CS4617 Computer Architecture

Lectures 21 – 22: Pipelining

Reference: Appendix C, Hennessy & Patterson

Dr J Vaughan

November 2013

MIPS data path implementation (unpipelined)

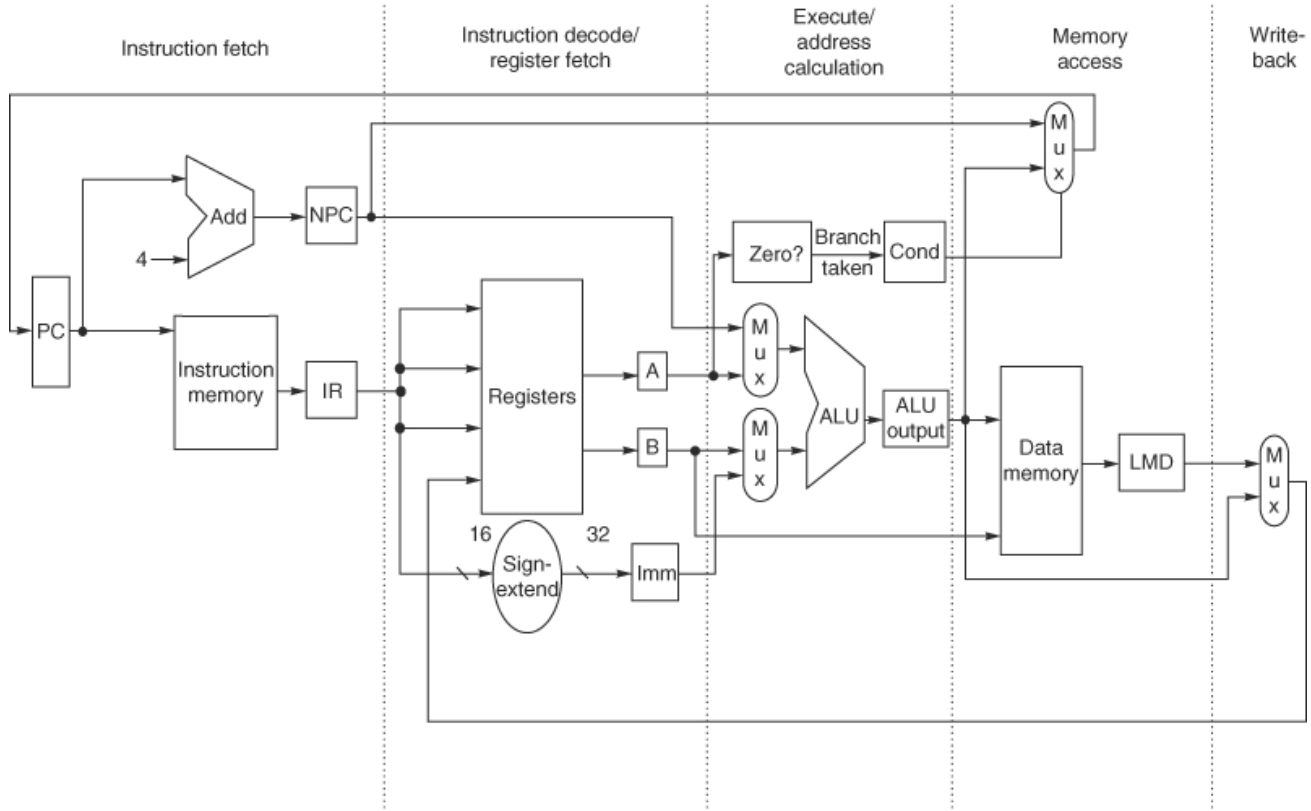

Figure C.21 The implementation of the MIPS data path allows every instruction to be executed in 4 or 5 clock cycles. Although the PC is shown in the portion of the data path that is used in instruction fetch and the registers are shown in the portion of the data path that is used in instruction decode/register 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, since they indicate the possibility of hazards.

Basic Pipeline for MIPS

- Figure C21 is adapted by adding interstage registers to become the pipeline shown in Figure C22.
- The pipeline registers carry data and control from one stage to the next
- Values are copied along the registers until no longer needed
- The temporary registers used in the unpipelined processor are unsuitable as the values they contain could be overwritten before being completely used
- All registers needed to hold values temporarily between clock cycles within one instruction are contained in pipeline registers
- The pipeline registers carry data and control from one stage to the next
- A value needed in a later stage must be copied between pipeline registers until it is no longer needed

MIPS data path implementation (pipelined)

