• Integer arithmetic/logic/Store instructions (inputs: two integer registers) and all Load instructions (input: one integer register)
• Floating-point arithmetic instructions (inputs: two floating-point registers)
• Floating-point Stores (inputs: one integer and one floating-point registers).

All values are forwarded as early as possible. Both register files are internally forwarded. All data hazards are resolved in the ID stage with a hazard detection unit (HDU). ID fetches registers from the integer and/or from the floating-point register file, as needed. The opcode selects the register file from which operands are fetched (S.D fetches from both)

a. To solve RAW data hazards on registers (integer and/or floating-point), hardware checks (interlocks) between the current instruction in ID and instructions in the pipeline may stall the instruction in ID. List first all **pipeline registers** that **must** be checked in ID. Since ME/WB may have two destination registers list them as ME/WB(int) or ME/WB(fp). Please don't list pipeline stages, list pipeline registers. Please make sure that the set of checks is minimum.

b. To solve WAW hazards on registers, we check the destination register in ID with the destination register of instructions in various pipeline stages. Please list the pipeline registers that must be checked. Make sure that the set of checks is minimum. IMPORTANT: remember that there is a mechanism in ID to avoid structural hazards on the write register ports of both register files.

Your solutions specifying the hazard detection logic should be written as follows for both RAW and WAW hazards:

--If Integer arithmetic/logic/Store/Load instruction in ID check <pipeline registers>
--If FP Load instruction in ID check <pipeline registers>
--If FP arithmetic instruction in ID check <pipeline registers>
--If FP Store instruction in ID check <pipeline registers>

Problem 3.9

Repeat Problem 3.8 for the superpipelined architecture in Figure 3.10. Assume forwarding for all instructions including FP Stores. Note that both floating-point and integer values can now be forwarded from both ME1/ME2 and ME2/WB.

Your solutions specifying the hazard detection logic should be written as follows for both RAW and WAW hazards:

--If Integer arithmetic/logic/Store instruction or Load instruction in ID check <pipeline registers>
--If FP Load instruction in ID check <pipeline registers>
--If FP arithmetic instruction in ID check <pipeline registers>
--If FP Store instruction in ID check <pipeline registers>

Problem 3.10

In the pipeline of Figure 3.9 WAW data hazards on registers are eliminated and exceptions can be handled in the WB stage where instructions complete in process order as in the classic 5-stage pipeline. As always values are forwarded to the input of the execution units.

a. List all required forwarding paths from pipeline registers to either EX or FP1 to fully forward values for all instructions. List them as source-->destination (e.g, FP2/FP1-->FP1)

b. Given those forwarding paths, indicate all checks that must be done in the hazard detection unit associated with ID to solve RAW hazards.
Your solutions specifying the hazard detection logic should be written as follows for RAW hazards on registers:

--If Integer arithmetic/logic/Store or Load instruction in ID check <pipeline registers>
--If FP arithmetic instruction in ID check <pipeline registers>
--If FP Store instruction in ID check <pipeline registers>

c. This architecture still has a subtle problem with respect to exception handling. Namely Stores are executed early and modify memory before they retire in the write-back stage. What is the problem. Can you propose a solution to this problem? (Please do not propose the solution of saving the memory value and then restoring it upon an exception.)

Problem 3.11

Consider the superscalar architecture of Figure 3.45. Two consecutive instructions are fetched at a time, incrementing PC by 8. To simplify pipeline interlocks, we split the decode stage into two stages ID1 and ID2. A switch with two settings (*straight* and *across*) separates ID1 and ID2. Upper ID2 must be an integer/branch instruction or an FP Load/Store. Lower ID2 must be an arithmetic FP instruction.

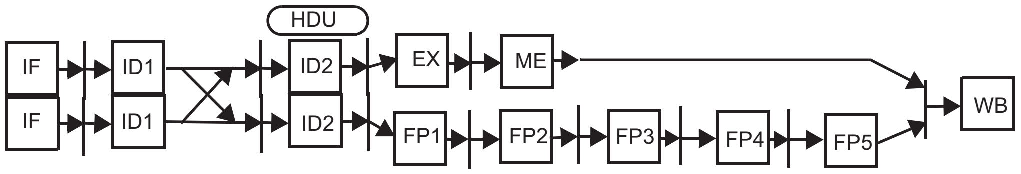


Figure 3.45. Two-way superscalar CPU

Let I1 be the upper instruction and I2 be the lower instruction in ID1. I1 must proceed to ID2 before or at the same time as I2 is allowed to proceed in order to adhere to process order. The following is done in ID1:

