Pipeline : Basic and Intermediate Concepts (Week#11)

Reducing the Cost of Branches through Prediction

Static Branch Prediction: A key way to improve compile-time branch prediction is to use profile information collected from earlier runs. The key observation is the behavior of branches is often bimodally distributed; that is, an individual branch is often highly biased toward taken or untaken.

Figure C.14 shows the success of branch prediction using this strategy. The effectiveness of any branch prediction scheme depends both on the accuracy of the scheme and the frequency of conditional branches, which vary in SPEC from 3% to 24%.

Misprediction rate for integer programs is higher and such programs have a higher branch frequency is a major limitation for static branch prediction.

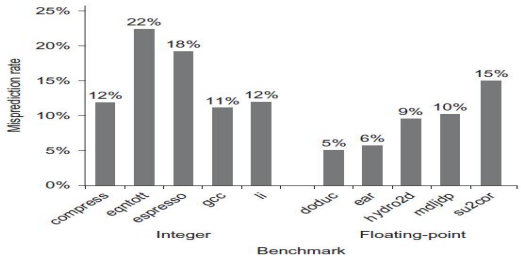

Figure C.14 Misprediction rate on SPEC92 for a profile-based predictor varies widely but is generally better for the floating-point programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%. The actual performance depends on both the prediction accuracy and the branch frequency, which vary from 3% to 24%.

Reducing the Cost of Branches through Prediction

Dynamic Branch Prediction and Branch-Prediction Buffers: The dynamic branch-prediction scheme is a branch-prediction buffer or branch history table.

A branch-prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not. This scheme is the simplest sort of buffer; it has no tags and is useful only to reduce the branch delay when it is longer than the time to compute the possible target PCs.

With such a buffer, we don't know if the prediction is correct—it may have been put there by another branch that has the same low-order address bits. The prediction is a hint that is assumed to be correct, and fetching begins in the predicted direction. If the hint turns out to be wrong, the prediction bit is inverted and stored back.

This buffer is effectively a cache where every access is a hit and the performance of the buffer depends on both how often the prediction is for the branch of interest and how accurate the prediction is when it matches.

Reducing the Cost of Branches through Prediction

1-bit prediction scheme: has a performance shortcoming: even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken, because the misprediction causes the prediction bit to be flipped.

To remedy this weakness, 2-bit prediction schemes are often used.

2-bit scheme Prediction Scheme: a prediction must miss twice before it is changed. Figure C.15 shows the finite-state processor for a 2-bit prediction scheme.

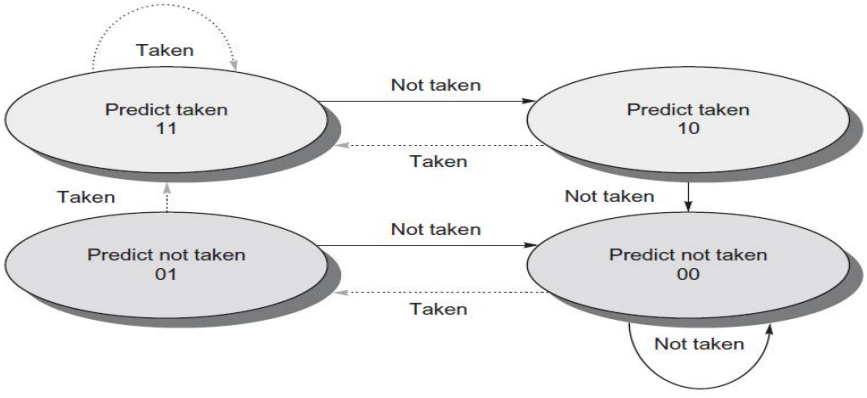

Figure C.15 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a

Reducing the Cost of Branches through Prediction

A branch-prediction buffer can be implemented as a small special “cache” accessed with the instruction address during the IF pipe stage, or as a pair of bits attached to each block in the instruction cache and fetched with the instruction.

If the instruction is decoded as a branch and if the branch is predicted as taken, fetching begins from the target as soon as the PC is known. Otherwise, sequential fetching and executing continue. Figure C.15 shows, if the prediction turns out to be wrong, the prediction bits are changed.

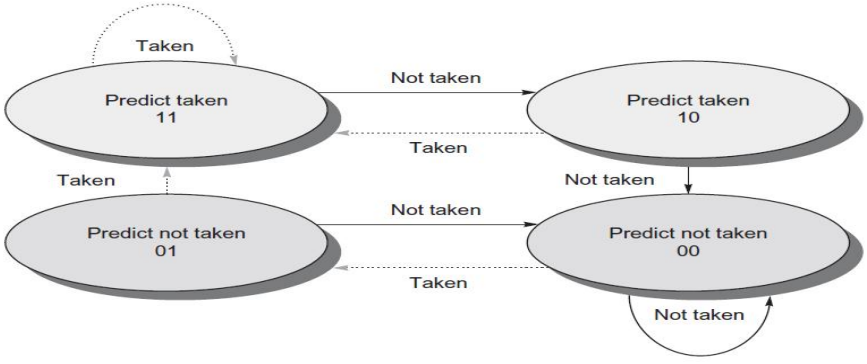

Figure C.15 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a

Reducing the Cost of Branches through Prediction

Figure C.16 shows that for SPEC89 benchmarks a branch-prediction buffer with 4096 entries results in a prediction accuracy ranging from over 99% to 82%, or a misprediction rate of 1%–18%. A 4K entry buffer used for these results is considered small and a larger buffer could produce better results.

