	WB

Here, you should note that the 2nd add can start EX during the WB stage of the 1st because they are independent. The 3rd instruction must, however, wait until after the 2nd's WB because it depends on the result that is broadcast via the CDB.

Example #2:

Draw a pipeline diagram for the following set of instructions:

```
daddui    R1, R2, R3
beq       R1, R4, target
```

```
add.d       F1, F2, F3
```

Assume that the add.d is the “target” and that the BTB correctly identifies this branch as taken (so the add.d can dispatch the cycle after the branch dispatches).

		1	2	3	4	5	6	7	8	9
daddui	R1, R2, R3	D	I	INT	WB					
beq	R1, R4, target		D	I	I	BR	WB			
add.d	F1, F2, F3			D	I	I	I	A1	A2	WB

Notice that the add.d instruction may dispatch the cycle after the beq instruction, but since it is control dependent on it, it may not start execution until after the beq has broadcast its result on the CDB.

Example #3:

Draw a pipeline diagram for the following set of instructions:

```
add.d F1, F2, F3
add.d F2, F3, F4
add.d F3, F4, F5
```

Solution:

		1	2	3	4	5	6	7	8	9
add.d	F1, F2, F3	D	I	A1	A2	WB				
add.d	F2, F3, F4		D	I	A1	A2	WB			
add.d	F3, F4, F5			D	D	D	I	A1	A2	WB

Since we have only 2 reservation stations for the FP adder, the 3rd add.d must wait until after the WB stage of the 1st before it may proceed to the issue stage.

Example #4:

Draw a pipeline diagram for the following set of instructions:

```
        dsubui R1, R1, #1
        beqz R1, target 1
        ...
target1: bneqz R2, target2
        ...
target 2: daddui R5, R3, R7
```

Assume a perfect BTB and that R1 is initially 1 while R2 is initially 5.

Solution:

		1	2	3	4	5	6	7	8	9	10	11
dsubui	R1, R1, #1	D	I	INT	WB							
beqz	R1, target1		D	I	I	BR	WB					
bneqz	R2, target2			D	D	D	D	I	BR	WB		
daddui	R5, R3, R7							D	I	I	INT	WB

This is another case where we have to wait for a reservation station to become free before going to issue. This time, the target of the first branch is another branch so bneqz must wait in dispatch until after beqz writes back. While we can dispatch and issue daddui, we may not execute it until the outcome of the 2nd branch is determined.

Problem #1:

Draw a pipeline diagram for the following set of instructions:

```
        mul.d F1, F2, F3
        mul.d F1, F1, F4
        s.d 0(R1), F1
        beqz R2, target

        …

target: l.d F5, 0(R1)
        daddui R1, R1, #4
```

Assume that we have a BTB that correctly tells us the address so we can dispatch the instruction at "target" the cycle after we dispatch the branch instruction. Also, assume that the initial value of R1 is 5000.

Solution:

		1	2	3	4	5	6	7	8	9	10	11	12	13
mul.d	F1, F2, F3	D	I	M1	M2	M3	WB							
mul.d	F1, F1, F4		D	I	I	I	I	M1	M2	M3	WB			
s.d	0(R1), F1			D	AQ	AR	I	I	I	I	I	MEM		
beqz	R2, target				D	I	BR	WB						
l.d	F5, 0(R1)					D	AQ	AR	I	I	I	I	MEM	WB
daddui	R1, R1, #4						D	I	INT	WB				

Notice that the load must wait to access mem because the store wrote to the same address (5000).

Problem #2:

Draw a pipeline diagram for the following set of instructions:

```
daddui R2, R2, R4
daddui R1, R1, R2
l.d F1, 0(R1)
l.d F2, 0(R7)
add.d F3, F1, F2
```

Assume the following initial register values: R1 = 1000, R2 = 4, R4 = 100, R7 = 2500.

Solution:

		1	2	3	4	5	6	7	8	9	10	11	12	13	14
daddui	R2, R3, R4	D	I	INT	WB										
daddui	R1, R1, R2		D	I	I	INT	WB								
l.d	F1, 0(R1)			D	AQ	AQ	AQ	AR	I	MEM	WB				
l.d	F2, 0(R7)				D	AQ	AQ	AQ	AR	I	MEM	WB			
add.d	F3, F1, F2					D	I	I	I	I	I	I	A1	A2	WB

Note that the 2nd load must wait in AQ even though its operand is available because it is not at the head of the queue.

Problem #3:

Draw a pipeline diagram for the following set of instructions.

l.d	F1, 0(R1)
add.d	F1,F1,F2
mul.d	F1, F1, F3
s.d	0(R2), F1
daddui	R1, R1, #8
daddui	R2, R2, #8
sdubui	R3, R3, #1
bnez	R3, loop
l.d	F1, 0(R1)
add.d	…

Assume the first instruction is labeled "loop" so that the 2nd l.d is actually from a 2nd iteration of the same code. Assume the following initial values: R1 = 1000, R2 = 2000.

Solution:

		1	2	3	4	5	6	7	8	9	10	11	12	13	14
l.d	F1, 0(R1)	D	AQ	AR	I	MEM	WB								
add.d	F1,F1,F2		D	I	I	I	I	A1	A2	WB					
mul.d	F1, F1, F3			D	I	I	I	I	I	I	M1	M2	M3	WB	
s.d	0(R2), F1				D	AQ	AR	I	I	I	I	I	I	I	MEM
daddui	R1, R1, #8					D	I	INT	WB						
daddui	R2, R2, #8						D	I	INT	WB					
sdubui	R3, R3, #1							D	I	INT	WB				
bnez	R3, loop								D	I	I	BR	WB		
l.d	F1, 0(R1)									D	AQ	AR	I	MEM	WB
add.d	…										D	...			

Problem #4:

Draw a pipeline diagram for the following set of instructions.

```
Loop:   add.d F1, F2, F3
        add.d F2, F3, F4
        mul.d F1, F1, F6
        dsubui R1, R1, #1
        bneqz R1, loop
        l.d F5, 0(R2)
        add.d F5, F5, F3
```

Assume that R1 is initially 1 but that the loop is predicted as taken. Assume R2 to initially be 1000.

Solution:

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
add.d	F1, F2, F3	D	I	A1	A2	WB													
add.d	F2, F3, F4		D	I	A1	A2	WB												
mul.d	F1, F1, F6			D	I	I	M1	M2	M3	WB									
dsubui	R1, R1, #1				D	I	INT	WB											
bneqz	R1, loop					D	I	I	BR	WB									
add.d	F1, F2, F3						D	I	I	I									
add.d	F2, F3, F4							D	I	I									
mul.d	F1, F1, F6								D	I									
dsubui	R1, R1, #1									D									
l.d	F5, 0(R2)										D	AQ	AR	I	MEM	WB			
add.d	F5, F5, F3											D	I	I	I	I	A1	A2	WB

The four instructions highlighted in yellow have been dispatched but will be squashed after the misprediction occurs. The correct instruction will dispatch only after the WB of the branch occurs.