Reduced Instruction Set Architecture (RISC) (Week#9 and 10)

RISC-V

Registers: **General Purpose Registers:** RV64G has 32 64-bit general-purpose registers (GPRs), named x0, x1, … , x31. GPRs are also sometimes known as integer registers.

Floating Point Registers: F and D extensions for floating point that are part of RV64G, come a set of 32 floating point registers (FPRs), named f0, f1, … , f31, which can hold 32 single-precision (32-bit) values or 32 double-precision (64-bit) values. (When holding one single precision number, the other half of the FPR is unused.) Both single- and double precision floating-point operations (32-bit and 64-bit) are provided. The value of x0 is always 0. Few special registers can be transferred to and from the general-purpose registers.

Example is floating-point status register, used to hold information about results of floating-point operations. There are also instructions for moving between an FPR and a GPR.

Data Types: data types are 8-bit bytes, 16-bit half words, 32-bit words and 64-bit doublewords for integer data and 32-bit single precision and 64-bit double precision for floating point. Half words were added because they are found in languages like C and are popular in some programs, such as the operating systems.

RV64G operations work on 64-bit integers and 32- or 64-bit floating point. Bytes, half words, and words are loaded into the general-purpose registers with either zeros or the sign bit replicated to fill the 64 bits of the GPRs. Once loaded, they are operated with the 64-bit integer operations.

RISC-V

Addressing Modes for RISC-V Data Transfers: O nly data addressing modes are immediate and displacement both with 12-bit fields.

Register indirect is accomplished simply by placing 0 in the 12-bit displacement field, and limited absolute addressing with a 12-bit field is accomplished by using register 0 as the base register.

Embracing zero gives us four effective modes, although only two are supported in the architecture.

RV64G memory is byte addressable with a 64-bit address and uses Little Endian byte numbering.

It is a load-store architecture, all references between memory and either GPRs or FPRs are through loads or stores.Memory accesses involving GPRs can be a byte, half word, word, or double word.

FPRs may be loaded and stored with single-precision or double-precision numbers. Memory accesses need not be aligned; it may be unaligned accesses run extremely slow.

RISC-V

Instruction Formats: RISC-V has two addressing modes, these can be encoded into opcode. Making processor easy to pipeline and decode, all instructions are 32 bits with a 7-bit primary opcode.

Figure A.23 shows the instruction layout of the four major instruction types. These formats are simple while providing 12-bit fields for displacement addressing, immediate constants, or PC-relative branch addresses

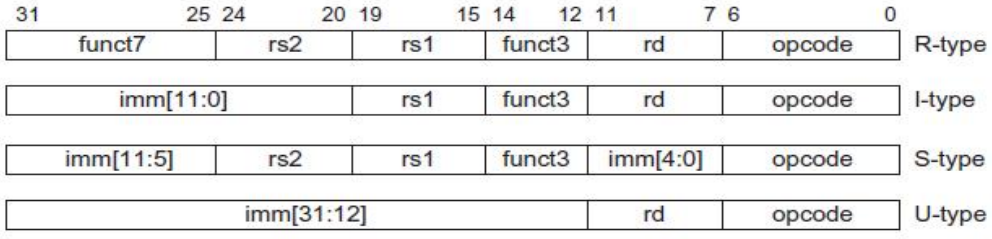

Figure A.23 The RISC-V instruction layout. There are two variations on these formata, called the SB and UJ formats; they deal with a slightly different treatment for immediate fields.

RISC-V

Instruction format	Primary use	rd	rs1	rs2	Immediate
R-type	Register-register ALU instructions	Destination	First source	Second source	
I-type	ALU immediates Load	Destination	First source base register		Value displacement
S-type	Store Compare and branch		Base register first source	Data source to store second source	Displacement offset
U-type	Jump and link Jump and link register	Register destination for return PC	Target address for jump and link register		Target address for jump and link

Figure A.24 The use of instruction fields for each instruction type. Primary use shows the major instructions that use the format. A blank indicates that the corresponding field is not present in this instruction type. The I-format is used for both loads and ALU immediates, with the 12-bit immediate holding either the value for an immediate or the displacement for a load. Similarly, the S-format encodes both store instructions (where the first source register is the base register and the second contains the register source for the value to store) and compare and branch instructions (where the register fields contain the sources to compare and the immediate field specifies the offset of the branch target). There are actually two other formats: SB and UJ that follow the same basic organization as S and J, but slightly modify the interpretation of the immediate fields.

RISC-V

To understand these figures we need to introduce a few additional extensions to our C description language used initially on page A-9:

- A subscript is appended to the symbol $\leftarrow$ whenever the length of the datum being transferred might not be clear. Thus, $\leftarrow_n$ means transfer an n-bit quantity. We use $x, y \leftarrow z$ to indicate that z should be transferred to x and y.
- A subscript is used to indicate selection of a bit from a field. Bits are labeled from the most-significant bit starting at 0. The subscript may be a single digit (e.g., $\texttt{Regs[x4]}_0$ yields the sign bit of x4) or a subrange (e.g., $\texttt{Regs[x3]}_{56..63}$ yields the least-significant byte of x3).
- The variable Mem, used as an array that stands for main memory, is indexed by a byte address and may transfer any number of bytes.
- A superscript is used to replicate a field (e.g., 0^{48} yields a field of zeros of length 48 bits).
- The symbol ## is used to concatenate two fields and may appear on either side of a data transfer, and the symbols $\ll$ and $\gg$ shift the first operand left or right by the amount of the second operand.