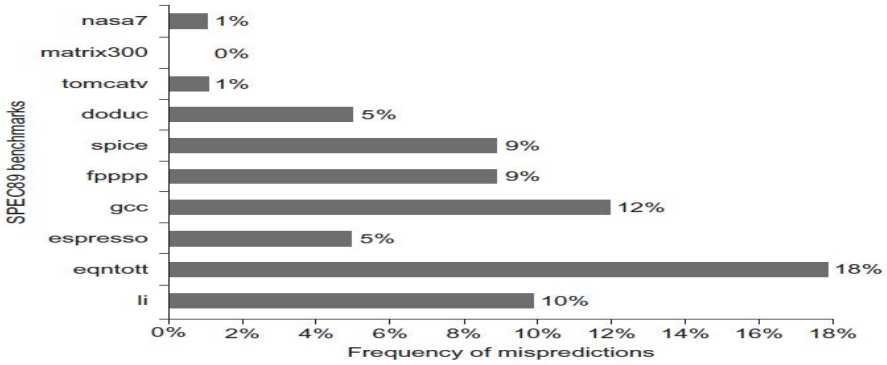

Figure C.16 Prediction accuracy of a 4096-entry 2-bit prediction buffer for the SPEC89 benchmarks. The misprediction rate for the integer benchmarks (gcc, espresso, eqntott, and li) is substantially higher (average of 11%) than that for the floating-point programs (average of 4%). Omitting the floating-point kernels (nasa7, matrix300, and tomcatv) still yields a higher accuracy for the FP benchmarks than for the integer benchmarks. These data, as well as the rest of the data in this section, are taken from a branch-prediction study done using the IBM Power architecture and optimized code for that system. See Pan et al. (1992). Although these data are for an older version of a subset of the SPEC benchmarks, the newer benchmarks are larger and would show slightly worse behavior, especially for the integer benchmarks.

How is Pipelining Implemented?

A Simple Implementation of RISC V: Every RISC V instruction can be implemented in 5 clock cycles. The 5 clock cycles are as follows:

1. Instruction fetch cycle (IF):

```
IR ← Mem[PC];
NPC ← PC + 4;
```

Operation—Send out the PC and fetch the instruction from memory into the instruction register (IR); increment the PC by 4 to address the next sequential instruction.

IR is used to hold the instruction thatwill be needed on subsequent clock cycles; register NPC is used to hold the next sequential PC.

2.Instruction decode/register fetch cycle (ID):

```
A ← Regs[rs1];
B ← Regs[rs2];
Imm ← sign-extended immediate field of IR;
```

How is Pipelining Implemented?

Operation—Decode the instruction and access the register file to read the registers (rs1 and rs2 are the register specifiers). The outputs of the general-purpose registers are read into two temporary registers (A and B) for use in later clock cycles. The lower 16 bits of the IR are also sign extended and stored into the temporary register Imm, for use in the next cycle.

Decoding is done in parallel with reading registers, which is possible because these fields are at a fixed location in the RISC V instruction format.

Immediate portion of a load and an ALU immediate is located in an identical place in every RISC V instruction, sign-extended immediate is also calculated during this cycle in case it is needed in the next cycle.

For stores, a separate sign-extension is needed because the immediate field is split in two pieces.

How is Pipelining Implemented?

3.Execution/effective address cycle (EX): ALU operates on the operands prepared in the prior cycle, performing one of four functions depending on the RISC V instruction type:

■ Memory reference:

```
ALUOutput ← A + Imm;
```

Operation—The ALU adds the operands to form the effective address and places the result into the register ALU Output.

■ Register-register ALU instruction:

```
ALUOutput ← A func B;
```

Operation—The ALU performs the operation specified by the function code (a combination of the func3 and func7 fields) on the value in register A and on the value in register B. The result is placed in the temporary register ALUOutput.

How is Pipelining Implemented?

■ Register-Immediate ALU instruction: `ALUOutput ← A op Imm;`

Operation— ALU performs the operation specified by the opcode on the value in register A and on the value in register Imm. Result is placed in the temporary register ALUOutput.

■ Branch:

```
ALUOutput ← NPC + (Imm << 2);
Cond ← (A == B)
```

Operation—ALU adds NPC to the sign-extended immediate value in Imm, which is shifted left by 2 bits to create a word offset, to compute the address of the branch target.

Register A has been read in the prior cycle,is checked to determine whether the branch is taken, by comparison with Register B, because we consider only branch equal.

Load-store architecture of RISC V means that effective address and execution cycles can be combined into a single clock cycle, because no instruction needs to simultaneously calculate a data address, calculate an instruction target address, and perform an operation on the data.

How is Pipelining Implemented?

4. Memory access/branch completion cycle (MEM):

The PC is updated for all instructions: `PC ← NPC;`

- Memory reference:

```
LMD ← Mem[ALUOutput] or
Mem[ALUOutput] ← B;
```

Operation—Access memory if needed. If the instruction is a load, data return from memory and are placed in the LMD (load memory data) register; if it is a store, then the data from the B register are written into memory. In either case, the address used is the one computed during the prior cycle and stored in the register ALUOutput.

- Branch:

```
if (cond) PC ← ALUOutput
```

Operation—If the instruction branches, the PC is replaced with the branch destination address in the register ALUOutput.

How is Pipelining Implemented?

5. Write-back cycle (WB):

- Register-register or Register-immediate ALU instruction:

```
Regs[rd] ← ALUOutput;
```

- Load instruction:

```
Regs[rd] ← LMD;
```

Operation—Write the result into the register file, whether it comes from the memory system (which is in LMD) or from the ALU (which is in ALUOutput) with rd designating the register.

How is Pipelining Implemented?

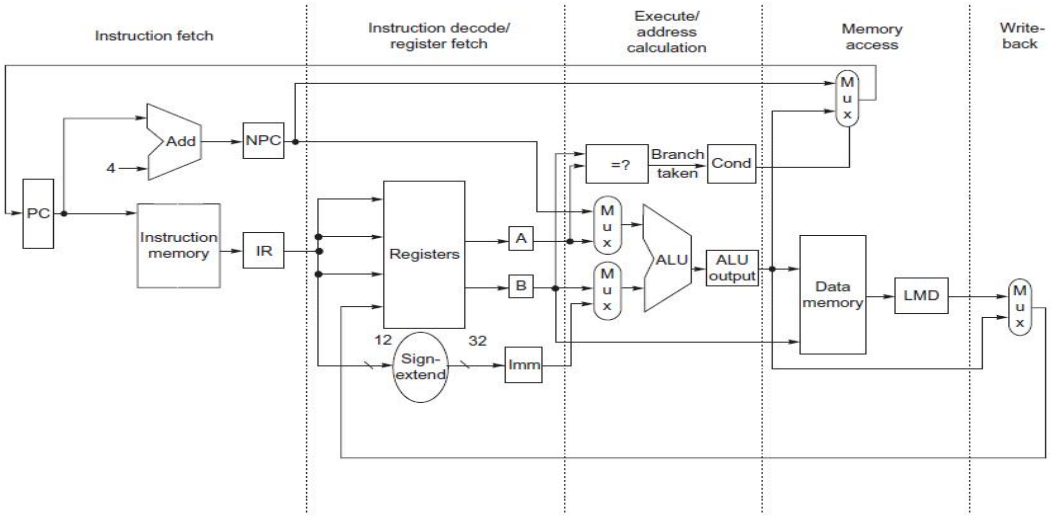

Figure C.18 The implementation of the RISC V data path allows every instruction to be executed in 4 or 5 clock cycles. Although the PC is shown in the portion of the data path that is used in instruction fetch and the registers are shown in the portion of the data path that is used in instruction decode/register fetch, both of these functional units are read as well as written by an instruction. Although we show these functional units in the cycle corresponding to where they are read, the PC is written during the memory access clock cycle and the registers are written during the write-back clock cycle. In both cases, the writes in later pipe stages are indicated by the multiplexer output (in memory access or write-back), which carries a value back to the PC or registers. These backward-flowing signals introduce much of the complexity of pipelining, because they indicate the possibility of hazards.

