CS230 Notes (Quiz-2)

Soham Joshi

March 2023

§1 Summary

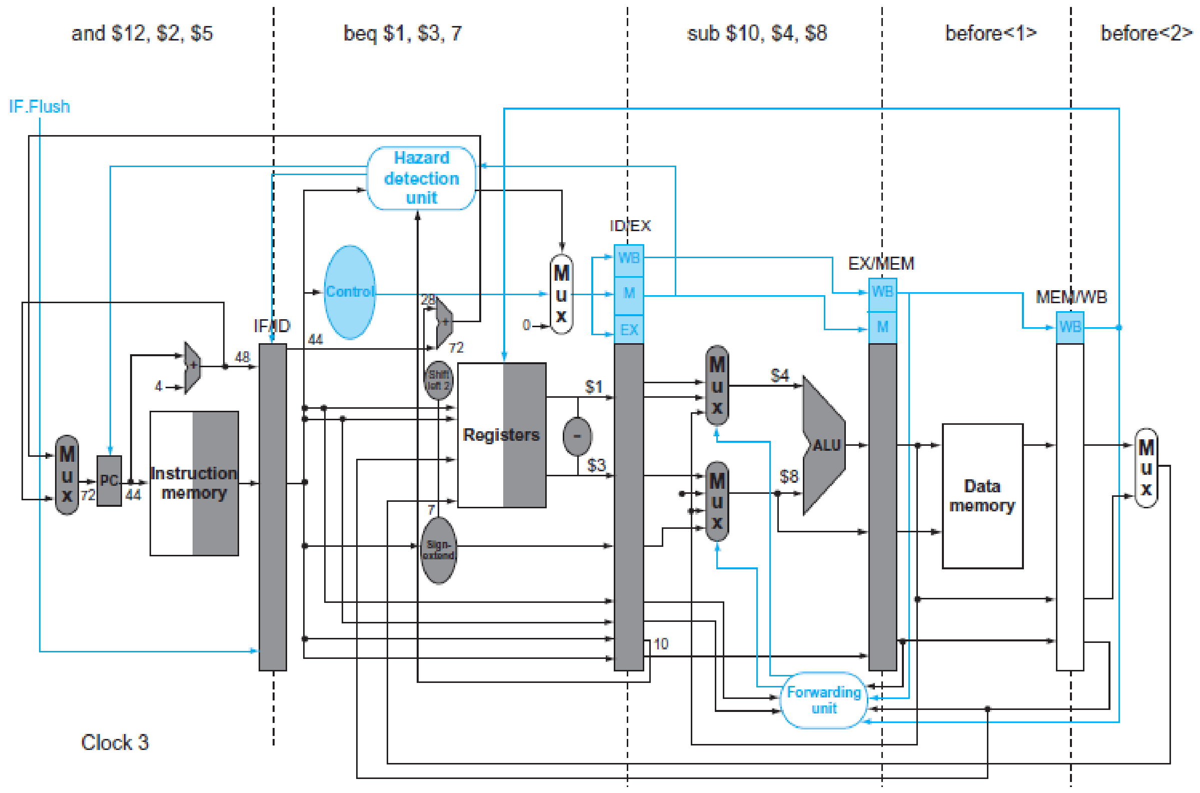
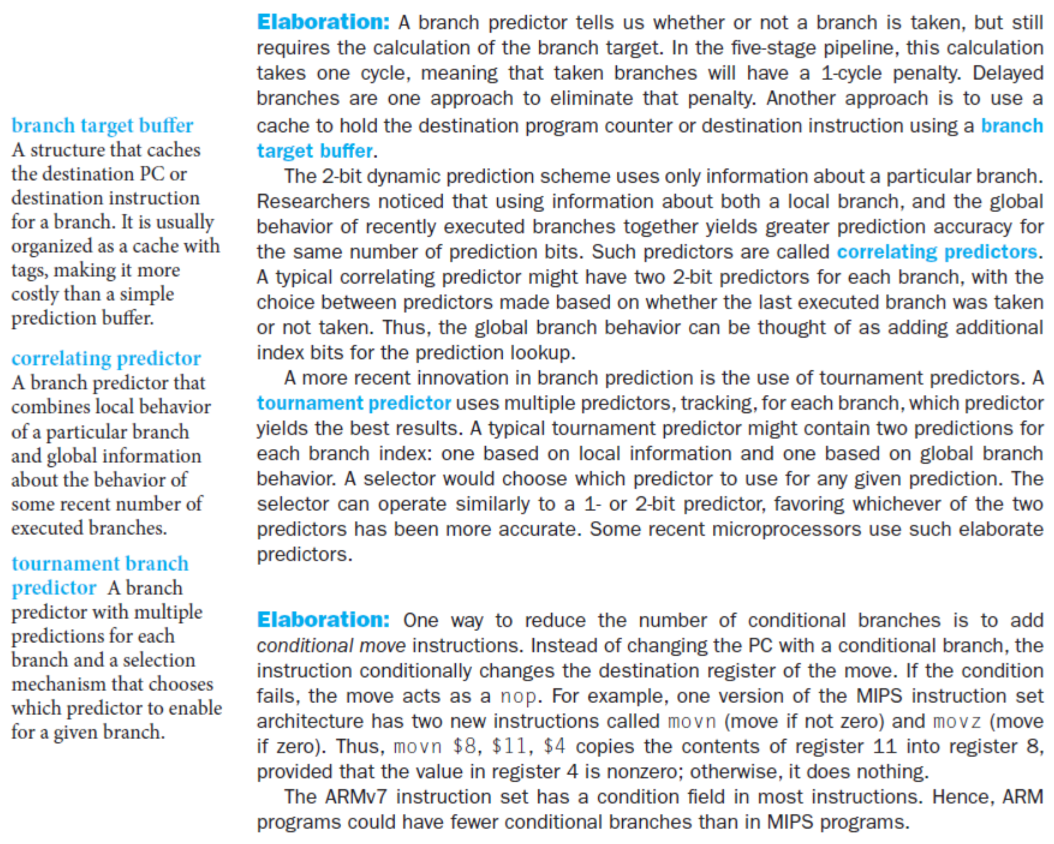