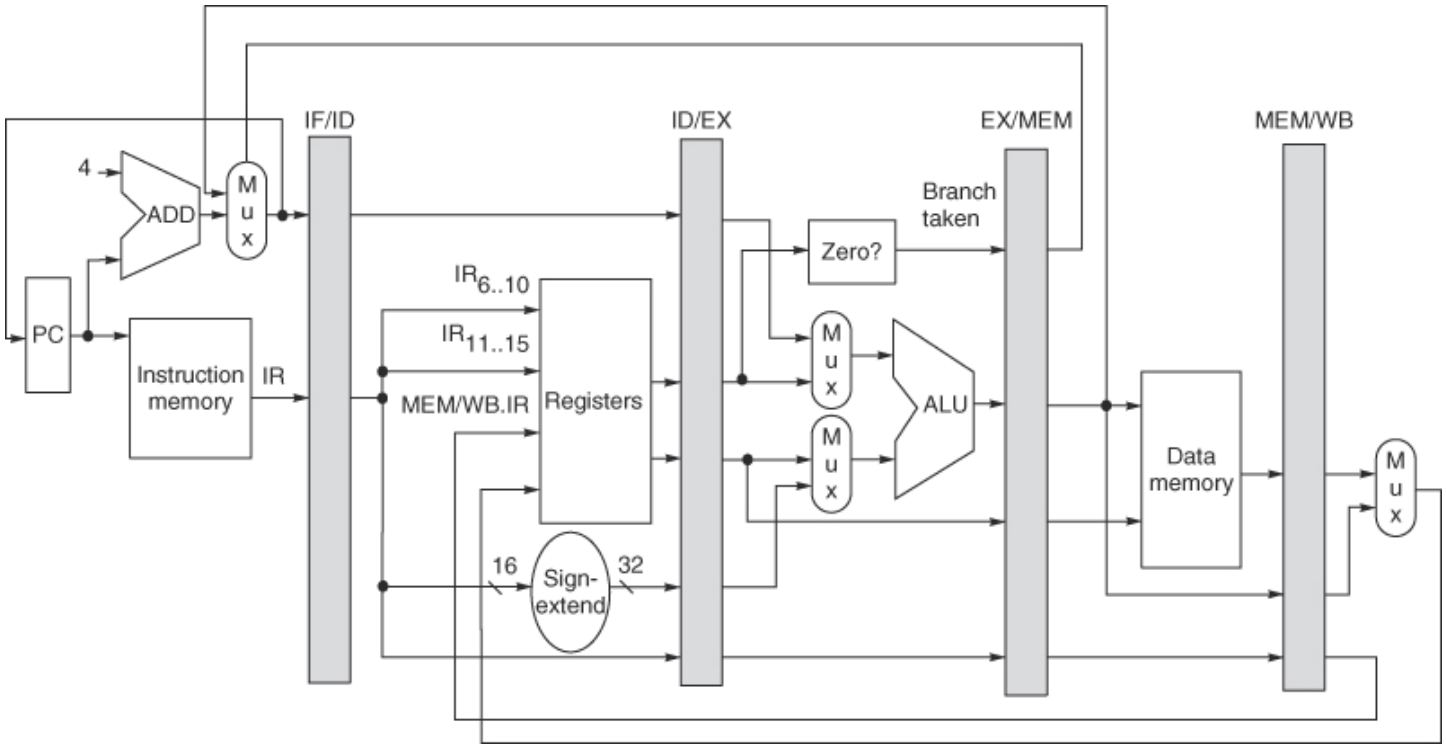

Figure C.22 The data path is pipelined by adding a set of registers, one between each pair of pipe stages. The registers serve to convey values and control information from one stage to the next. We can also think of the PC as a pipeline register, which sits before the IF stage of the pipeline, leading 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, since 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.

2

- Using just one temporary register as in the unpipelined data path could cause values to be overwritten before all uses were completed
- For example, the field for a register operand for a write in a word or ALU operation comes from MEM/WB rather than IF/ID
- Any actions taken on behalf of an instruction occur between a pair of pipeline registers
- Figure C23 shows pipeline stage activities for various instruction types
- Actions in stages 1 and 2 are independent of instruction type as the instruction has not been decoded yet

MIPS pipeline IF & ID stage events

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[rs]]; ID/EX.B ← Regs[IF/ID.IR[rt]];
	ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR;
	ID/EX.Imm ← sign-extend(IF/ID.IR[immediate field]);

Table: Figure C.23(a): Events on stages IF and ID of the MIPS pipeline

MIPS pipeline EX, MEM and WB events

Stage	ALU Instruction	Load or Store 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 == 0);
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; Or Regs[MEM/WB.IR[rt]] ← MEM/WB.ALUOutput;	For load only: Regs [MEM/WB.IR[rt]] ← MEM/WB.LMD;	