Basic Pipeline for RISC V

Figure C.19 shows the RISC V pipeline with the appropriate registers, called pipeline registers or pipeline latches, between each pipeline stage.

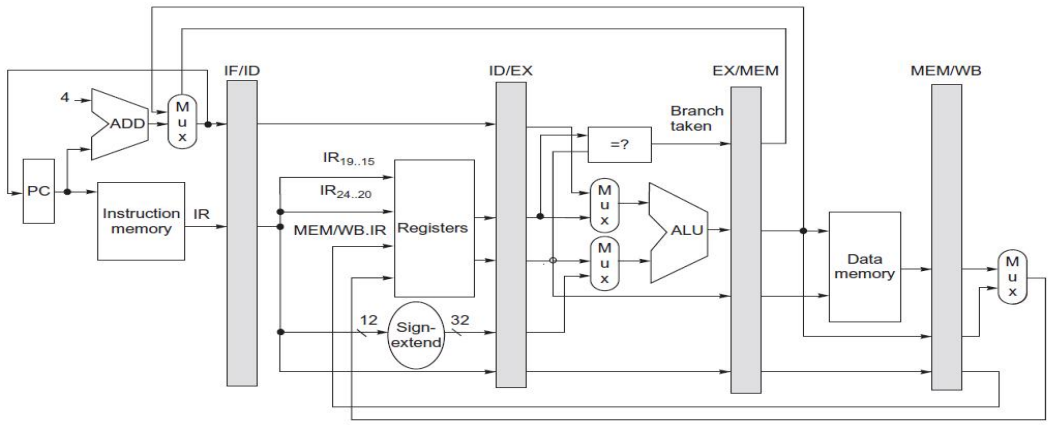

Figure C.19 The data path is pipelined by adding a set of registers, one between each pair of pipe stages. The registers serve to convey values and control information from one stage to the next. We can also think of the PC as a pipeline register, which sits before the IF stage of the pipeline, leading to one pipeline register for each pipe stage. Recall that the PC is an edge-triggered register written at the end of the clock cycle; hence, there is no race condition in writing the PC. The selection multiplexer for the PC has been moved so that the PC is written in exactly one stage (IF). If we didn't move it, there would be a conflict when a branch occurred, because two instructions would try to write different values into the PC. Most of the data paths flow from left to right, which is from earlier in time to later. The paths flowing from right to left (which carry the register write-back information and PC information on a branch) introduce complications into our pipeline.

Basic Pipeline for RISC V

- ✓ how to set the control for four multiplexers in the data path of Figure C.19. Two multiplexers in the ALU stage are set depending on the instruction type, which is dictated by IR field of the ID/EX register.
- ✓ Top ALU input multiplexer is set whether the instruction is a branch or not.
- ✓ Bottom multiplexer is set by whether the instruction is a register-register ALU operation or any other type of operation.
- ✓ Multiplexer in the IF stage chooses whether to use the value of the incremented PC or the value of the EX/MEM. ALUOutput (the branch target) to write into the PC. This multiplexer is controlled by the field EX/MEM.cond.
- ✓ Fourth multiplexer is controlled whether the instruction in the WB stage is a load or an ALU operation.
- ✓ One additional multiplexer needed that is not drawn in Figure C.19, but whose existence is clear from WB stage of an ALU operation.
- ✓ The destination register field is in one of two different places depending on the instruction type (register-register ALU versus either ALU immediate or load). Need a multiplexer to choose the correct portion of IR in MEM/WB register to specify the register destination field.

Stage	Any instruction		
IF	IF/ID.IR ← Mem[PC] IF/ID.NPC,PC ← (if ((EX/MEM.opcode == branch) & EX/MEM.cond){EX/MEM.ALUOutput) else (PC+4));		
ID	ID/EX.A ← Regs[IF/ID.IR[rs1]]; ID/EX.B ← Regs[IF/ID.IR[rs2]]; ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← sign-extend(IF/ID.IR[immediate field]);		
	ALU instruction	**Load instruction**	**Branch instruction**
EX	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput ← ID/EX.A func ID/EX.B; or EX/MEM.ALUOutput ← ID/EX.A *op* ID/EX.Imm;	EX/MEM.IR to ID/EX.IR EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm; EX/MEM.B ← ID/EX.B;	EX/MEM.ALUOutput ← ID/EX.NPC + (ID/EX.Imm << 2); EX/MEM.cond ← (ID/EX.A == ID/EX.B);
MEM	MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUOutput ← EX/MEM.ALUOutput;	MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOutput]; or Mem[EX/MEM.ALUOutput] ← EX/MEM.B;	
WB	Regs[MEM/WB.IR[rd]] ← MEM/WB.ALUOutput;	For load only: Regs[MEM/WB.IR[rd]] ← MEM/WB.LMD;	

Figure C.20 Events on every pipe stage of the RISC V pipeline. Let's review the actions in the stages that are specific to the pipeline organization. In IF, in addition to fetching the instruction and computing the new PC, we store the incremented PC both into the PC and into a pipeline register (NPC) for later use in computing the branch-target address. This structure is the same as the organization in Figure C.19, where the PC is updated in IF from one of two sources. In ID, we fetch the registers, extend the sign of the 12 bits of the IR (the immediate field), and pass along the IR and NPC. During EX, we perform an ALU operation or an address calculation; we pass along the IR and the B register (if the instruction is a store). We also set the value of cond to 1 if the instruction is a taken branch. During the MEM phase, we cycle the memory, write the PC if needed, and pass along values needed in the final pipe stage. Finally, during WB, we update the register field from either the ALU output or the loaded value. For simplicity we always pass the entire IR from one stage to the next, although as an instruction proceeds down the pipeline, less and less of the IR is needed.