Figure 1: Shifting branch decision to ID stage



branch target buffer A structure that caches the destination PC or destination instruction for a branch. It is usually organized as a cache with tags, making it more costly than a simple prediction buffer.

correlating predictor A branch predictor that combines local behavior of a particular branch and global information about the behavior of some recent number of executed branches.

tournament branch predictor A branch predictor with multiple predictions for each branch and a selection mechanism that chooses which predictor to enable for a given branch.

Elaboration: A branch predictor tells us whether or not a branch is taken, but still requires the calculation of the branch target. In the five-stage pipeline, this calculation takes one cycle, meaning that taken branches will have a 1-cycle penalty. Delayed branches are one approach to eliminate that penalty. Another approach is to use a cache to hold the destination program counter or destination instruction using a **branch target buffer**.

The 2-bit dynamic prediction scheme uses only information about a particular branch. Researchers noticed that using information about both a local branch, and the global behavior of recently executed branches together yields greater prediction accuracy for the same number of prediction bits. Such predictors are called **correlating predictors**. A typical correlating predictor might have two 2-bit predictors for each branch, with the choice between predictors made based on whether the last executed branch was taken or not taken. Thus, the global branch behavior can be thought of as adding additional index bits for the prediction lookup.

A more recent innovation in branch prediction is the use of tournament predictors. A **tournament predictor** uses multiple predictors, tracking, for each branch, which predictor yields the best results. A typical tournament predictor might contain two predictions for each branch index: one based on local information and one based on global branch behavior. A selector would choose which predictor to use for any given prediction. The selector can operate similarly to a 1- or 2-bit predictor, favoring whichever of the two predictors has been more accurate. Some recent microprocessors use such elaborate predictors.

Elaboration: One way to reduce the number of conditional branches is to add *conditional move* instructions. Instead of changing the PC with a conditional branch, the instruction conditionally changes the destination register of the move. If the condition fails, the move acts as a `nop`. For example, one version of the MIPS instruction set architecture has two new instructions called `movn` (move if not zero) and `movz` (move if zero). Thus, `movn $8, $11, $4` copies the contents of register 11 into register 8, provided that the value in register 4 is nonzero; otherwise, it does nothing.

The ARMv7 instruction set has a condition field in most instructions. Hence, ARM programs could have fewer conditional branches than in MIPS programs.

Figure 2: Branch predictors (types)

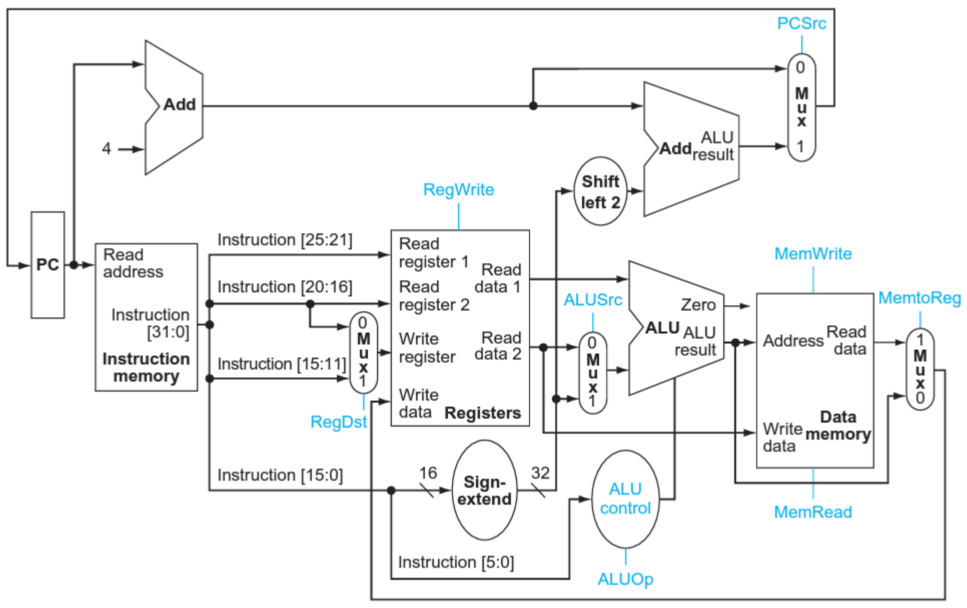
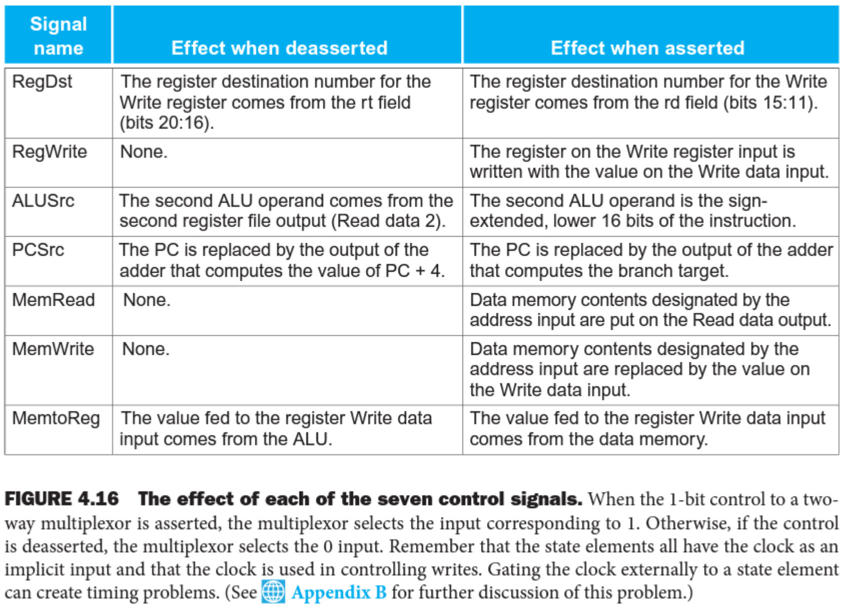