• If I1 is an integer/branch instruction or an FP Load/Store or a NOOP and I2 does not depend on I1 (the expected case) then set switch to straight.
• If I1 is an FP Load and I2 is an instruction using the value returned by the Load, stall I2 in ID1 and move I1 to ID2 with switch set to straight (Lower ID2 is NOOPed).
• If I1 is an FP arithmetic instruction, stall I2, and move I1 to ID2 with switch set to across (Upper ID2 is NOOPed).
• If I1 and I2 are both an integer/branch instruction or an FP Load/Store, stall I2 in ID1 and move I1 to ID2 with switch set to straight.(Lower ID2 is NOOPed)
• If I2 is an integer/branch instruction or an FP Load/Store and I1 is a NOOP, move I1, I2 to ID2 with switch set to across.

Thus if the two fetched instructions are dependent or are the wrong pair, they are serialized in ID1. Instructions in ID2 are subject to stalls due to pipeline hazard as in the single issue processor and proceed if they have no data hazard with previous instructions still in the pipeline. We deploy the same forwarding paths as in Figure 3.8. When instruction(s) are stalled in ID2, then instructions in IF and ID1 are stalled as well.

a. Describe briefly the function of the HDU associated with ID2

b. Explain how a branch is processed (consider both cases when the branch is upper or lower in ID1), assuming that branches are always predicted untaken by the hardware.

c. Consider the following code:

```
LOOP       L.D F2,0(R1)
           ADD.D F4,F2,F4
           L.D F6, -8(R1)
           ADD.D F8,F6,F4
           S.D F8, 0(R1)
           SUBI R1,R1,16
           BNEZ R1, LOOP
```

Compare the execution times of one iteration of this loop (not the last iteration) on this machine and the machines of Problem 3.8 and 3.9. For the machine of Problem 3.9 assume that branches can be resolved in EX1.

Problem 3.12

It would seem that superpipelining is a scalable solution to providing higher performance by deeply pipelining and increasing the clock rate. However there are three impediments to this:

• As the number of stages increases the functional logic delay decreases proportionally but the delay of the pipeline registers does not change
• The penalties (bubbles) in cycles caused by data dependencies increase
• The number of stages to flush on a mispredicted branch increases
• The penalty of cache misses increases as memory is not improved by deeper pipelines.

We model these effects as follows. Let T be the clock period of the single-cycle CPU. Let K be the number of stages in the pipeline. The clock cycle of the pipelined CPU with K stages is modeled as:

$$T_K = \frac{T}{K} + t_l = \frac{1}{f_K}$$

where t_l is the time needed to latch the output of each stage (setup time).

The penalty per instruction (in cycles) due to data hazards is modeled as:

$$\Delta_{data} = \alpha_d \frac{K}{5}$$

Similarly the penalty per instruction due to mispredicted branches (control hazards) is modeled as:

$$\Delta_{branches} = 2\alpha_b \frac{K}{5}$$

Finally, cache miss penalty also affects the speedup of deep pipelines because more cycles are needed to resolve cache misses. This can be modeled as:

$$\Delta_{memory} = \alpha_m \frac{K}{5}$$

α_d is approximately the average number of stalls per instruction in ID because of data hazard in the 5 stage pipeline, α_b is the fraction of instructions that are mispredicted branches (assuming a 2 clock penalty in the 5-stage pipeline), and α_m is the average number of cycles wasted by cache misses per instruction in the 5-stage pipeline.

a. Explain the rationale for these models (Hint: they are roughly based on the penalties in a 5-stage pipeline.)

b. Show an equation for the expected instruction throughput as a function of the number of stages *K*.

c. Is there an optimum pipeline depth with optimum throughput? If so, what is the optimum pipeline depth, as a function of t_l, α_d ,α_b, and α_m?

d. Take a practical case, with α_d =0.2 (one out of every 5 instructions has a 1 cycle delay due to raw hazards on registers), α_b = 0.06 (one out of 5 instructions is a branch and the static branch prediction success rate is 70%), and α_m=0.5 (0.05 Misses_Per_Instruction and 10 cycle miss penalty). Assume also that T, the instruction latency time in the single cycle CPU is 10nsec and the pipeline register overhead is 100ps. What is the optimum pipeline depth? What is the throughput of this optimum pipeline depth and how does it compare with the 5-stage pipeline, under the same assumptions?