Implementing the Control for the RISC V Pipeline

Instruction Issue: The process of letting an instruction move from the instruction decode stage (ID) into the execution stage (EX) of this pipeline is usually called instruction issue.

For RISC V integer pipeline, all the data hazards can be checked during the ID phase of the pipeline. If a data hazard exists, the instruction is stalled before it is issued.

Detecting interlocks early in the pipeline reduces the hardware complexity because the hardware never has to suspend an instruction that has updated the state of the processor, unless the entire processor is stalled.

Alternatively, we can detect the hazard or forwarding at the beginning of a clock cycle that uses an operand (EX and MEM for this pipeline).

How the interlock for a read after write (RAW) hazard with the source coming from a load instruction (called a load interlock) can be implemented by a check in ID, while the implementation of forwarding paths to the ALU inputs can be done during EX.

Implementing the Control for the RISC V Pipeline

Situation	Example code sequence	Action

No dependence	`ld x1,45(x2)`
	`add x5,x6,x7`
	`sub x8,x6,x7`
	`or x9,x6,x7`
		No hazard possible because no dependence exists on x1 in the immediately following three instructions

Dependence requiring stall	`ld x1,45(x2)`
	`add x5,x1,x7`
	`sub x8,x6,x7`
	`or x9,x6,x7`
		Comparators detect the use of x1 in the `add` and stall the `add` (and `sub` and `or`) before the `add` begins EX

Dependence overcome by forwarding	`ld x1,45(x2)`
	`add x5,x6,x7`
	`sub x8,x1,x7`
	`or x9,x6,x7`
		Comparators detect use of x1 in `sub` and forward result of load to ALU in time for `sub` to begin EX

Dependence with accesses in order	`ld x1,45(x2)`
	`add x5,x6,x7`
	`sub x8,x6,x7`
	`or x9,x1,x7`
		No action required because the read of x1 by `or` occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half

Figure C.21 Situations that the pipeline hazard detection hardware can see by comparing the destination and sources of adjacent instructions. This table indicates that the only comparison needed is between the destination and the sources on the two instructions following the instruction that wrote the destination. In the case of a stall, the pipeline dependences will look like the third case once execution continues (dependence overcome by forwarding). Of course, hazards that involve x0 can be ignored because the register always contains 0, and the preceding test could be extended to do this.

Implementing the Control for the RISC V Pipeline

Let's start with impementing the load interlock. If there is a RAW hazard with the source instruction being a load.

load instruction will be in the EX stage when an instruction that needs the load data will be in the ID stage.

Figure C.22 shows a table that detects all load interlocks when the instruction using the load result is in the ID stage.

Opcode field of ID/EX ($ID/EX.IR_{0..5}$)	Opcode field of IF/ID ($IF/ID.IR_{0..6}$)	Matching operand fields
Load	Register-register ALU, load, store, ALU immediate, or branch	`ID/EX.IR[rd] == IF/ID.IR[rs1]`
Load	Register-register ALU, or branch	`ID/EX.IR[rd] == IF/ID.IR[rs2]`

Figure C.22 The logic to detect the need for load interlocks during the ID stage of an instruction requires two comparisons, one for each possible source. Remember that the IF/ID register holds the state of the instruction in ID, which potentially uses the load result, while ID/EX holds the state of the instruction in EX, which is the load instruction.

Implementing the Control for the RISC V Pipeline

- ✓ Once a hazard has been detected, the control unit must insert the pipeline stall and prevent the instructions in the IF and ID stages.
- ✓ All the control information is carried in the pipeline registers. When a hazard is detected, need to change the control portion of the ID/EX pipeline register to all 0s, which happens to be a no-op (an instruction that does nothing, such as add x0,x0,x0).
- ✓ In a pipeline with more complex hazards, we can detect the hazard by comparing some set of pipeline registers and shift in no-ops to prevent erroneous execution.
- ✓ To implement the forwarding logic, pipeline registers contain both the data to be forwarded as well as the source and destination register fields.
- ✓ All forwarding logically happens from ALU or data memory output to ALU input, the data memory input, or the zero detection unit.
- ✓ Implement the forwarding by a comparison of the destination registers of the IR contained in the EX/MEM and MEM/WB stages against the source registers.

Implementing the Control for the RISC V Pipeline

Figure C.23 shows the comparisons and possible forwarding operations.

Pipeline register of source instruction	Opcode of source instruction	Pipeline register of destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	Register-register ALU, ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	`EX/MEM.IR[rd] == ID/EX.IR[rs1]`
EX/MEM	Register-register ALU, ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	`EX/MEM.IR[rd] == ID/EX.IR[rs2]`
MEM/WB	Register-register ALU, ALU immediate, Load	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	`MEM/WB.IR[rd] == ID/EX.IR[rs1]`
MEM/WB	Register-register ALU, ALU immediate, Load	ID/EX	Register-register ALU	Bottom ALU input	`MEM/WB.IR[rd] == ID/EX.IR[rs2]`

Figure C.23 Forwarding of data to the two ALU inputs (for the instruction in EX) can occur from the ALU result (in EX/MEM or in MEM/WB) or from the load result in MEM/WB. There are 10 separate comparisons needed to tell whether a forwarding operation should occur. The top and bottom ALU inputs refer to the inputs corresponding to the first and second ALU source operands, respectively, and are shown explicitly in Figure C.18 on page C.30 and in Figure C.24 on page C.36. Remember that the pipeline latch for destination instruction in EX is ID/EX, while the source values come from the ALUOutput portion of EX/MEM or MEM/WB or the LMD portion of MEM/WB. There is one complication not addressed by this logic: dealing with multiple instructions that write the same register. For example, during the code sequence `add x1, x2, x3; addi x1, x1, 2; sub x4, x3, x1`, the logic must ensure that the `sub` instruction uses the result of the `addi` instruction rather than the result of the `add` instruction. The logic shown here can be extended to handle this case by simply testing that forwarding from MEM/WB is enabled only when forwarding from EX/MEM is not enabled for the same input. Because the `addi` result will be in EX/MEM, it will be forwarded, rather than the `add` result in MEM/WB.

Implementing the Control for the RISC V Pipeline

Figure C.24 shows segments of pipelined data path with additional multiplexers and connections.