FIGURE 4.15 The datapath of Figure 4.11 with all necessary multiplexors and all control lines identified. The control lines are shown in color. The ALU control block has also been added. The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address.

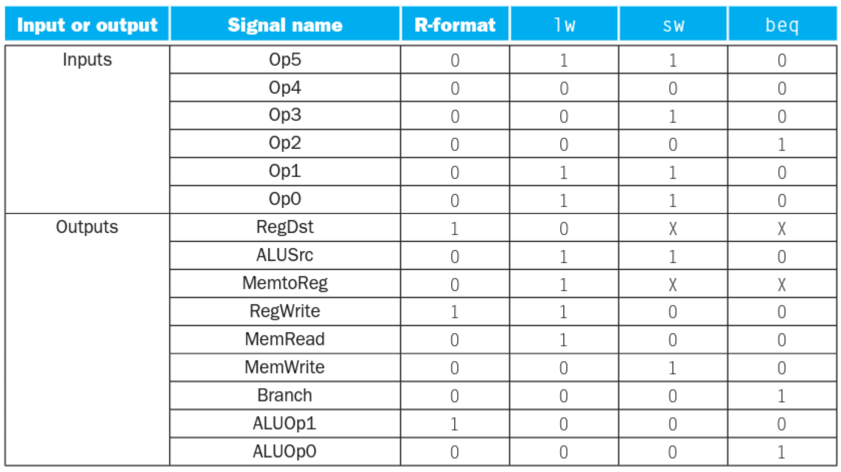
Figure 3: Single cycle CPU



Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

FIGURE 4.16 The effect of each of the seven control signals. When the 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes. Gating the clock externally to a state element can create timing problems. (See Appendix B for further discussion of this problem.)

Figure 4: Control signals for pipeline



Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Figure 5: Control signals for pipeline (values)

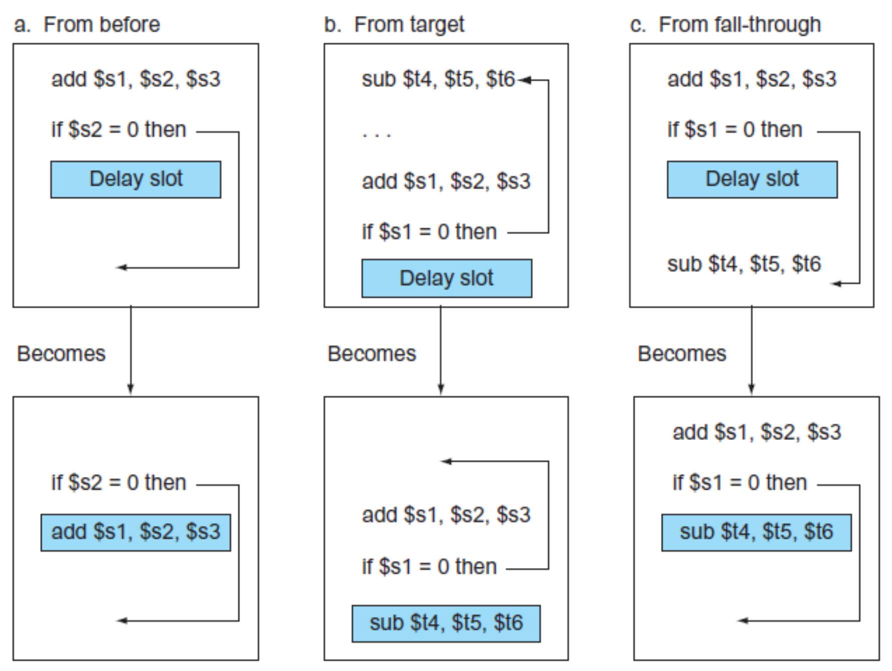
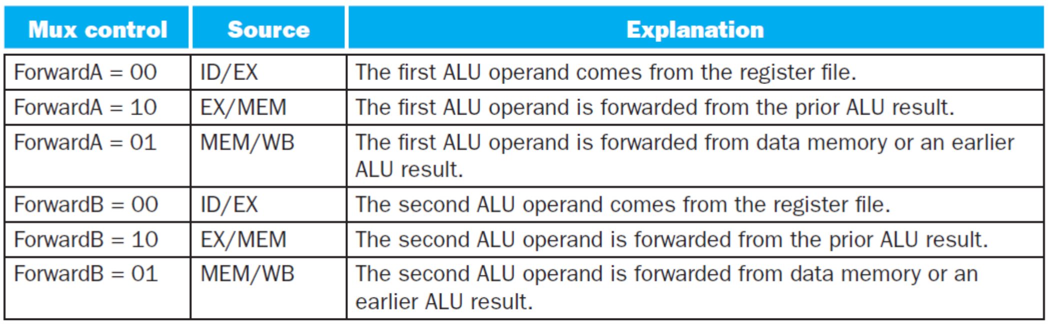