RISC-V

Operations: There are four broad classes of instructions: loads and stores, ALU operations, branches and jumps, and floating-point operations.Any general-purpose or floating-point registers may be loaded or stored, except that loading x0 has no effect. Figure A.25 gives example of load and store instructions.

Example instruction	Instruction name	Meaning
ld x1,80(x2)	Load doubleword	Regs[x1] ← Mem[80+Regs[x2]]
lw x1,60(x2)	Load word	Regs[x1] ←$_{64}$ Mem[60+Regs[x2]]$_0$)^{32} ## Mem[60+Regs[x2]]
lwu x1,60(x2)	Load word unsigned	Regs[x1] ←$_{64}$ 0^{32} ## Mem[60+Regs[x2]]
lb x1,40(x3)	Load byte	Regs[x1] ←$_{64}$ (Mem[40+Regs[x3]]$_0$)^{56} ## Mem[40+Regs[x3]]
lbu x1,40(x3)	Load byte unsigned	Regs[x1] ←$_{64}$ 0^{56} ## Mem[40+Regs[x3]]
lh x1,40(x3)	Load half word	Regs[x1] ←$_{64}$ (Mem[40+Regs[x3]]$_0$)^{48} ## Mem[40+Regs[x3]]
flw f0,50(x3)	Load FP single	Regs[f0] ←$_{64}$ Mem[50+Regs[x3]] ## 0^{32}
fld f0,50(x2)	Load FP double	Regs[f0] ←$_{64}$ Mem[50+Regs[x2]]
sd x2,400(x3)	Store double	Mem[400+Regs[x3]] ←$_{64}$ Regs[x2]
sw x3,500(x4)	Store word	Mem[500+Regs[x4]] ←$_{32}$ Regs[x3]$_{32..63}$
fsw f0,40(x3)	Store FP single	Mem[40+Regs[x3]] ←$_{32}$ Regs[f0]$_{0..31}$
fsd f0,40(x3)	Store FP double	Mem[40+Regs[x3]] ←$_{64}$ Regs[f0]
sh x3,502(x2)	Store half	Mem[502+Regs[x2]] ←$_{16}$ Regs[x3]$_{48..63}$
sb x2,41(x3)	Store byte	Mem[41+Regs[x3]] ←$_{8}$ Regs[x2]$_{56..63}$

Figure A.25 The load and store instructions in RISC-V. Loads shorter than 64 bits are available in both sign-extended and zero-extended forms. All memory references use a single addressing mode. Of course, both loads and stores are available for all the data types shown. Because RV64G supports double precision floating point, all single precision floating point loads must be aligned in the FP register, which are 64-bits wide.

RISC-V

As an example, assuming that x8 and x10 are 32-bit registers:

Regs[x10] ←$_{64}$ (Mem[Regs[x8]]$_0$)^{32} ## Mem[Regs[R8]]

means that the word at the memory location addressed by the contents of register x8 is sign-extended to form a 64-bit quantity that is stored into register x10.

ALU instructions are register-register instructions. Figure A.26 gives some examples of the arithmetic/logical instructions

Example instrucmtion	Instruction name	Meaning
add x1,x2,x3	Add	Regs[x1] ← Regs[x2]+Regs[x3]
addi x1,x2,3	Add immediate unsigned	Regs[x1] ← Regs[x2]+3
lui x1,42	Load upper immediate	Regs[x1] ← 0^{32}##42##0^{12}
sll x1,x2,5	Shift left logical	Regs[x1] ← Regs[x2]<<5
slt x1,x2,x3	Set less than	if (Regs[x2]<Regs[x3]) Regs[x1] ← 1 else Regs[x1] ← 0

Figure A.26 The basic ALU instructions in RISC-V are available both with register-register operands and with one immediate operand. LUI uses the U-format that employs the rs1 field as part of the immediate, yielding a 20-bit immediate.

RISC-V

Operations include simple arithmetic and logical operations: add, subtract, AND, OR, XOR, and shifts. Immediate forms of all these instructions are provided using a 12-bit sign-extended immediate.

Operation LUI (load upper immediate) loads bits 12–31 of a register, sign extends the immediate field to the upper 32-bits and sets low-order 12-bits of the register to 0.

LUI allows a 32-bit constant to be built in two instructions, or a data transfer using any constant 32-bit address in one extra instruction.

x0 is used to analyze popular operations. Loading a constant is simply an add immediate where the source operand is x0, and a register-register move is simply an add (or an or) where one of the sources is x0.

RISC-V

RISC-V Control Flow Instructions: Control is handled through a set of jumps and a set of branches as shown in Figure A.27.

Two jump instructions (jump and link and jump and link register) are unconditional transfers and store "link," which is address of instruction sequentially following jump in register specified by the rd field.

When link address is not needed, rd field can simply be set to x0, which results in a typical unconditional jump.