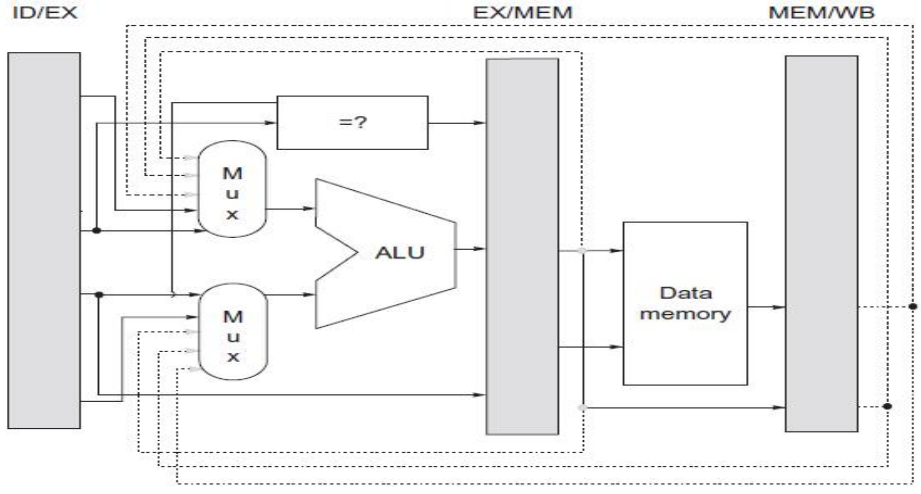

Figure C.24 Forwarding of results to the ALU requires the addition of three extra inputs on each ALU multiplexer and the addition of three paths to the new inputs. The paths correspond to a bypass of: (1) the ALU output at the end of the EX, (2) the ALU output at the end of the MEM stage, and (3) the memory output at the end of the MEM stage.

Dealing with Branches in Pipeline

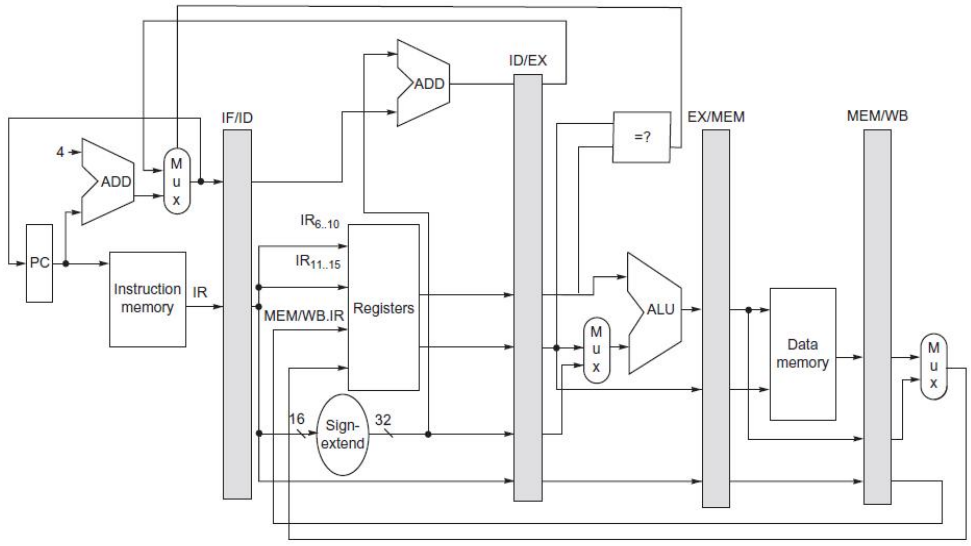

Figure C.25 To minimize the impact of deciding whether a conditional branch is taken, we compute the branch target address in ID while doing the conditional test and final selection of next PC in EX. As mentioned in Figure C.19, the PC can be thought of as a pipeline register (e.g., as part of ID/IF), which is written with the address of the next instruction at the end of each IF cycle.

Dealing with Branches in Pipeline

- ✓ Figure C.25 shows a pipelined data path assuming adder in ID and the evaluation of the branch condition in EX.
- ✓ This pipeline incur a two-cycle penalty on branches. RISC processors such as MIPS, test on branches was restricted to allow the test to occur in ID reducing branch delay to one cycle.
- ✓ ALU operation to a register followed by a conditional branch incurred a data hazard, which does not occur if the branch condition is evaluated in EX.
- ✓ As pipeline depths increased, branch delay increased which made dynamic branch prediction necessary.
- ✓ For example, a processor with separate decode and register fetch stages will probably have a branch delay at least 1 clock cycle longer.
- ✓ The branch delay turns into a branch penalty. Many older processors that implement more complex instruction sets have branch delays of 4 clock cycles or more and deeply pipelined processors have branch penalties of 6 or 7.
- ✓ Aggressive high-end superscalars such as Intel i7 may have branch penalties of 10–15 cycles. Deeper the pipeline, worse branch penalty in clock cycles.

Dealing with Exceptions

Exceptional situations are harder to handle in a pipelined processor because the overlapping of instructions makes it difficult to know whether an instruction can safely change the state of the processor.

In a pipelined processor, an instruction is executed piece by piece and is not completed for several clock cycles. Other instructions in the pipeline can raise exceptions that may force the processor to abort the instructions in the pipeline before they complete.

Types of Exceptions and Requirements: Exception cover all these mechanisms including following:

- I/O device request
- Invoking an operating system service from a user program
- Tracing instruction execution
- Breakpoint (programmer-requested interrupt)
- Integer arithmetic overflow
- FP arithmetic anomaly
- Page fault (not in main memory)

Dealing with Exceptions

- Misaligned memory accesses (if alignment is required)
- Memory protection violation
- Using an undefined or unimplemented instruction
- Hardware malfunctions
- Power failure

The requirements on exceptions can be characterized on five semi independent axes:

1. Synchronous versus asynchronous: If the event occurs at the same place every time the program is executed with the same data and memory allocation, the event is synchronous.

With the exception of hardware malfunctions, asynchronous events are caused by devices external to the processor and memory.

Asynchronous events usually can be handled after the completion of the current instruction, which makes them easier to handle.

Dealing with Exceptions

2. User requested versus coerced: User-requested exceptions are not really exceptions, because they are predictable. They are treated as exceptions because the same mechanisms that are used to save and restore the state are used for these user-requested events.