FIGURE 4.64 Scheduling the branch delay slot. The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code. In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) is not possible. In the code sequences for (b) and (c), the use of `$s1` in the branch condition prevents the `add` instruction (whose destination is `$s1`) from being moved into the branch delay slot. In (b) the branch delay slot is scheduled from the target of the branch; usually the target instruction will need to be copied because it can be reached by another path. Strategy (b) is preferred when the branch is taken with high probability, such as a loop branch. Finally, the branch may be scheduled from the not-taken fall-through as in (c). To make this optimization legal for (b) or (c), it must be OK to execute the `sub` instruction when the branch goes in the unexpected direction. By "OK" we mean that the work is wasted, but the program will still execute correctly. This is the case, for example, if `$t4` were an unused temporary register when the branch goes in the unexpected direction.

Figure 6: Delay slots as a method for branch control hazard mitigation



Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

FIGURE 4.55 The control values for the forwarding multiplexors in Figure 4.54. The signed immediate that is another input to the ALU is described in the *Elaboration* at the end of this section.

Figure 7: Forwarding (mux values)

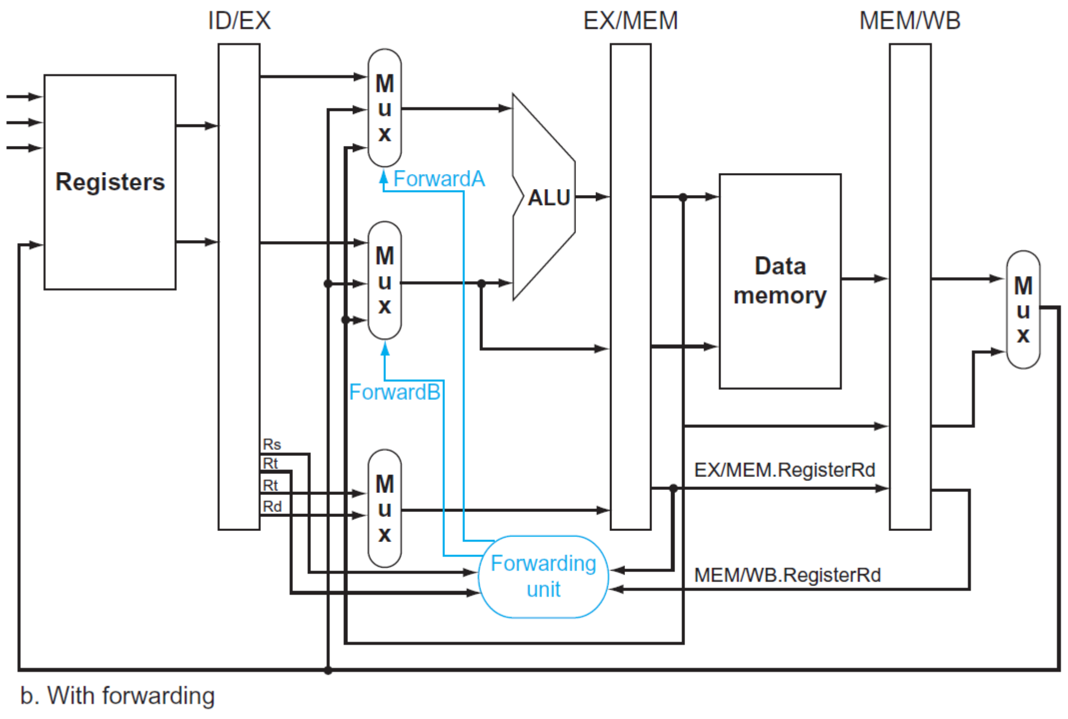


Figure 8: Forwarding unit for a pipeline

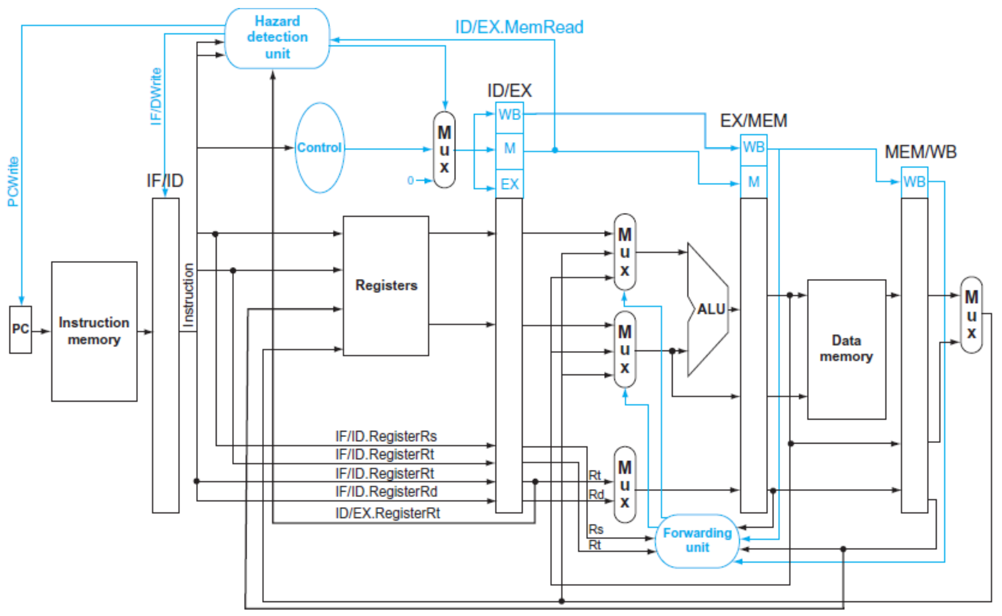
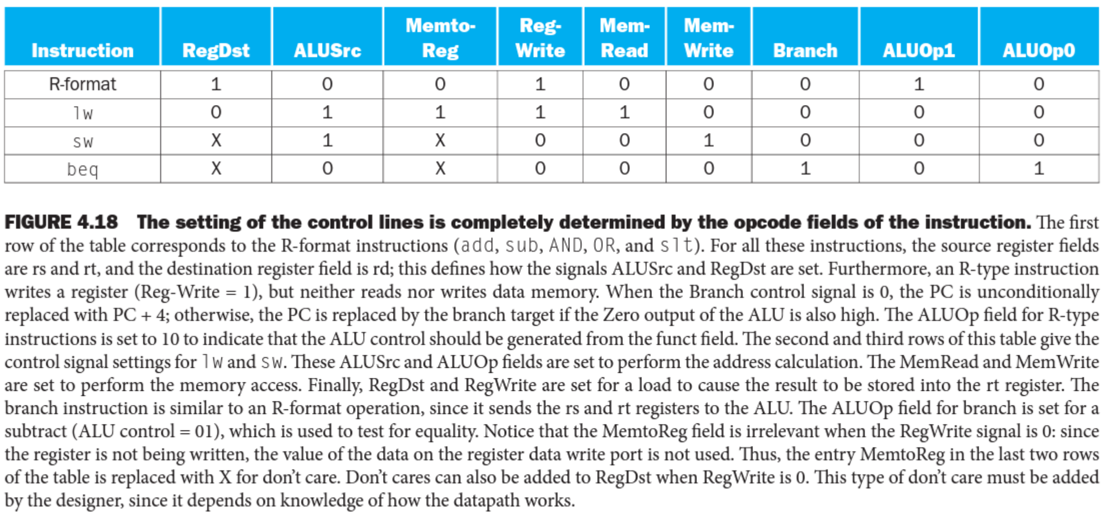


FIGURE 4.60 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements.

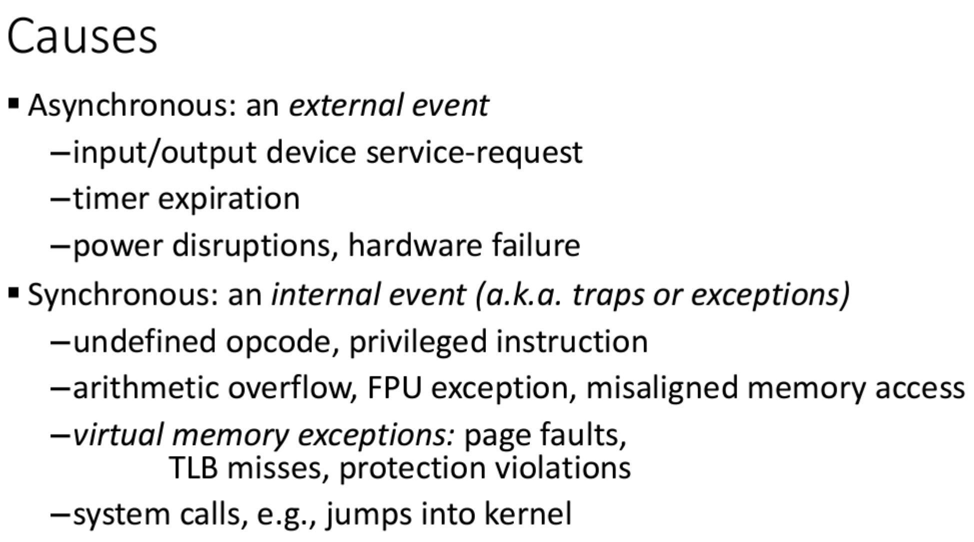
Figure 9: Hazard and forwarding unit for a pipeline



Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

FIGURE 4.18 The setting of the control lines is completely determined by the opcode fields of the instruction. The first row of the table corresponds to the R-format instructions (`add`, `sub`, `AND`, `OR`, and `slt`). For all these instructions, the source register fields are rs and rt, and the destination register field is rd; this defines how the signals ALUSrc and RegDst are set. Furthermore, an R-type instruction writes a register (Reg-Write = 1), but neither reads nor writes data memory. When the Branch control signal is 0, the PC is unconditionally replaced with PC + 4; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high. The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct field. The second and third rows of this table give the control signal settings for `lw` and `sw`. These ALUSrc and ALUOp fields are set to perform the address calculation. The MemRead and MemWrite are set to perform the memory access. Finally, RegDst and RegWrite are set for a load to cause the result to be stored into the rt register. The branch instruction is similar to an R-format operation, since it sends the rs and rt registers to the ALU. The ALUOp field for branch is set for a subtract (ALU control = 01), which is used to test for equality. Notice that the MemtoReg field is irrelevant when the RegWrite signal is 0: since the register is not being written, the value of the data on the register data write port is not used. Thus, the entry MemtoReg in the last two rows of the table is replaced with X for don't care. Don't cares can also be added to RegDst when RegWrite is 0. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

Figure 10: Instruction type to control signals in a MIPS instruction set



Causes

- Asynchronous: an *external event*
  - –input/output device service-request
  - –timer expiration
  - –power disruptions, hardware failure
- Synchronous: an *internal event (a.k.a. traps or exceptions)*
  - –undefined opcode, privileged instruction
  - –arithmetic overflow, FPU exception, misaligned memory access
  - –*virtual memory exceptions:* page faults, TLB misses, protection violations
  - –system calls, e.g., jumps into kernel

Figure 11: Causes of interrups

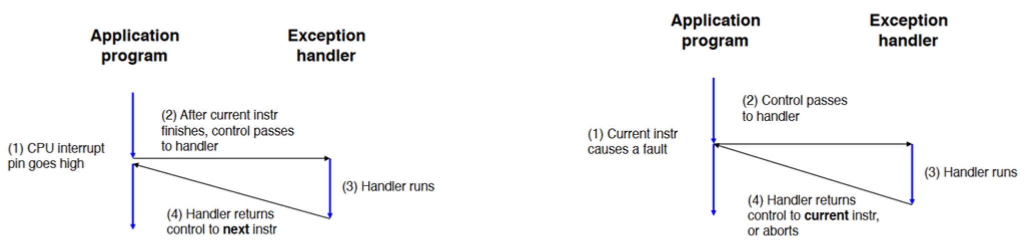
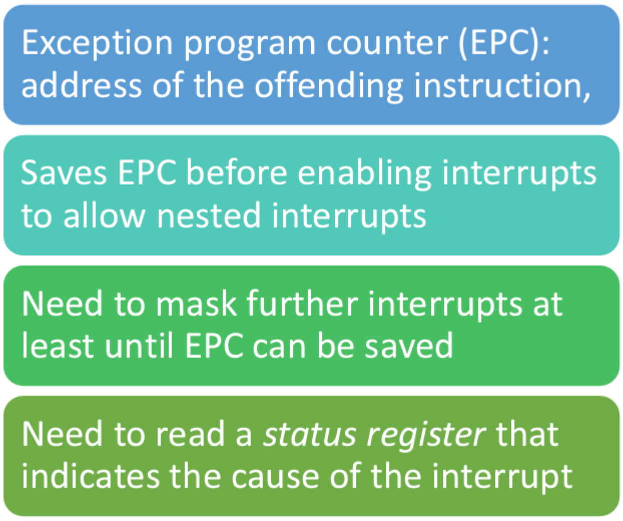

Application program
Exception handler
(1) CPU interrupt pin goes high
(2) After current instr finishes, control passes to handler
(3) Handler runs
(4) Handler returns control to **next** instr
Application program
Exception handler
(1) Current instr causes a fault
(2) Control passes to handler
(3) Handler runs
(4) Handler returns control to **current** instr, or aborts

Figure 12: Interrupt handling for a program



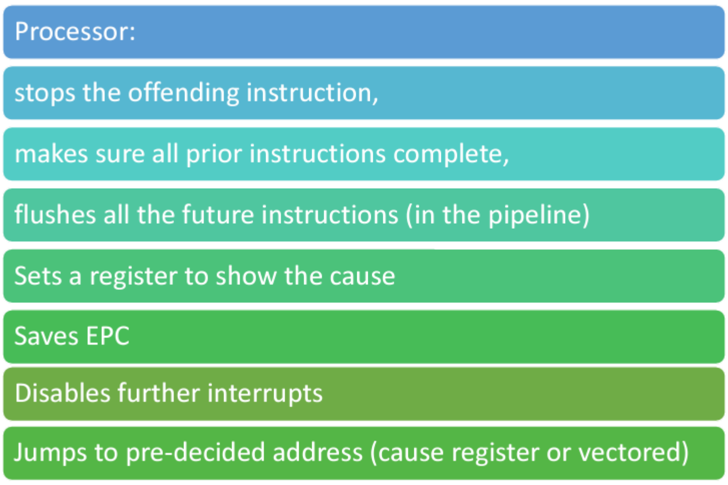
Exception program counter (EPC): address of the offending instruction,