Two jump instructions are differentiated by: whether the address is computed by adding an immediate field to the PC or by adding the immediate field to the contents of a register. Offset is interpreted as a half word offset for compatibility with compressed instruction set R64C, which includes 16-bit instructions.

Example instruction	Instruction name	Meaning
`jal x1,offset`	Jump and link	`Regs[x1]←PC+4; PC←PC+(offset<<1)`
`jalr x1,x2,offset`	Jump and link register	`Regs[x1]←PC+4; PC←Regs[x2]+offset`
`beq x3,x4,offset`	Branch equal zero	`if (Regs[x3]==Regs[x4]) PC←PC+(offset<<1)`
`bgt x3,x4,name`	Branch not equal zero	`if (Regs[x3]>Regs[x4]) PC←PC+(offset<<1)`

Figure A.27 Typical control flow instructions in RISC-V. All control instructions, except jumps to an address in a register, are PC-relative.

RISC-V

All branches are conditional. Branch condition is specified by the instruction, and any arithmetic comparison (equal, greater than, less than, and their inverses) is permitted.

Branch-target address is specified with a 12-bit signed offset that is shifted left one place (to get 16-bit alignment) and then added to current program counter.

Branches based on contents of floating point registers are implemented by executing a floating point comparison (e.g., feq.d or fle.d), which sets an integer register to 0 or 1 based on the comparison and then executing a beq or bne with x0 as an operand.

Example instruction	Instruction name	Meaning
`jal x1,offset`	Jump and link	`Regs[x1]←PC+4; PC←PC+(offset<<1)`
`jalr x1,x2,offset`	Jump and link register	`Regs[x1]←PC+4; PC←Regs[x2]+offset`
`beq x3,x4,offset`	Branch equal zero	`if (Regs[x3]==Regs[x4]) PC←PC+(offset<<1)`
`bgt x3,x4,name`	Branch not equal zero	`if (Regs[x3]>Regs[x4]) PC←PC+(offset<<1)`

Figure A.27 Typical control flow instructions in RISC-V. All control instructions, except jumps to an address in a register, are PC-relative.

RISC-V

RISC V Floating Point Operations: Floating-point instructions manipulate floating-point registers and indicate whether the operation to be performed is single or double precision.

Floating-point operations are add, subtract, multiply, divide, square root, as well as fused multiply-add and multiply-subtract. All floating point instructions begin with the letter f and use the suffix d for double precision and s for single precision (e.g., fadd.d, fadd.s, fmul.d, fmul.s, fmadd.d fmadd.s).

Floating point compares set an integer register based on the comparison, similarly to the integer instruction set-less-than and set-great-than.

In addition to floating-point loads and stores (flw, fsw, fld, fsd), instructions are provided for converting between different FP precisions, for moving between integer and FP registers (fmv), and for converting between floating point and integer (fcvt, which uses the integer registers for source or destination as appropriate).

Instruction type/opcode	Instruction meaning
Data transfers	*Move data between registers and memory, or between the integer and FP; only memory address mode is 12-bit displacement + contents of a GPR*
lb, lbu, sb	Load byte, load byte unsigned, store byte (to/from integer registers)
lh, lhu, sh	Load half word, load half word unsigned, store half word (to/from integer registers)
lw, lwu, sw	Load word, store word (to/from integer registers)
ld, sd	Load doubleword, store doubleword
Arithmetic/logical	*Operations on data in GPRs. Word versions ignore upper 32 bits*
add, addi, addw, addiw, sub, subi, subw, subiw	Add and subtract, with both word and immediate versions
slt, sltu, slti, sltiu	set-less-than with signed and unsigned, and immediate
and, or, xor, andi, ori, xori	and, or, xor, both register-register and register-immediate
lui	Load upper immediate: loads bits 31..12 of a register with the immediate value. Upper 32 bits are set to 0
auipc	Sums an immediate and the upper 20-bits of the PC into a register; used for building a branch to any 32-bit address
sll, srl, sra, slli, srli, srai, sllw, slliw, srliw, sraiw	Shifts: logical shift left and right and arithmetic shift right, both immediate and word versions (word versions leave the upper 32 bit untouched)
mul, mulw, mulh, mulhsu, mulhu, div, divw, divu, rem, remu, remw, remuw	Integer multiply, divide, and remainder, signed and unsigned with support for 64-bit products in two instructions. Also word versions
Control	*Conditional branches and jumps; PC-relative or through register*
beq, bne, blt, bge, bltu, bgeu	Branch based on compare of two registers, equal, not equal, less than, greater or equal, signed and unsigned
jal, jalr	Jump and link address relative to a register or the PC
Floating point	*All FP operation appear in double precision (.d) and single (.s)*
flw, fld, fsw, fsd	Load, store, word (single precision), doubleword (double precision)
fadd, fsub, fmult, fiv, fsqrt, fmadd, fmsub, fnmadd, fnmsub, fmin, fmax, fsgn, fsgnj, fsjnx	Add, subtract, multiply, divide, square root, multiply-add, multiply-subtract, negate multiply-add, negate multiply-subtract, maximum, minimum, and instructions to replace the sign bit. For single precision, the opcode is followed by: .s, for double precision: .d. Thus fadd.s, fadd.d
feq, flt, fle	Compare two floating point registers; result is 0 or 1 stored into a GPR
fmv.x.*, fmv.*.x	Move between the FP register abd GPR, "*" is s or d
fcvt.*.l, fcvt.l.*, fcvt.*.lu, fcvt.lu.*, fcvt.*.w, fcvt.w.*, fcvt.*.wu, fcvt.wu.*	Converts between a FP register and integer register, where "*" is S or D for single or double precision. Signed and unsigned versions and word, doubleword versions

Figure A.28 A list of the vast majority of instructions in RV64G. This list can also be found on the back inside cover. This table omits system instructions, synchronization and atomic instructions, configuration instructions, instructions to reset and access performance counters, about 10 instructions in total.