Problem 3.13

In this problem we compare the performance of three dynamically scheduled processor architectures on a simple piece of code computing Y=Y*X+Z, where X,Y and Z are (double-precision--8bytes) floating-point vectors.

Using the core ISA of Table 3.3 in the notes, the loop body can be compiled as follows:

```
LOOP      L.D F0,0(R1)          /X[i] loaded in F0
          L.D F2,0(R2)          /Y[i] loaded in F2
          L.D F4,0(R3)          /Z[i] loaded in F4
          MUL.D F6,F2,F0        /Multiply X by Y
          ADD.D F8,F6,F4        /Add Z
          ADDI R1,R1,#8         /update address registers
          ADDI R2,R2,#8
          ADDI R3,R3,#8
          S.D F8, -8(R2)        /store in Y[i]
          BNE R4,R2,LOOP/       /(R4)-8 points to the last element of Y
```

The initial values in R1, R2, and R3 are such that the values are never equal during the entire execution (This is important for memory disambiguation.) The architectures are given in Figures 3.15, 3.23 and 3.27 and the same parameters apply. Branch BNE is always predicted taken (except in Tomasulo, where branches are not predicted at all and stall in the dispatch stage until their outcome is known).

Please keep in mind the following important rules (whenever they apply):

- Instructions are always fetched, decoded and dispatched in process order
- In speculative architectures, instructions always retire in process order
- In speculative architectures, Stores must wait until they reach the top of the ROB before they can issue to cache.

a. **Tomasulo algorithm--no speculation.** Please fill Table 3.24 clock by clock for the first iteration of the loop. Each entry should be the clock number when the event occurs, starting with clock 1. Add comments as you see fit (This helps understand your thinking.)

b. **Tomasulo algorithm with speculation.** Please fill Table 3.25 clock by clock for the first itera-

Table 3.24 Tomasulo algorithm--no speculation

		Dispatch	Issue	Exec/ start	Exec/ complete	Cache	CDB	COMMENTS
I1	L.D F0,0(R1)							
I2	L.D F2,0(R2)							

tion of the loop. Each entry should be the clock number when the event occurs, starting with clock 1. Please be attentive to the fact that (contrary to Tomasulo with no speculation) Stores cannot execute in cache until they reach the top of the ROB. Also branches are now predicted taken.

Table 3.25 Speculative Tomasulo algorithm

		Dispatch	Issue	Exec start	Exec complete	Cache	CDB	Retire	COMMENTS
I1	L.D F0,0(R1)								
I2	L.D F2,0(R2)								

c. **Speculative scheduling.** Please fill Table 3.26 clock by clock for the first iteration of the loop,. Each entry should be the clock number when the event occurs, starting with clock 1.

Table 3.26 Speculative scheduling

		Dispatch	Issue	Register fetch	Exec start	Exec complete	Cache	CDB	Retire	COMMENTS
I1	L.D F0,0(R1)									
I2	L.D F2,0(R2)									

d. Compute the minimum possible execution time given by the delay of the critical path in the dataflow graph of one iteration. Each node of the data flow graph is one instruction of the loop iteration. Nodes in the data flow graph are connected by directed edges. Each directed edge corresponds to a RAW dependency between two instructions, a parent and a child. An edge is labelled by the execution time of the parent instruction (in cycles). Only data dependencies are considered (assuming infinite amount of hardware resources, 100% cache hit rate and perfect branch prediction).

Draw the dataflow graph for the code of one iteration. Identify the critical path in the graph and compute the best possible execution time given by the data flow graph. Compare it with the execution times of the first iteration of the loop in all three cases above. To compute the execution time of the loop you can take the difference between the clock cycles when the first load issues in both iterations.

Problem 3.14

This problem is complex because we now deal with aspects of speculative execution not dealt with before, including multiple instruction dispatch, and structural hazards on the ROB.
To simplify, we use the same architecture as in Problem 3.13, part b, i.e., Tomasulo with speculation, in which the role of the ROB is to hold speculative values and track the thread order of instructions.

We dispatch two instructions per clock.
The ROB's size is 8 entries. When the ROB is full, dispatch is stalled. Dispatch waits until two entries are freed in the ROB before it dispatches its two instructions, so that instructions are always dispatched in pairs.