Saves EPC before enabling interrupts to allow nested interrupts

Need to mask further interrupts at least until EPC can be saved

Need to read a *status register* that indicates the cause of the interrupt

Figure 13: Exception program counter (interrupt handler)



Processor:

stops the offending instruction,

makes sure all prior instructions complete,

flushes all the future instructions (in the pipeline)

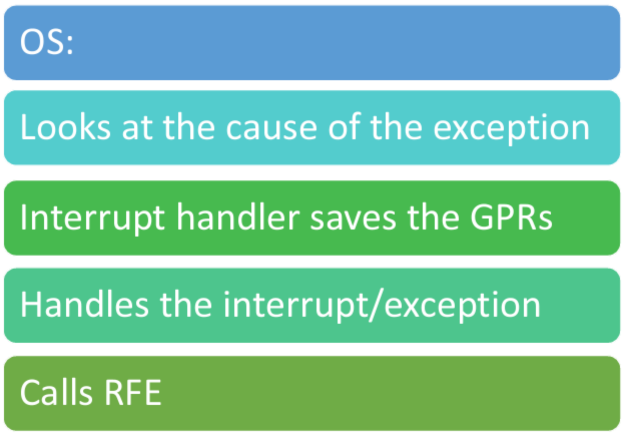
Sets a register to show the cause

Saves EPC

Disables further interrupts

Jumps to pre-decided address (cause register or vectored)

Figure 14: Processor's role (interrupt handler)



OS:

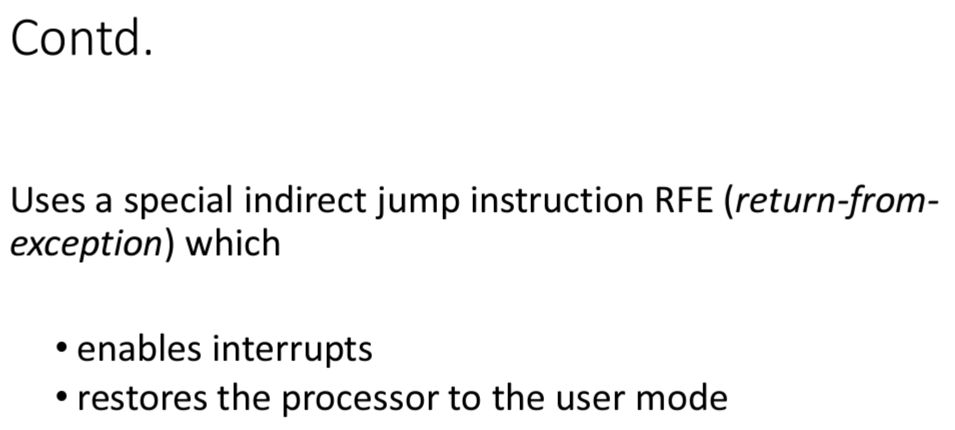
Looks at the cause of the exception

Interrupt handler saves the GPRs

Handles the interrupt/exception

Calls RFE

Figure 15: OS's role (interrupt handler)



Contd.

Uses a special indirect jump instruction RFE (*return-from-exception*) which

- enables interrupts
- restores the processor to the user mode

Figure 16: RFE instruction for exception handling (interrupt handler)

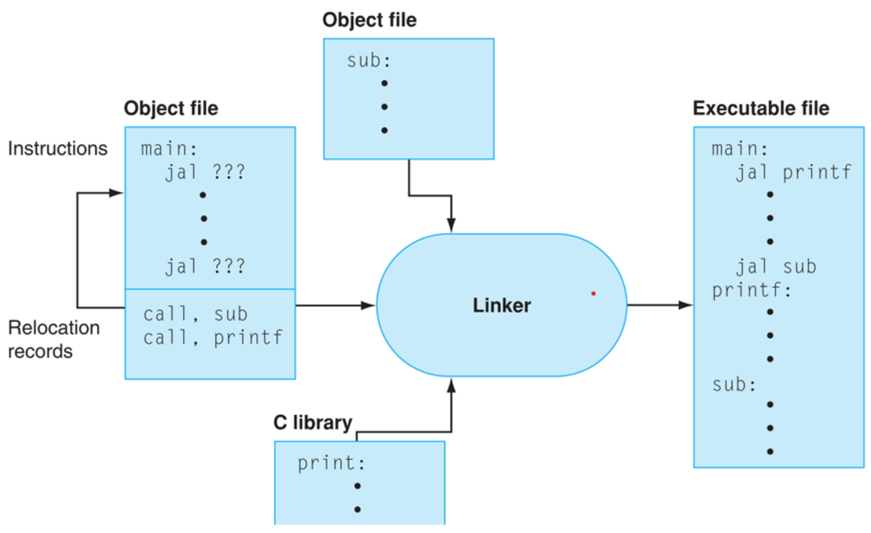


Figure 17: Linker (diagrammatic depiction)