RISC-V

Shows IMUL instruction, which multiplies ECX by 32 to compute the offset for Q[I].VAL, since multiplication by 32 is equivalent to a left shift by 5 bits (because 32 = 2 ^5).

15.5 Consider the following code fragment in a high-level language:

```
for I in 1...100 loop
    S ← S + Q(I).VAL
end loop;
```

Assume that Q is an array of 32-byte records and the VAL field is in the first 4 bytes of each record. Using x86 code, we can compile this program fragment as follows:

```
        MOV     ECX,1           ;use register ECX to hold I
LP:     IMUL    EAX, ECX, 32    ;get offset in EAX
        MOV     EBX, Q[EAX]     ;load VAL field
        ADD     S, EBX          ;add to S
        INC     ECX             ;increment I
        CMP     ECX, 101        :compare to 101
        JNE     LP              ;loop until I = 100
```

This program makes use of the IMUL instruction, which multiplies the second operand by the immediate value in the third operand and places the result in the first operand (see Problem 10.13). A RISC advocate would like to demonstrate that a clever compiler can eliminate unnecessarily complex instructions such as IMUL. Provide the demonstration by rewriting the above x86 program without using the IMUL instruction.

RISC-V

To eliminate the IMUL instruction, we can replace it with a SHL (shift left) instruction, which is simpler and performs the same operation as multiplying by 32.

```
MOV ECX, 1       ; use register ECX to hold I
LP:
SHL ECX, 5       ; shift ECX left by 5 bits to multiply ECX by 32
MOV EBX, Q[ECX]  ; load VAL field from Q[ECX] (equivalent to Q[ECX*32])
ADD S, EBX       ; add to S
DEC ECX          ; decrement ECX by 1 to adjust for the shift
CMP ECX, 100     ; compare ECX with 100
JNE LP           ; loop until ECX = 100
```

MOV ECX, 1 initializes the loop variable I to 1

SHL ECX, 5 performs a left shift on ECX by 5 bits, multiplying ECX by 32 (equivalent to the previous IMUL instruction).

MOV EBX, Q[ECX] loads the value of Q[ECX].VAL into EBX.
Shift operation ensures that Q[ECX] corresponds to the correct record.

ADD S, EBX adds the value of Q[ECX].VAL (now in EBX) to S.

DEC ECX decrements the loop counter by 1, so the loop properly ends when ECX reaches 101.

CMP ECX, 100 checks if the loop counter has reached 100

JNE LP causes the loop to repeat if it has not.

Pipeline : Basic and Intermediate Concepts

What is Pipelining

Pipelining: is the process of accumulating instruction from the processor through a pipeline. It allows storing and executing instructions in an orderly process. It is also known as pipeline processing. Pipelining is a technique where multiple instructions are overlapped during execution.

Throughput: of an instruction pipeline is determined by how often an instruction exits the pipeline. Pipe stages are hooked together, all the stages must be ready to proceed at the same time.

Processor Cycle: The time required between moving an instruction one step down the pipeline is a processor cycle. All stages proceed at the same time, the length of a processor cycle is determined by the time required for the slowest pipe stage. In a computer, this processor cycle is almost always 1 clock cycle.

If the stages are perfectly balanced, then the time per instruction on the pipelined processor is equal to:

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

Basics of RISC V Instruction Set

All RISC architectures are characterized by a few key properties:

■ All operations on data apply to data in registers and typically change the entire register (32 or 64 bits per register).

■ The only operations that affect memory are load and store operations that move data from memory to a register or to memory from a register, respectively.

Load and store operations that load or store less than a full register (e.g., a byte, 16 bits, or 32 bits) are often available.

■ The instruction formats are few in number, with all instructions typically being one size. In RISC V, the register specifiers: rs1, rs2, and rd are always in the same place simplifying the control.

Simple Implementation of RISC Instruction Set

Every instruction in RISC subset can be implemented in 5 clock cycles. The 5 clock cycles are as follows.

1. Instruction fetch cycle (IF): Send program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential instruction by adding 4 (because each instruction is 4 bytes) to the PC.

2. Instruction decode/register fetch cycle (ID): Decode the instruction and read the registers corresponding to register source specifiers from register file. Sign-extend the offset field of the instruction. Compute the possible branch target address by adding the sign-extended offset to the incremented PC.

Fixed Field Decoding: is done in parallel with reading registers, which is possible because register specifiers are at a fixed location in a RISC architecture. This technique is known as fixed-field decoding.

For loads and ALU immediate operations, immediate field is always in same place, so we can easily sign extend it. For implementation of RISC V, we need to compute two different sign-extended values, because immediate field for store is in a different location.