Only function of an instruction that triggers this exception is to cause the exception. user-requested exceptions can always be handled after the instruction has completed.

Coerced exceptions are caused by some hardware event that is not under the control of the user program. Coerced exceptions are harder to implement because they are not predictable.

3.User maskable versus user nonmaskable: If an event can be masked or disabled by a user task, it is user maskable. This mask simply controls whether the hardware responds to the exception or not.

Dealing with Exceptions

4.Within versus between instructions: This classification depends on whether the event prevents instruction completion by occurring in the middle of execution no matter how short or it is recognized between instructions.

Exceptions that occur within instructions are usually synchronous, because the instruction triggers the exception. It's harder to implement exceptions that occur within instructions than those between instructions, because the instruction must be stopped and restarted.

Asynchronous exceptions that occur within instructions arise from catastrophic situations (e.g., hardware malfunction) and always cause program termination.

5. Resume versus terminate: If the program's execution always stops after the interrupt, it is a terminating event.

If the program's execution continues after the interrupt, it is a resuming event.

It is easier to implement exceptions that terminate execution, because processor need not be able to restart execution of same program after handling the exception.

Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory protection violations	Synchronous	Coerced	Nonmaskable	Within	Resume
Using undefined instructions	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunctions	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

Figure C.26 Five categories are used to define what actions are needed for the different exception types. Exceptions that must allow resumption are marked as resume, although the software may often choose to terminate the program. Synchronous, coerced exceptions occurring within instructions that can be resumed are the most difficult to implement. We might expect that memory protection access violations would always result in termination; however, modern operating systems use memory protection to detect events such as the first attempt to use a page or the first write to a page. Thus, processors should be able to resume after such exceptions.

Dealing with Exceptions

- ✓ The difficult task is implementing interrupts occurring within instructions where the instruction must be resumed.
- ✓ Implementing such exceptions requires another program must be invoked to save the state of the executing program, correct the cause of the exception, and then restore the state of the program before the instruction that caused the exception can be tried again. This process must be effectively invisible to the executing program.
- ✓ If a pipeline provides the ability for the processor to handle the exception, save the state, and restart without affecting the execution of the program, the pipeline or processor is said to be restartable.
- ✓ early supercomputers and microprocessors often lacked this property, almost all processors today support it, at least for the integer pipeline, because it is needed to implement virtual memory.

Stopping and Restarting Execution

In unpipelined implementations, the most difficult exceptions have two properties:

(1) they occur within instructions (that is, in the middle of the instruction execution corresponding to EX or MEM pipe stages)

(2) they must be restartable.

- ✓ In RISC V pipeline, for example a virtual memory page fault resulting from a data fetch cannot occur until MEM (memory acces) stage of the instruction.
- ✓ A page fault must be restartable and requires the intervention of another process such as the operating system.
- ✓ Pipeline must be safely shut down and the state saved so that the instruction can be restarted in the correct state.
- ✓ Restarting is usually implemented by saving the PC of the instruction at which to restart.
- ✓ If the restarted instruction is not a branch then we will continue to fetch sequential successors and begin their execution in the normal fashion.
- ✓ If the restarted instruction is a branch, then we will reevaluate the branch condition and begin fetching from either the target or the fall-through.

Stopping and Restarting Execution

When an exception occurs, the pipeline control can take the following steps to save the pipeline state safely:

1. Force a trap instruction into the pipeline on the next IF.

2. Until the trap is taken, turn off all writes for the faulting instruction and for all instructions that follow in the pipeline; this can be done by placing zeros into the pipeline latches of all instructions in the pipeline, starting with the instruction that generates the exception, but not those that precede that instruction. This prevents any state changes for instructions that will not be completed before the exception is handled.

3. After the exception-handling routine in the operating system receives control, it immediately saves the PC of the faulting instruction. This value will be used to return from the exception later.

A 'Fall-through Branch' refers to the path taken when a branch instruction is not executed, leading to low coverage monitoring effectiveness. It is important to detect both the fall-through path and the target path to ensure comprehensive testing of branches in a program

Stopping and Restarting Execution

- ✓ After exception has been handled, special instructions return the processor from exception by reloading PCs and restarting instruction stream.
- ✓ If pipeline can be stopped so instructions before the faulting instruction are completed and can be restarted from scratch, pipeline is said to have precise exceptions.
- ✓ Faulting instruction would not have changed state and correctly handling exceptions require that faulting instruction have no effects.
- ✓ For floating-point exceptions, faulting instruction on some processors writes its result before exception can be handled. Hardware must be prepared to retrieve source operands, even if the destination is identical to one of source operands.
- ✓ To overcome this, many high-performance processors have introduced two modes of operation. One mode has precise exceptions and other (fast or performance mode) does not. Precise exception mode is slower, since it allows less overlap among floating-point instructions.
- ✓ Supporting precise exceptions is a requirement. Any processor with demand paging or IEEE arithmetic trap handlers must make its exceptions precise, either in hardware or software support. For integer pipelines, task of creating precise exceptions is easier.

Exceptions in RISC V

Figure C.27 shows RISC V pipeline stages and which problem exceptions might occur in each stage. With pipelining, multiple exceptions may occur in same clock cycle because there are multiple instructions in execution. For example, consider this instruction sequence:

ld	IF	ID	EX	MEM	WB	
add		IF	ID	EX	MEM	WB

Pipeline stage	Problem exceptions occurring
IF	Page fault on instruction fetch; misaligned memory access; memory protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory protection violation
WB	None

Figure C.27 Exceptions that may occur in the RISC V pipeline. Exceptions raised from instruction or data memory access account for six out of eight cases.

Instruction Set Complications

- ✓ RISCV pipeline writes that result only at the end of an instruction's execution. When an instruction is guaranteed to complete, it is called committed.
- ✓ In the RISC V integer pipeline, all instructions are committed when they reach the end of the MEM stage (or beginning of WB) and no instruction updates the state before that stage.
- ✓ **Precise exceptions:** are straightforward. Some processors have instructions that change the state in the middle of the instruction execution, before the instruction and its predecessors are guaranteed to complete.
- ✓ For example, autoincrement addressing modes in the IA-32 architecture cause the update of registers in themiddle of an instruction execution.
- ✓ If the instruction is aborted because of an exception, it will leave the processor state altered. Although we know which instruction caused the exception, without additional hardware support the exception will be imprecise because the instruction will be half finished.