Table: Figure C.23(b): Events on stages EX, MEM and WB of the MIPS pipeline

Figure C.23: 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.22, where the PC is updated in IF from one of two sources
- In ID, we fetch the registers, extend the sign of the lower 16 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

MIPS pipeline control

- To control the simple pipeline, control the four multiplexers in the data path of Figure C22
- IF stage multiplexer controlled by EX/MEM.cond field
  - Chooses either PC+4 or EX/MEM.ALUOutput (the branch target) to write to the PC
- ALU stage multiplexers controlled by ID/EX.IR field
  - Top multiplexer set by whether or not instruction is a branch
  - Lower multiplexer set by whether or not instruction type is reg-reg ALU
- WB stage multiplexer controlled by whether instruction is Load or ALU operation
- Fifth multiplexer not shown in Figure C22
- Refer to Figure A22 for instruction formats

MIPS instruction formats

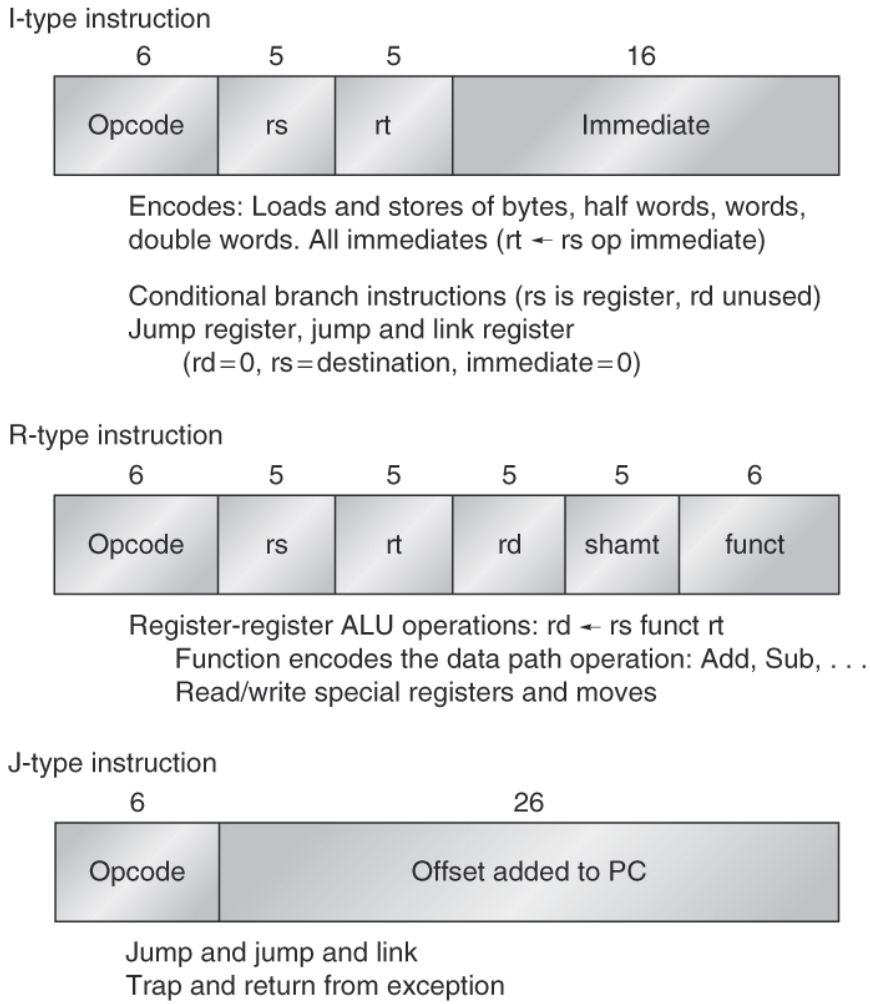

Figure A.22 Instruction layout for MIPS. All instructions are encoded in one of three types, with common fields in the same location in each format.

Write-back stage: the 5th multiplexer

- The destination field for WB is in a different place depending on instruction type
- Reg-reg ALU: $rd \leftarrow rx\ funct\ rt$
  Rd in bit positions 16-20 (counting from left)
- ALU immediate: $rt \leftarrow rs\ Op\ immediate$
  Rt in bit positions 11-15 (counting from left)
- Reg-reg ALU: $rt \leftarrow mem[rs + immediate]$
  Rt in bit positions 11-15 (counting from left)
- The fifth multiplexer is needed to select either *rd* or *rt* as the specifier field for the register destination.