Simple Implementation of RISC Instruction Set

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

- **Memory reference:** ALU adds base register and offset to form the effective address.
- **Register-Register ALU instruction:** ALU performs the operation specified by the ALU opcode on the values read from the register file.
- **Register-Immediate ALU instruction:** ALU performs the operation specified by the ALU opcode on the first value read from the register file and the sign-extended immediate.
- **Conditional branch:** Determine whether the condition is true. In a load-store architecture, effective address and execution cycles can be combined into a single clock cycle, because no instruction needs to simultaneously calculate a data address and perform an operation on data.

Simple Implementation of RISC Instruction Set

4. Memory access (MEM): If the instruction is a load, the memory does a read using the effective address computed in the previous cycle. If it is a store, then the memory writes the data from the second register read from the register file using the effective address.

5. Write-back cycle (WB):

- **Register-Register ALU instruction or load instruction:** Write the result into the register file, whether it comes from the memory system (for a load) or from the ALU (for an ALU instruction).

In this implementation, branch instructions require three cycles, store instructions require four cycles, and all other instructions require five cycles.

Classic Five-Stage Pipeline for a RISC Processor

	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i*+1		IF	ID	EX	MEM	WB			
Instruction *i*+2			IF	ID	EX	MEM	WB		
Instruction *i*+3				IF	ID	EX	MEM	WB	
Instruction *i*+4					IF	ID	EX	MEM	WB

Figure C.1 Simple RISC pipeline. On each clock cycle, another instruction is fetched and begins its five-cycle execution. If an instruction is started every clock cycle, the performance will be up to five times that of a processor that is not pipelined. The names for the stages in the pipeline are the same as those used for the cycles in the unpipelined implementation: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, and WB = write-back.

Execution pattern shown in Figure C.1 a pipeline structure. Each instruction takes 5 clock cycles to complete, during each clock cycle the hardware will initiate a new instruction and will be executing some part of five different instructions.

Classic Five-Stage Pipeline for a RISC Processor

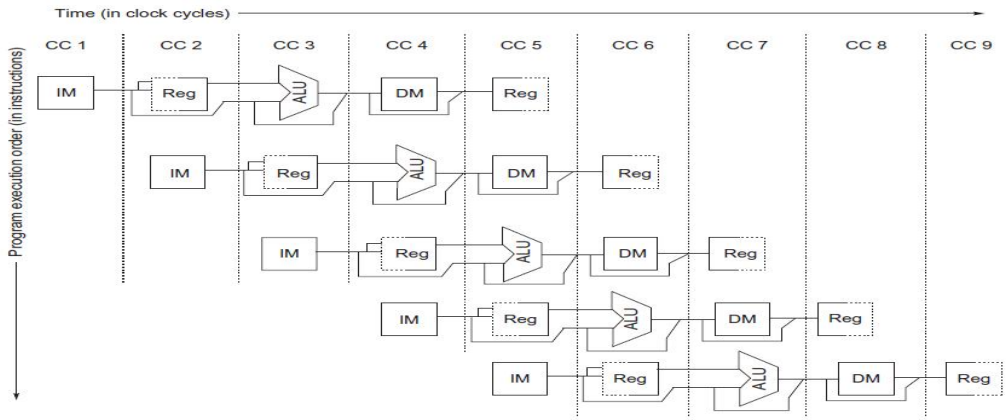

Figure C.2 The pipeline can be thought of as a series of data paths shifted in time. This figure shows the overlap among the parts of the data path, with clock cycle 5 (CC 5) showing the steady-state situation. Because the register file is used as a source in the ID stage and as a destination in the WB stage, it appears twice. We show that it is read in one part of the stage and written in another by using a solid line, on the right or left, respectively, and a dashed line on the other side. The abbreviation IM is used for instruction memory, DM for data memory, and CC for clock cycle.

Figure C.2 shows a simplified version of a RISC data path drawn in pipeline fashion. Major functional units are used in different cycles and overlapping the execution of multiple instructions introduces relatively few conflicts.

Classic Five-Stage Pipeline for a RISC Processor

There are three observations on which this fact rests.

1. Use separate instruction and data memories, which we implement with separate instruction and data caches. Use of separate caches eliminates a conflict for a single memory that arise between instruction fetch and data memory access. If our pipelined processor has a clock cycle that is equal to that of the unpipelined version, the memory system must deliver five times the bandwidth. This increased demand is one cost of higher performance.

2. Register file is used in the two stages: one for reading in ID and one for writing in WB. To perform two reads and one write every clock cycle. To handle reads and a write to same register, we perform register write in first half of clock cycle and read in second half.

3.Figure C.2 does not deal with the PC. To start a new instruction every clock, we must increment and store the PC every clock, and this must be done during the IF stage in preparation for the next instruction. We must have an adder to compute the potential branch target address during ID. Need ALU in the ALU stage to evaluate the branch condition.

Classic Five-Stage Pipeline for a RISC Processor

Figure C.3 shows the pipeline drawn with these pipeline registers.