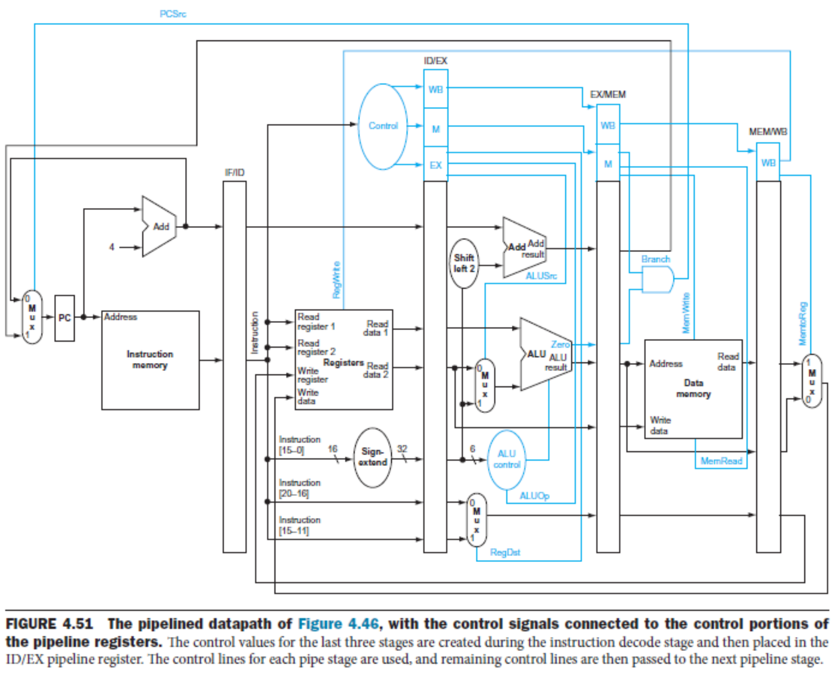
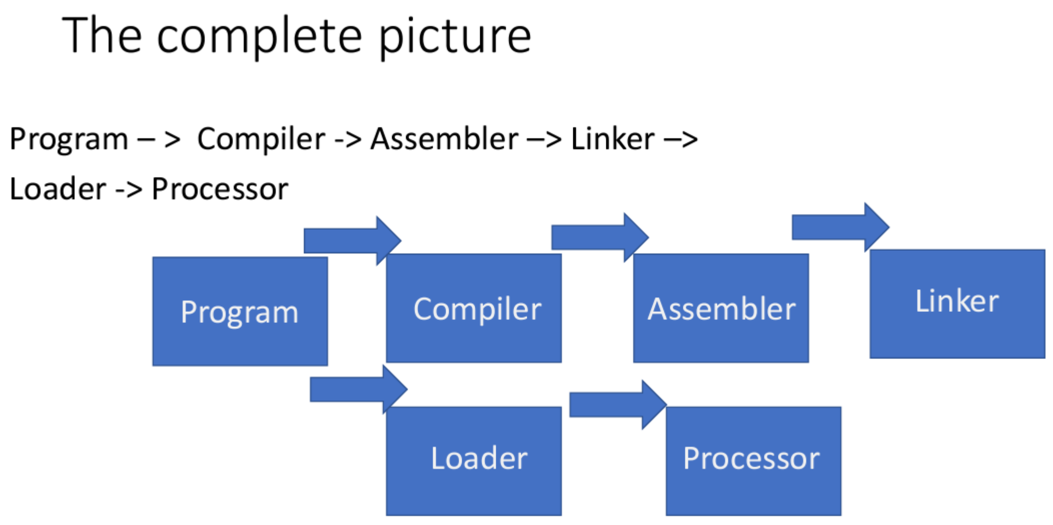


FIGURE 4.51 The pipelined datapath of Figure 4.46, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

Figure 18: A pipeline with all control signals



The complete picture

Program – > Compiler -> Assembler –> Linker –>
Loader -> Processor

Figure 19: The complete path of “agents (programs/components)” from a program to an executable

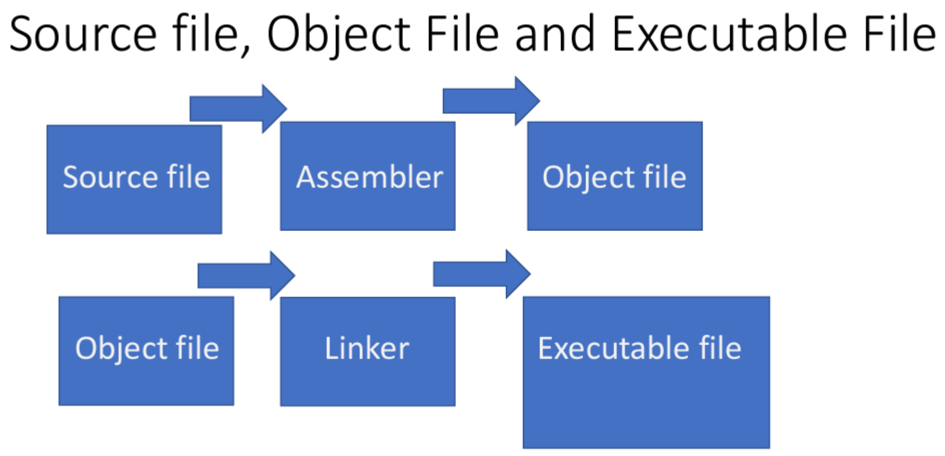


Figure 20: The path of files from source file to executable