Implementing control for the MIPS pipeline

- *Instruction issue*: letting an instruction move from ID to EX of a pipeline
- An instruction that has passed from ID to EX is said to have *issued*
- In MIPS pipeline, all data hazards can be checked during ID
- If a hazard exists, the instruction is stalled before it is issued
- Any forwarding necessary can be determined during ID
- Detecting interlocks early in the pipeline reduces hardware complexity since the hardware never has to suspend an instruction that has updated the state of the processor unless the entire processor has stalled
- An alternative approach is to detect the hazard/forwarding at the beginning of a clock cycle that uses an operand (EX and MEM for this pipeline)

Example: Different approaches

1. Interlock for read after write (RAW) hazard
   - Source from load instruction (*load interlock*);
   - Check in ID
2. Forwarding paths to ALU inputs
   - Do during EX

Figure C24 shows different circumstances that must be handled

MIPS pipeline hazard detection comparisons

Situation	Example code sequence	Action
No dependence	LD R1,45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9,R6,R7	No hazard possible because no dependence exists on R1 in the immediately following three instructions
Dependence requiring stall	LD R1,45(R2) DADD R5,R1,R7 DSUB R8,R6,R7 OR R9,R6,R7	Comparators detect the use of R1 in the DADD and stall the DADD (and DSUB and OR) before the DADD begins EX
Dependence overcome by forwarding	LD R1,45(R2) DADD R5,R6,R7 DSUB R8,R1,R7 OR R9,R6,R7	Comparators detect use of R1 in DSUB and forward result of load to ALU in time for DSUB to begin EX
Dependence with accesses in order	LD R1,45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9,R1,R7	No action required because the read of R1 by OR occurs in the second half of the 1D phase, while the write of the loaded data occurred in the first half

Table: Figure C.24: Situations that the pipeline hazard detection hardware can see by comparing the destination and sources of adjacent instructions

Figure C.24: Legend

- 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
- Of course, hazards that involve R0 can be ignored since the register always contains 0, and the test above could be extended to do this

Load Interlock

- RAW hazard with source instruction = Load
- Load instruction is in EX when instruction that needs the data is in ID
- All possible hazards can be described in a small table that can translate directly to an implementation
- Figure C25 shows a table that detects all load interlocks when the instruction using the load result is in ID

MIPS pipeline hazard detection comparisons

Opcode field of ID/EX ($ID/EX.IR_{0..5}$)	**Opcode field of IF/ID** ($IF/ID.NR_{0..5}$)	**Matching operand fields**
Load	Register-register ALU	ID/EX.IR[rt] == IF/ID.IR[rs]
Load	Register-register ALU	ID/EX.IR[rt] == IF/ID.IR[rt]
Load	Load, store, ALU immediate, or branch	ID/EX.IR[rt] == IF/ID.IR[rs]

Table: Figure C.25: The logic to detect the need for load interlocks during the ID stage of an instruction requires three comparisons

Figure C.25: Legend

- The logic to detect the need for load interlocks during the ID stage of an instruction requires three comparisons
- Lines 1 and 2 of the table test whether the load destination register is one of the source registers for a register-register operation in ID
- Line 3 of the table determines if the load destination register is a source for a load or store effective address, an ALU immediate, or a branch test
- 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

After hazard has been detected

- Control unit must insert the stall and prevent instruction in IF and ID from advancing
- All control information is carried in the pipeline registers
- The instruction itself is carried and this is sufficient as all control is derived from it.
- Thus, when a hazard is detected
  1. Change the control portion of the ID/EX pipeline register to all zeros a NOP
  2. Recirculate the contents of the IF/ID registers to hold the stalled instruction
- In a pipeline with more complex hazards, apply the same ideas: detect the hazard by comparing some set of pipeline registers and shift in NOPs to prevent incorrect execution.

Forwarding Logic

- Similar to hazard treatment, but more cases to consider
- Pipeline register contain the data to be forwarded.
- Pipeline register contain source and destination register fields
- Forwarding is from ALU or data memory output to the ALU input, the data memory input, or the zero detection unit
- Can implement the forwarding by a comparison of the destination registers of the IR contained in the EX/MEM and MEM/WB registers.
- Figure C26 shows comparisons and possible forwarding when the destination of the forwarded result is an ALU input for the instruction currently in EX

MIPS pipeline forwarding comparisons (a)

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

Table: Figure C.26(a): Forwarding of data to the two ALU inputs (for the instruction in EX)

MIPS pipeline forwarding comparisons(b)

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

Table: Figure C.26(b): Forwarding of data to the two ALU inputs (for the instruction in EX)

Figure C.26: Legend

- 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.21 and in Figure C.27
- 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 DADD R1, R2, R3; DADDI R1, R1, #2; DSUB R4, R3, R1, the logic must ensure that the DSUB instruction uses the result of the DADDI instruction rather than the result of the DADD instruction
- The logic shown above 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 DADDI result will be in EX/MEM, it will be forwarded, rather than the DADD result in MEM/WB