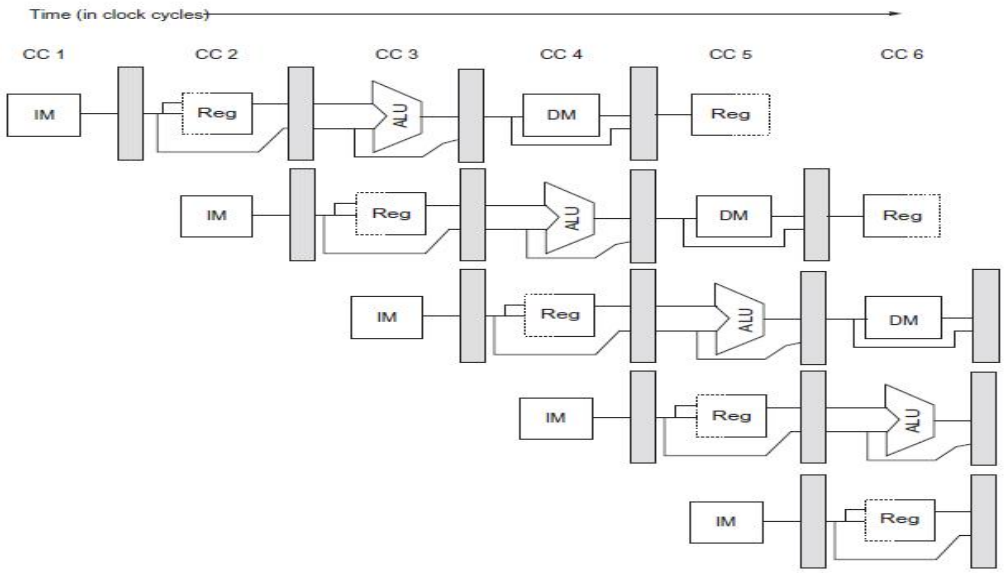

Figure C.3 A pipeline showing the pipeline registers between successive pipeline stages. Notice that the registers prevent interference between two different instructions in adjacent stages in the pipeline. The registers also play the critical role of carrying data for a given instruction from one stage to the other. The edge-triggered property of registers—that is, that the values change instantaneously on a clock edge—is critical. Otherwise, the data from one instruction could interfere with the execution of another!

Classic Five-Stage Pipeline for a RISC Processor

Similar registers would be needed even in a multicycle data path that had no pipelining.

In case of a pipelined processor, pipeline registers play the key role of carrying intermediate results from one stage to another where the source and destination may not be directly adjacent.

For example, register value to be stored during a store instruction is read during ID, but not actually used until MEM; it is passed through two pipeline registers to reach the data memory during the MEM stage.

Likewise, the result of an ALU instruction is computed during EX, but not actually stored until WB; it arrives there by passing through two pipeline registers.

It is useful to name the pipeline registers, and we follow the convention of naming them by the pipeline stages they connect, so the registers are called IF/ID, ID/EX, EX/MEM, and MEM/WB.

Basic Performance Issues in Pipelining

- ✓ Execution time of each instruction does not decrease put limits on the practical depth of a pipeline.
- ✓ In addition to limitations arising from pipeline latency, limits arise from imbalance among the pipe stages and from pipelining overhead.
- ✓ Imbalance among pipe stages reduce performance because clock can run no faster than the time needed for slowest pipeline stage.
- ✓ Pipeline overhead arises from combination of pipeline register delay and clock skew.
- ✓ Pipeline registers add setup time, which is the time that a register input must be stable before the clock signal that triggers a write occurs, plus propagation delay to the clock cycle.
- ✓ **Clock skew:** is the maximum delay between when the clock arrives at any two registers, also contributes to the lower limit on the clock cycle.
- ✓ Once the clock cycle is as small as the sum of the clock skew and latch overhead, no further pipelining is useful because there is no time left in cycle for useful work.

Basic Performance Issues in Pipelining

Example Consider the unpipelined processor in the previous section. Assume that it has a 4 GHz clock (or a 0.5 ns clock cycle) and that it uses four cycles for ALU operations and branches and five cycles for memory operations. Assume that the relative frequencies of these operations are 40%, 20%, and 40%, respectively. Suppose that due to clock skew and setup, pipelining the processor adds 0.1 ns of overhead to the clock. Ignoring any latency impact, how much speedup in the instruction execution rate will we gain from a pipeline?

Major Hurdle of Pipelining - Pipelining Hazards

Hazards: There are situations called hazards, that prevent the next instruction in the instruction stream from executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining. There are three classes of hazards:

1.Structural hazards: arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.

In modern processors, structural hazards occur primarily in special purpose functional units that are less frequently used (such as floating point divide or other complex long running instructions).