Instruction Set Complications

- ✓ Restarting the instruction stream after such an imprecise exception is difficult.
- ✓ Alternatively, we could avoid updating state before instruction commits, but this may be difficult or costly, because there may be dependences on the updated state: consider a VAX instruction that autoincrements the same register multiple times.
- ✓ To maintain a precise exception model, most processors with such instructions have ability to back out any state changes made before the instruction is committed.
- ✓ If an exception occurs, processor uses this ability to reset the state of processor to its value before the interrupted instruction started.
- ✓ A related source of difficulties arises from instructions that update memory state during execution, such as the string copy operations on Intel architecture.
- ✓ To restart these instructions, the instructions are defined to use general-purpose registers as working registers.
- ✓ State of partially completed instruction is always in registers, which are saved on an exception and restored after exception.

Instruction Set Complications

- ✓ A different set of difficulties arises from odd bits of state that may create additional pipeline hazards or may require extra hardware to save and restore.
- ✓ Condition codes are example. Many processors set condition codes implicitly as part of instruction. This approach has advantages because condition codes decouple the evaluation of condition from actual branch.
- ✓ Implicitly set condition codes can cause difficulties in scheduling any pipeline delays between setting condition code and branch because most instructions set condition code and cannot be used in the delay slots between condition evaluation and the branch.
- ✓ In processors with condition codes, processor must decide when branch condition is fixed. This involves finding out when condition code has been set for the last time before the branch.
- ✓ In most processors with implicitly set condition codes, this is done by delaying the branch condition evaluation until all previous instructions have had a chance to set the condition code.

Instruction Set Complications

A final thorny area in pipelining is multicycle operations. Imagine trying to pipeline a sequence of x86 instructions such as this:

```
mov       BX, AX          ; moves between registers
add       42(BX+SI),BX    ;adds memory contents and register
                          ;  to same memory location
sub       BX,AX           ;subtracts registers
rep movsb                 ;moves a character string of
                          ; length given by register CX
```

- ✓ None of these instructions is particularly long (an x86 instruction can be up to 15 bytes), they differ radically in number of clock cycles from as low as hundreds of clock cycles.
- ✓ These instructions also require different numbers of data memory accesses from zero to possibly hundreds. The data hazards are very complex and occur both between and within instructions.
- ✓ Simple solution of making all instructions execute for same number of clock cycles is unacceptable because it introduces an enormous number of hazards and bypass conditions and makes long pipeline.
- ✓ Pipeline the microinstruction execution; a microinstruction is a simple instruction used in sequences to implement a more complex instruction set. Microinstructions are simple, pipeline control is much easier.
- ✓ Intel IA-32 microprocessors have used this strategy of converting the IA-32 instructions into microoperations and then pipelining the microoperations.

Extending the RISC V Integer Pipeline to Handle Multicycle Operations

RISC V FP operations complete in 1 clock cycle or even in 2. Accepting a slow clock or using enormous amounts of logic in the FP units or both.

FP pipeline will allow for a longer latency for operations. FP instructions as having the same pipeline as the integer instructions, with two important changes.

a) EX cycle may be repeated as many times as needed to complete the operation. Number of repetitions can vary for different operations.

b) There may be multiple FP functional units. A stall will occur if instruction to be issued will cause either a structural hazard for the functional unit.

Let's assume that there are four separate functional units in our RISC V implementation:

1. The main integer unit that handles loads and stores, integer ALU operations, and branches
2. FP and integer multiplier
3. FP adder that handles FP add, subtract, and conversion
4. FP and integer divider

Extending the RISC V Integer Pipeline to Handle Multicycle Operations

- ✓ Figure C.28 shows the resulting pipeline structure. EX is not pipelined, no other instruction using that functional unit may issue until the previous instruction leaves EX.
- ✓ If an instruction cannot proceed to EX stage, entire pipeline behind that instruction will be stalled.
- ✓ intermediate results are probably not cycled around the EX unit.
- ✓ EX pipeline stage has some number of clock delays larger than 1.To allow pipelining of some stages and multiple ongoing operations.

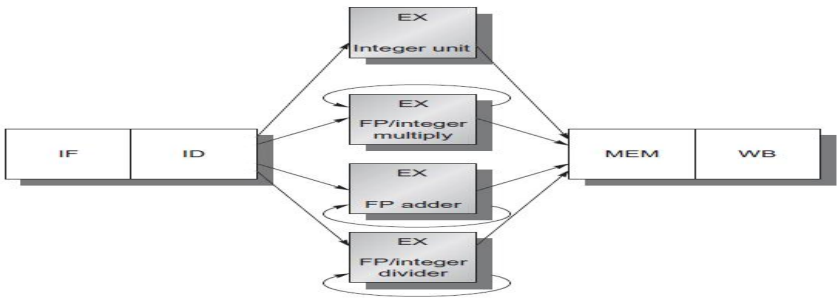

Figure C.28 The RISC V pipeline with three additional unpipelined, floating-point, functional units. Because only one instruction issues on every clock cycle, all instructions go through the standard pipeline for integer operations. The FP operations simply loop when they reach the EX stage. After they have finished the EX stage, they proceed to MEM and WB to complete execution.

Extending the RISC V Integer Pipeline to Handle Multicycle Operations

To describe such a pipeline, define both the latency of the functional units and also the initiation interval or repeat interval.

Latency: Number of intervening cycles between an instruction that produces a result and an instruction that uses the result. The initiation or repeat interval is the number of cycles that must elapse between issuing two operations of a given type.

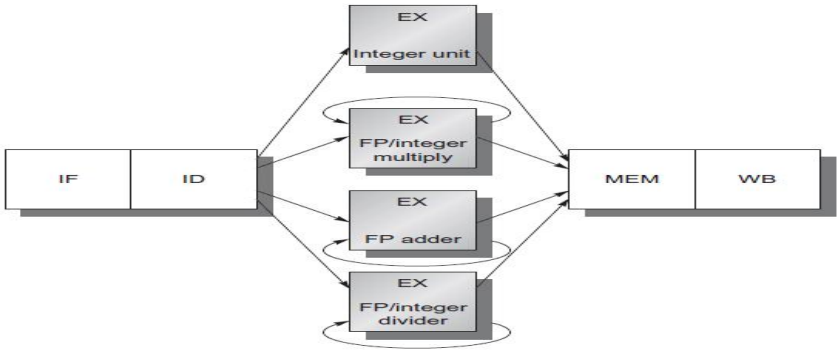

Figure C.28 The RISC V pipeline with three additional unpipelined, floating-point, functional units. Because only one instruction issues on every clock cycle, all instructions go through the standard pipeline for integer operations. The FP operations simply loop when they reach the EX stage. After they have finished the EX stage, they proceed to MEM and WB to complete execution.