Forwarding needs

- Comparators and combinational logic to enable forwarding path
- Enlarged multiplexers at ALU inputs
- Connections from pipeline registers used to forward results
- Figure C.27 shows relevant segments of the pipelined data path.
- MIPS hazard detection and forwarding is relatively simple
- A floating-point extension is more complicated

MIPS result forwarding

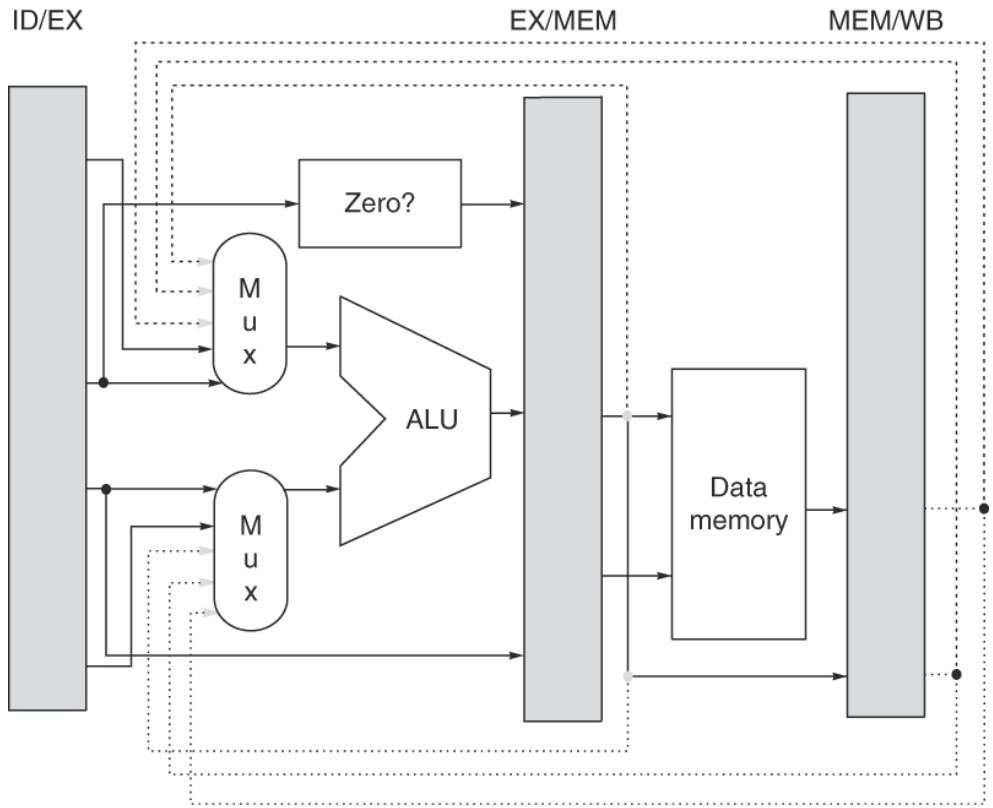

Figure C.27 Forwarding of results to the ALU requires the addition of three extra inputs on each ALU multiplexer and the addition of three paths to the new inputs. The paths correspond to a bypass of: (1) the ALU output at the end of the EX, (2) the ALU output at the end of the MEM stage, and (3) the memory output at the end of the MEM stage.

Dealing with branches in the pipeline

- BEQ, BNE $\Longrightarrow$ test register for equality to another register, which may be R0
- Consider only BEQZ and BNEZ (zero test)
- Can complete decision by end of ID by moving the zero test into that cycle
- To take advantage of an early branch decision, must compute PC and NPC early
- Extra adder needed to calculate branch-target address during ID, because ALU is not usable until EX.
- Figure C.28 shows the revised pipeline data path
- Now only 1-clock cycle stall on branches
- However, an ALU instruction followed by a branch on its result will cause a data hazard stall
- Figure C.29 shows the branch part of the revised pipeline table from Figure C.23
- Some processors have more expensive branch hazards due to longer times required for calculation of the branch condition and the destination
- For example, this can occur if there are separate decode and register fetch stages
- The branch delay (length of control hazard) can become a significant branch penalty
- In general, the deeper the pipeline, the worse the branch penalty