2. Data hazards: arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.

3. Control hazards: arise from the pipelining of branches and other instructions that change the PC.

Performance of Pipelines With Stalls

A stall causes the pipeline performance to degrade from the ideal performance. Let's look at a simple equation for finding the actual speedup from pipelining, starting with the formula from the previous section:

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$

$$= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}$$

$$= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}$$

Pipelining can be thought of as decreasing the CPI or the clock cycle time. Because it is traditional to use the CPI to compare pipelines, let's start with that assumption. The ideal CPI on a pipelined processor is almost always 1. Hence, we can compute the pipelined CPI:

$$\text{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}$$

$$= 1 + \text{Pipelines stall clock cycles per instruction}$$

Performance of Pipelines With Stalls

If we ignore the cycle time overhead of pipelining and assume that the stages are perfectly balanced, then the cycle time of the two processors can be equal, leading to

$$\text{Speedup} = \frac{\text{CPI unpiplined}}{1 + \text{Pipeline stall cycles per instruction}}$$

One important simple case is where all instructions take the same number of cycles, which must also equal the number of pipeline stages (also called the *depth of the pipeline*). In this case, the unpipelined CPI is equal to the depth of the pipeline, leading to

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

If there are no pipeline stalls, this leads to the intuitive result that pipelining can improve performance by the depth of the pipeline.

Performance of Pipelines With Stalls

C.3 [5/15/10/10]<C.2>We begin with a computer implemented in single-cycle implementation. When the stages are split by functionality, the stages do not require exactly the same amount of time. The original machine had a clock cycle time of 7 ns. After the stages were split, the measured times were IF, 1 ns; ID, 1.5 ns; EX, 1 ns; MEM, 2 ns; and WB, 1.5 ns. The pipeline register delay is 0.1 ns.

a. [5]<C.2>What is the clock cycle time of the 5-stage pipelined machine?

b. [15]<C.2>If there is a stall every four instructions, what is the CPI of the new machine?

c. [10]<C.2>What is the speedup of the pipelined machine over the single-cycle machine?

d. [10]<C.2>If the pipelined machine had an infinite number of stages, what would its speedup be over the single-cycle machine?

Data Hazards

Data Hazards: occur when an instruction depends on the result of previous instruction and that result of instruction has not yet been computed. whenever two different instructions use the same storage. the location must appear as if it is executed in sequential order.

1. Read After Write (RAW) hazard: RAW also known as True dependency or Flow dependency. It occurs when the value produced by an instruction is required by a subsequent instruction. For example,

```
ADD R1, --, --;
SUB --, R1, --;
```

2. Write After Read (WAR) hazard: WAR also known as anti dependency. These hazards occur when the output register of an instruction is used right after read by a previous instruction. For example,

```
ADD --, R1, --;
SUB R1, --, --;
```

3. Write After Write (WAW) hazard: WAW also known as output dependency. These hazards occur when the output register of an instruction is used for write after written by previous instruction. For example,

```
ADD R1, --, --;
SUB R1, --, --;
```

Data Hazards

Consider the pipelined execution of these instructions:

```
add   x1,x2,x3
sub   x4,x1,x5
and   x6,x1,x7
or    x8,x1,x9
xor   x10,x1,x11
```

All the instructions after the add use the result of the add instruction.

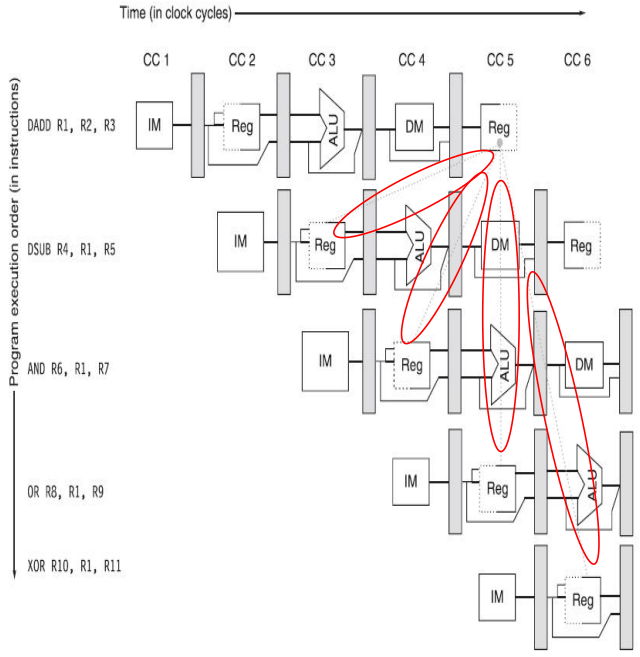

Figure C.4 The use of the result of the add instruction in the next three instructions causes a hazard, because the register is not written until after those instructions read it.

Data Hazards

IF ID EX MEM WB

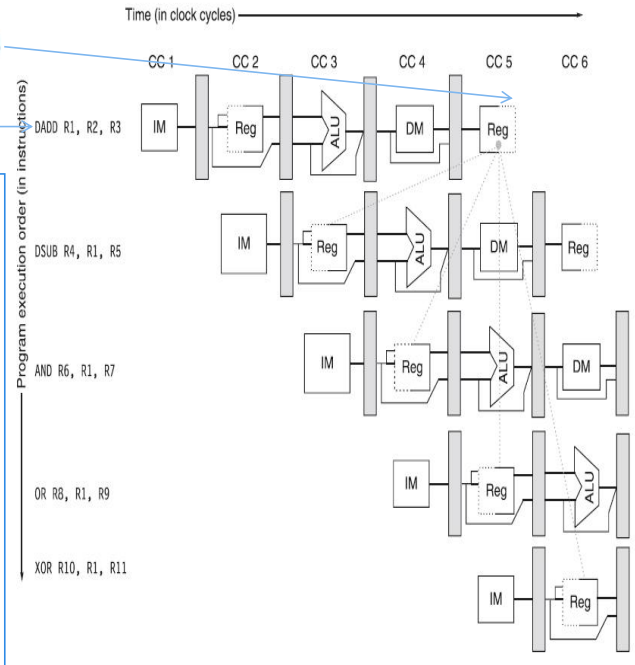

Figure C.4 The use of the result of the add instruction in the next three instructions causes a hazard, because the register is not written until after those instructions read it.