Table 3.27 Tomasulo algorithm with speculation (two way superscalar)

		Dispatch	Issue	Exec start	Exec complete	Cache	CDB	Retire	Comment
I1	L.D F0,0(R1)	1(7)	2	(3)	3	(4)	(5)	6	
I2	L.D F2,0(R2)	1(6)	3	(4)	4	(5)	(6)	7	

In the dispatch column, show the number of entries left in the ROB AT THE END OF THE CYCLE when it is dispatched between parentheses, just after the clock cycle number. An ROB entry is occupied in the cycle after a new instruction has dispatched. An ROB entry is freed in the same cycle an instruction enters the retire stage, and is available to a new instruction in the same cycle.

To see the effects of ROB hazards, we track two loop iterations. Please fill Table 3.27. The first two rows have been filled.

As in the previous problem, estimate the loop iteration time by the difference in cycle times between the issue clocks of the first load of the second iteration and of the first load of the third iteration. Does dual dispatch improve performance? Where are the bottlenecks?

Problem 3.15

In this problem we explore the effect of memory disambiguation using a very simple move in memory:

```
for (i=0;i<100;i++)
   A[i] = B[i];
```

In this code vector A and B are in different areas of memory so that they don't have common elements. The assembly code is:

```
LOOP      L.D F2,0(R1)
          ADDI R1,R1,#8
          ADDI R2,R2,#8
          S.D F2,-8(R2)
          BNEQ R1,R3,LOOP
```

The architecture is the architecture of Problem 3.14 (Tomasulo with speculation and two-way dispatch). Fill Table 3.28. Fill the table for two cases: 1) Conservative (a Load is not issued to cache until the addresses of all previous Stores are known and 2) Speculative (a Load is issued to cache optimistically when addresses of prior Stores are unknown). Remember that Stores can only issue to cache once they are at the top of the ROB.

Table 3.28 Tomasulo algorithm with speculation (two way superscalar)

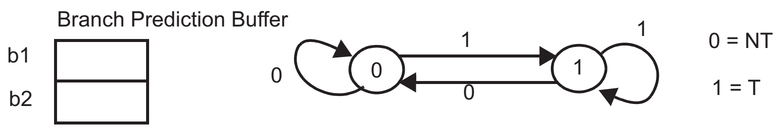
		Dispatch	Issue	Exec start	Exec complete	Cache	CDB	Retire	Comment
I1	L.D F2,0(R1)	1(7)	2	(3)	3	(4)	(5)	6	
I2	ADDI R1,R1,#8	1(6)	2	(3)	3	--	(4)	7	
I19	L.D F2,0(R1)								

Problem 3.16

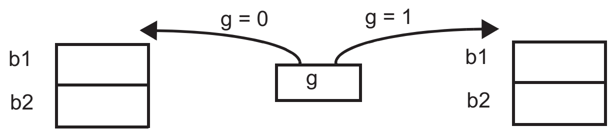
Consider the following code segment for a loop:

```
if (x is odd) then                  <-(branch b1)
    increment a                     <-(b1 untaken)
if (x is a multiple of 5) then      <-(branch b2)
    increment b                     <-(b2 untaken)
```

Assume that the following list of 9 values of x is processed by 9 iterations of this loop:
8, 9, 10, 11, 7, 20, 29, 30, 31.



(a) BPB with 1-bit predictor



(b) BPB with 1-bit predictors and 1-bit global history

Figure 3.46. Branch prediction Buffers

a. Assume that a one-bit state machine (see Figure 3.46(a)) is used as the prediction algorithm for predicting the execution of the two branches in this loop.

Show the predicted and actual branch directions of both b1 and b2 branch instructions for each iteration of this loop. Assume the initial state is 0, i.e. NT (not taken), for the predictor.
What are the prediction accuracies for b1 and for b2?
What is the overall prediction accuracy for both branches?

b. Assume now a two-level branch prediction scheme is used. In addition to the one-bit predictor, a one-bit global history register (g) is used. g stores the direction of the last executed branch (which may or may not be the same branch as the branch currently being predicted) and is used to index into two separate one-bit predictor tables as shown Figure 3.46(b).

Depending on the value of g, one of the two predictor table is selected and used for the normal one-bit prediction. Again, fill in the predicted and actual branch directions of b1 and b2 for nine iterations of the loop. Assume the initial value of g=0, i.e. NT. For each prediction, depending on the current value of g, only one of the two predictor tables is accessed and updated.

For each iteration of the loop show the value of g, the predicted and the actual branch directions of both b1 and b2 branch instructions. The initial state of the predictor tables is all 0's.
What are the prediction accuracies for b1 and b2?
What is the overall prediction accuracy?

c. What is the prediction success rate for branch b2 when g=0? Explain why this is.