Reducing the branch hazard stall

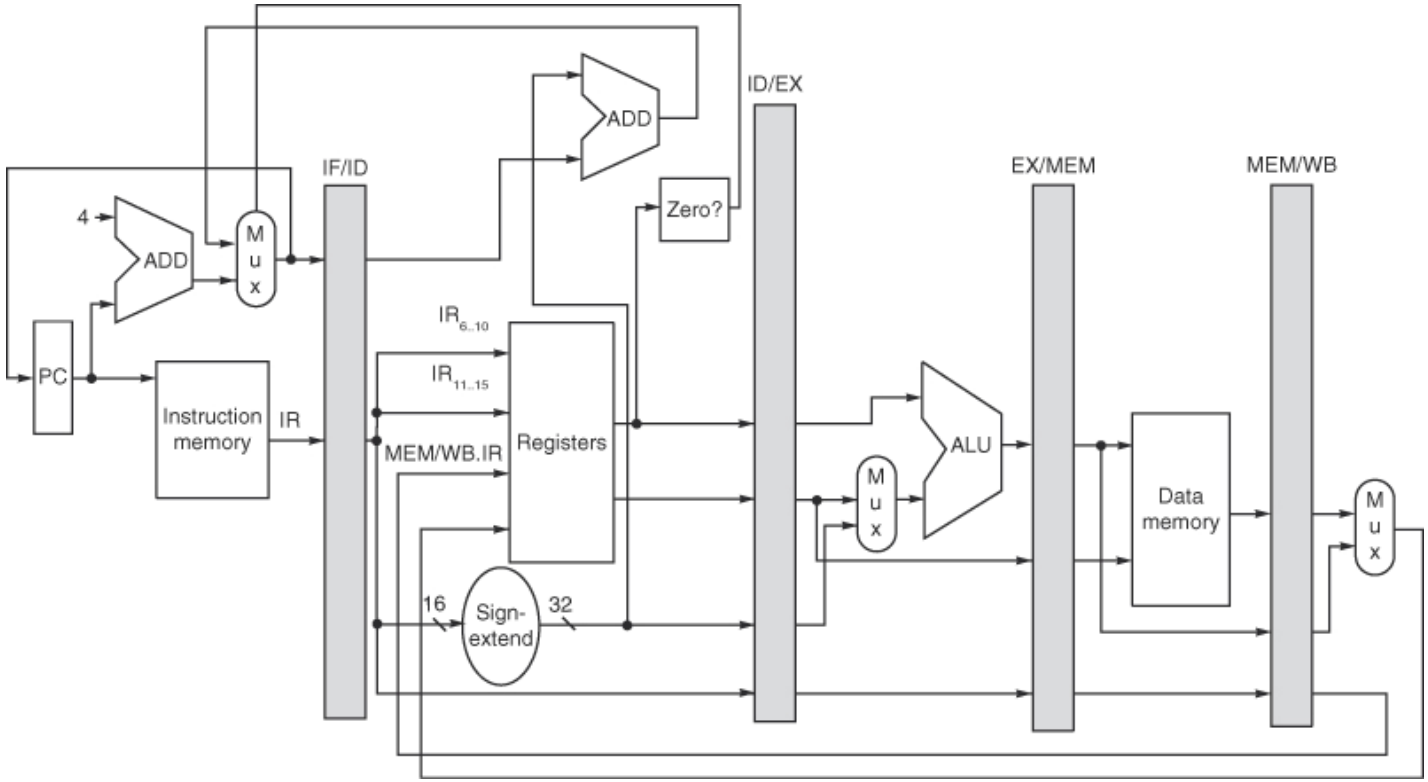

Figure C.28 The stall from branch hazards can be reduced by moving the zero test and branch-target calculation into the ID phase of the pipeline. Notice that we have made two important changes, each of which removes 1 cycle from the 3-cycle stall for branches. The first change is to move both the branch-target address calculation and the branch condition decision to the ID cycle. The second change is to write the PC of the instruction in the IF phase, using either the branch-target address computed during ID or the incremented PC computed during IF. In comparison, Figure C.22 obtained the branch-target address from the EX/MEM register and wrote the result during the MEM clock cycle. As mentioned in Figure C.22, 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.

MIPS Revised pipeline structure

Pipe stage	Branch instruction
IF	IF/ID.IR $\leftarrow$ Mem[PC]; IF/ID.NPC, PC $\leftarrow$ (if ((IF/ID.opcode == branch) & (Regs[IF/ID.$IR_{6..10}$] op 0)) {IF/ID.NPC + sign-extended (IF/ID.IR[immediate field] << 2)} else {PC + 4});
ID	ID/EX.A $\leftarrow$ Regs[IF/ID.$IR_{6..10}$]; ID/EX.B $\leftarrow$ Regs[IF/ID.$IR_{11..15}$]; ID/EX.IR $\leftarrow$ IF/ID.IR; ID/EX.Imm $\leftarrow$ (IF/ID.IR_{16})^{16}##IF/ID.$IR_{16..31}$
EX	
MEM	
WB	

Table: Figure C.29: Revised pipeline structure based on the original in Figure C.23