Extending the RISC V Integer Pipeline to Handle Multicycle Operations

- ✓ Latencies and initiation intervals shown in Figure C.29. Integer ALU operations have a latency of 0, because results can be used on the next clock cycle, and loads have a latency of 1, because their results can be used after one intervening cycle.
- ✓ Most operations consume their operands at the beginning of EX. Latency is the number of stages after EX that an instruction produces a result. For example, zero stages for ALU operations and one stage for loads.
- ✓ The primary exception is stores which consume the value being stored one cycle later. Latency to a store for the value being stored, but not for the base address register will be one cycle less.
- ✓ Pipeline latency is equal to one cycle less than depth of the execution pipeline, which is the number of stages from EX stage to the stage that produces the result.
- ✓ For the preceding example pipeline, the number of stages in an FP add is four, while the number of stages in an FP multiply is seven.
- ✓ To achieve a higher clock rate, designers need to put fewer logic levels in each pipe stage, which makes the number of pipe stages required for more complex operations larger. Penalty for faster clock rate is longer latency for operations.

Extending the RISC V Integer Pipeline to Handle Multicycle Operations

Example pipeline structure in Figure C.29 allows up to four outstanding FP adds, seven outstanding FP/integer multiplies, and one FP divide.

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

Figure C.29 Latencies and initiation intervals for functional units.

Extending the RISC V Integer Pipeline to Handle Multicycle Operations

The repeat interval is implemented in Figure C.30 by adding additional pipeline stages, which will be separated by additional pipeline registers. Units are independent. Pipeline stages that take multiple clock cycles such as divide unit are further subdivided to show latency of those stages.They are not complete stages, only one operation may be active.

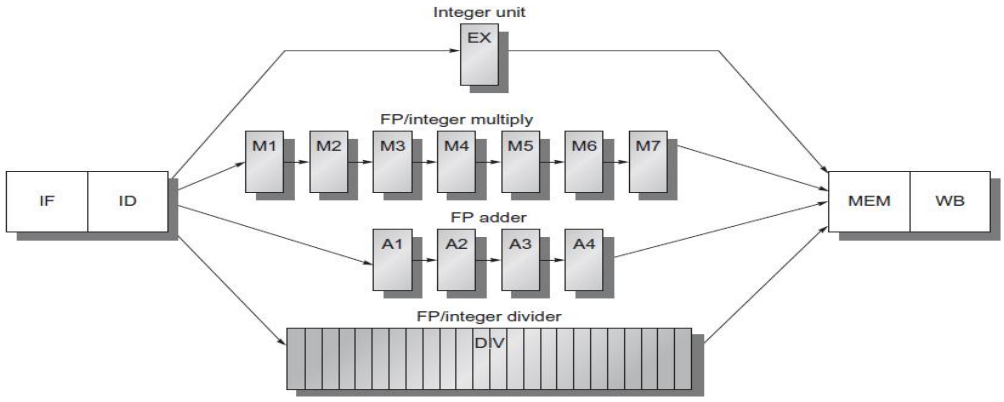

Figure C.30 A pipeline that supports multiple outstanding FP operations. The FP multiplier and adder are fully pipelined and have a depth of seven and four stages, respectively. The FP divider is not pipelined, but requires 24 clock cycles to complete. The latency in instructions between the issue of an FP operation and the use of the result of that operation without incurring a RAW stall is determined by the number of cycles spent in the execution stages. For example, the fourth instruction after an FP add can use the result of the FP add. For integer ALU operations, the depth of the execution pipeline is always one and the next instruction can use the results.

Extending the RISC V Integer Pipeline to Handle Multicycle Operations

Figure C.31 shows for a set of independent FP operations and FP loads and stores.

fmul.d	IF	ID	*M1*	M2	M3	M4	M5	M6	M7	MEM	WB
fadd.d		IF	ID	*A1*	A2	A3	**A4**	**MEM**	**WB**		
fadd.d			IF	ID	*EX*	**MEM**	**WB**				
fsd				IF	ID	*EX*	**MEM**	**WB**			

Figure C.31 The pipeline timing of a set of independent FP operations. The stages in italics show where data are needed, while the stages in bold show where a result is available. FP loads and stores use a 64-bit path to memory so that the pipelining timing is just like an integer load or store.

Hazards and Forwarding in Longer Latency Pipelines

There are a number of different aspects to the hazard detection and forwarding for a pipeline like that shown in Figure C.30.

1. Divide unit is not fully pipelined, structural hazards can occur. These will need to be detected and issuing instructions will need to be stalled.

2. Instructions have varying running times, the number of register writes required in a cycle can be larger than 1.

3. Write after write (WAW) hazards are possible, because instructions no longer reach WB in order. Write after read (WAR) hazards are not possible, because the register reads always occur in ID.

4. Instructions can complete in a different order than they were issued, causing problems with exceptions; we deal with this in the next subsection.

5. Because of longer latency of operations, stalls for RAW hazards will be more frequent.

Hazards and Forwarding in Longer Latency Pipelines

The increase in stalls arising from longer operation latencies is fundamentally the same as that for the integer pipeline. Fig C.32 shows a FP code sequence and the resultant stalls.

	Clock cycle number																
Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
`fld f4,0(x2)`	IF	ID	EX	MEM	WB												
`fmul.d f0,f4,f6`		IF	ID	Stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
`fadd.d f2,f0,f8`			IF	Stall	ID	Stall	Stall	Stall	Stall	Stall	Stall	A1	A2	A3	A4	MEM	WB
`fsd f2,0(x2)`					IF	Stall	Stall	Stall	Stall	Stall	Stall	ID	EX	Stall	Stall	Stall	MEM

Figure C.32 **A typical FP code sequence showing the stalls arising from RAW hazards.** The longer pipeline substantially raises the frequency of stalls versus the shallower integer pipeline. Each instruction in this sequence is dependent on the previous and proceeds as soon as data are available, which assumes the pipeline has full bypassing and forwarding. The `fsd` must be stalled an extra cycle so that its MEM does not conflict with the `fadd.d`. Extra hardware could easily handle this case.