Figure C.29 Legend: revised pipeline structure is based on the original in Figure C.23

- It uses a separate adder, as in Figure C.28, to compute the branch-target address during ID
- The operations that are new or have changed are in bold
- Because the branch-target address addition happens during ID, it will happen for all instructions; the branch condition ($Regs[IF/ID.IR_{6..10}]$ *op* 0) will also be done for all instructions
- The selection of the sequential PC or the branch-target PC still occurs during IF, but it now uses values from the ID stage that correspond to the values set by the previous instruction
- This change reduces the branch penalty by 2 cycles:
  - one from evaluating the branch target and condition earlier
  - and one from controlling the PC selection on the same clock rather than on the next clock
- Since the value of cond is set to 0, unless the instruction in ID is a taken branch, the processor must decode the instruction before the end of ID
- Because the branch is done by the end of ID, the EX, MEM, and WB stages are unused for branches
- An additional complication arises for jumps that have a longer offset than branches
- We can resolve this by using an additional adder that sums the PC and lower 26 bits of the IR after shifting left by 2 bits

Dealing with Exceptions

- Terms *interrupt*, *fault*, *exception* widely used An exception can be due to any of the following:
  - I/O device request
  - SVC (supervisor call)
  - Trace instruction execution
  - Breakpoint
  - Integer arithmetic overflow
  - FP arithmetic anomaly
  - Page fault
  - Misaligned memory access
  - Memory protection violation
  - Undefined instruction
  - Hardware malfunction
  - Power failure
  - Figure C.30 shows common exceptions in different architectures

Names of common exceptions

Exception event	IBM 360	VAX	Motorola 680x0	Intel 80x86
I/O device request	Input/output interruption	Device interrupt	Exception (LO to L7 autovector)	Vectored interrupt
Invoking the operating system service from a user program	Supervisor call interruption	Exception (change mode supervisor trap)	Exception (unimplemented instruction) – on Macintosh	Interrupt (INT instruction)
Tracing instruction execution	Not applicable	Exception (trace fault)	Exception (trace) Interrupt (single-step trap)	
Breakpoint	Not applicable	Exception (breakpoint fault)	Exception (illegal instruction or breakpoint)	Interrupt (breakpoint trap)
Integer arithmetic overflow or underflow; FP trap	Program interruption (overflow or underflow exception)	Exception (integer overflow trap or floating underflow fault)	Exception (floating-point coprocessor errors)	Interrupt (overflow trap or math unit exception)
Page fault (not in main memory)	Not applicable (only in 370)	Exception (translation not valid fault)	Exception (memory-management unit errors)	Interrupt(page fault)

Table: Figure C.30(a): The names of common exceptions vary across four different architectures

Names of common exceptions (continued)

Exception event	IBM 360	VAX	Motorola 680x0	Intel 80x86
Misaligned memory accesses	Program interruption (specification exception)	Not applicable	Exception (address error)	Not applicable
Memory protection violations	Program interruption (protection exception)	Exception (access control violation fault)	Exception (bus error)	Interrupt (protection exception)
Using undefined instructions	Program interruption (operation exception)	Exception (opcode privileged/reserved fault)	Exception (illegal instruction or breakpoint/unimplemented instruction)	Interrupt (invalid opcode)
Hardware malfunctions	Machine-check interruption	Exception (machine-check abort)	Exception (bus error)	Not applicable
Power failure	Machine-check interruption	Urgent interrupt	Not applicable	Nonmaskable interrupt

Table: Figure C.30(b): The names of common exceptions vary across four different architectures

Figure C.30 Legend

- The names of common exceptions vary across four different architectures
- Every event on the IBM 350 and 80x86 is called an interrupt
- Every event on the 680x0 is called an exception
- VAX divides events into interrupts or exceptions. The adjectives device, software, and urgent are used with VAX interrupts, whereas VAX exceptions are subdivided into faults, traps, and aborts.

Exception requirements

1. Synchronous vs Asynchronous
   - If the event occurs at the same place every time, the program executes with the same data and memory allocation, it is synchronous
   - Asynchronous events (excluding hardware malfunction) are caused by devices external to CPU and main memory
   - Asynchronous events can be handled after the current instruction completes
2. User-requested vs coerced
   - User-requested exceptions are treated as exceptions because the exception mechanism is used for their initial processing
   - The only function of the instruction is to cause the exception, so user-requested exceptions can be handled after the instruction completes
   - Coerced exceptions are caused by a hardware event not under program control, are unpredictable and harder to implement

Exception requirements (continued)

3. User maskable vs nonmaskable
   - The mask controls whether the hardware responds to the interrupt or not
4. Within vs between instructions: When is exception recognised
   - Events that occur within instructions are usually synchronous
   - The instruction must be stopped and restarted
   - Asynchronous exceptions within instructions are catastrophic and cause program termination

Exception requirements (continued)

5 Rename vs terminate

- Renaming event: program executions continues after interrupt
- Terminating event: program execution stops after interrupt

Figure C.31 displays the example exceptions according to these five categories

Actions needed for different exception types

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

Table: Figure C.31(a): Five categories are used to define what actions are needed for the different exception types shown in Figure C.30

Actions needed for different exception types (continued)

Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
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

Table: Figure C.31(b): Five categories are used to define what actions are needed for the different exception types shown in Figure C.30

Figure C.31 Legend

- Five categories are used to define what actions are needed for the different exception types shown in Figure C.30
- Exceptions that most 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, CPUs should be able to resume after such exceptions

Exception requirements (continued)

- Pipeline provides ability for processor to handle exception, save state, and restart without affecting program execution $\Longrightarrow$ pipeline is *restartable*
- All processors support this feature now, as it is needed to implement virtual memory
- Stopping and restarting execution
- The most difficult exceptions have two properties
  1. They occur within instructions (i.e., in the middle of the execution, corresponding to EX or MEM)
  2. They must be restartable

Example

- Virtual memory page fault from data fetch cannot occur until MEM. By that time, several other instructions will be in execution
- A page fault must be restartable
- The pipeline must be shut down and state saved
- PC of restart instruction must be saved
- If restarted instruction is not a branch, continue to fetch sequential successors and begin their execution
- If restarted instruction is a branch, re-evaluate the condition and begin fetching from Instruction i+1 or branch target

Dealing with pipeline state when an exception occurs

- On exception, save pipeline state
  1. Force a trap instruction into the pipeline on the next IF
  2. Until the trap, turn off all writes for the faulting instruction and for all instructions that follow in the pipeline
     - This can be done by putting zeros in the pipeline registers of all instruction 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 complete before the exception is handled
  3. After the OS exceptions handler gets control, it saves the PC of the faulting instruction
- When delayed branches are used, instructions in the pipeline may not be sequentially related
- A number of PCs must be saved equal to the length of the branch delay plus one. This is done in step three.
- After handling the exception, reload PCs and restart the instruction stream (MIPS RFE instruction)

Precise exceptions

- Precise exceptions: Pipeline is stopped so that instructions just before the faulting instruction is completed and those after it are restarted from scratch
- Ideally, faulting instruction has not changed the state
- Correct handling of some exceptions requires that the faulting instruction has no effects
- For integer pipelines, precise exceptions are easier and are generally provided
- Precise exceptions for memory references are necessary to support virtual memory
- Floating-point operations present challenges to the implementation of precise exceptions

Exceptions in MIPS

- Figure C.32 lists the potential exceptions for MIPS pipeline stages
- Pipelining $\Longrightarrow$ multiple exceptions in the same clock cycle because there are several instructions in execution
- Example

	Clock cycle					
Instr No	**1**	**2**	**3**	**4**	**5**	**6**
LD	IF	ID	EX	MEM	WB	
DADD		IF	ID	EX	MEM	WB

- This instruction pair can cause a data page fault and an arithmetic exception at the same time, since LD is in MEM when DADD is in EX
- This can be handled by dealing with only the page fault and restarting the instruction. The second exception will reoccur and can be handled independently
- Exceptions can occur out of order: An instruction can cause an exception before an earlier instruction causes one
- For instance, in the LD/DADD example, an instruction page fault could be caused by DADD in IF before a data page fault caused by LD in MEM
- To implement precise exceptions, the pipeline must handle the LD exception first
- A pipeline cannot handle an exception when it occurs in time, as exceptions may occur out of unpipelined order

Exceptions that may occur in the MIPS pipeline

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
WE	None

Table: Figure C.32: Exceptions that may occur in the MIPS pipeline. Exceptions raised from instruction or data memory access account for six out of eight cases

Method

- Exceptions caused by instruction I_i are posted in a status vector associated with that segment
- The exception status vector accompanies I_i as it passes through the pipeline
- Once an exception indication is set in the exception status vector, any control signal that may cause a data value to be written is turned off
- This includes register writes and memory writes
- Because a store can cause an exception during MEM, it must be prevented from completing if it raises an exception

- As an instruction passes the MEM/WB interface, its exception status vector is checked
- Exceptions that have been posted in the status vector are handled in the order in which they would occur in time in an unpipelined processor
- The exception corresponding to the earliest instruction (and usually the earliest pipe stage for that instruction) is handled first
- This guarantees that I_i exceptions are seen before those of I_{i+1}
- Earlier pipe stage actions for I_i may be invalid but not state could have changed because memory and register file writes are disabled