-Add instruction writes the value of x1 in the WB pipe stage, but the sub instruction reads the value during its ID stage, which results in aRAW hazard.
- sub instruction will read the wrong value and try to use it.

-If an interrupt occur between add and sub instructions, WB stage of add will complete, and the value of x1 at that point will be the result of the add.

Data Hazards

IF ID EX MEM WB

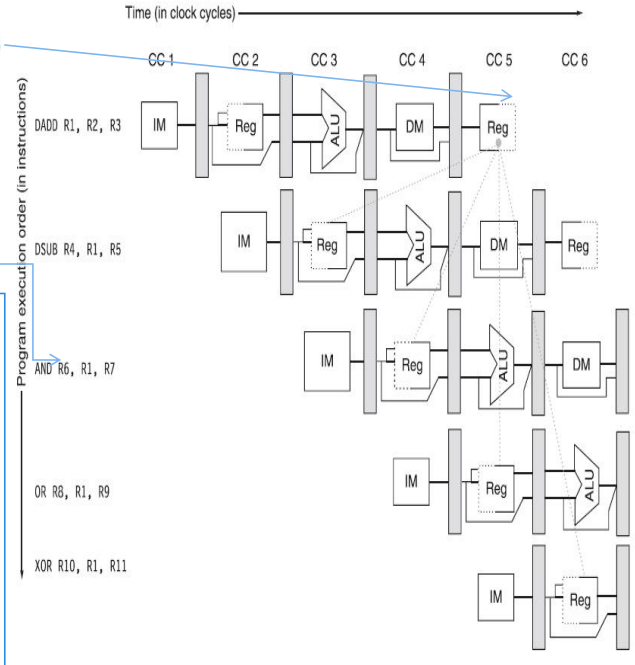

Figure C.4 The use of the result of the add instruction in the next three instructions causes a hazard, because the register is not written until after those instructions read it.

- AND instruction also creates a possible RAW hazard.

-write of x1 does not complete until the end of clock cycle 5.

- AND instruction that reads the registers during clock cycle 4 will receive the wrong results.

Data Hazards

IF ID EX MEM WB

-XOR instruction operates properly because its register read occurs in clock cycle 6, after the register write.

-OR instruction also operates without incurring a hazard because we perform the register file reads in the second half of the cycle and the writes in the first half.

-XOR instruction still depends on the add, but it no longer creates a hazard;

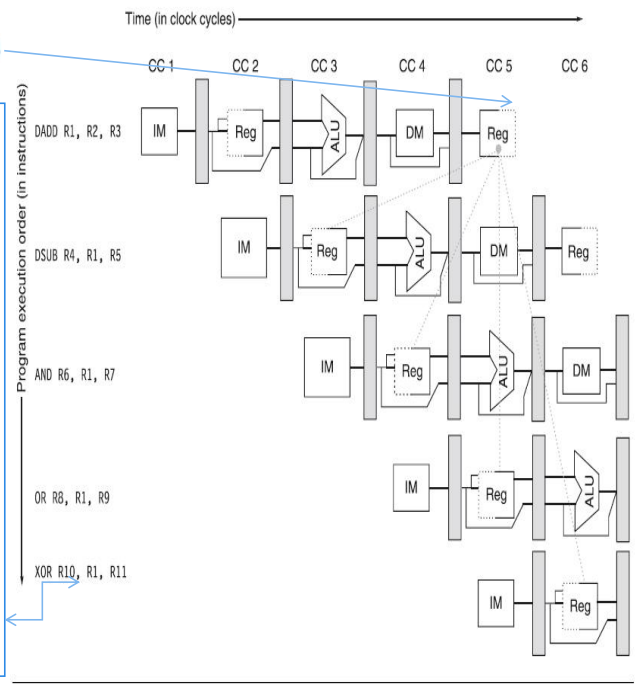

Figure C.4 The use of the result of the add instruction in the next three instructions causes a hazard, because the register is not written until after those instructions read it.

Minimizing Data Hazard Stalls by Forwarding

Problem posed in Figure C.4 can be solved with a simple hardware technique called forwarding (also called bypassing and sometimes short-circuiting).

Forwarding: result is not really needed by sub until after the add actually produces it. If the result can be moved from the pipeline register where the add stores it to where the sub needs it, then the need for a stall can be avoided. Using this observation, forwarding works as follows:

1. ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs.

2. If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.

Notice that with forwarding, if the sub is stalled, the add will be completed and the bypass will not be activated. This relationship is also true for the case of an interrupt between the two instructions.

Minimizing Data Hazard Stalls by Forwarding

Figure C.5 shows our example with bypass paths in place and highlighting the timing of the register read and writes. This code sequence can be executed without stalls.

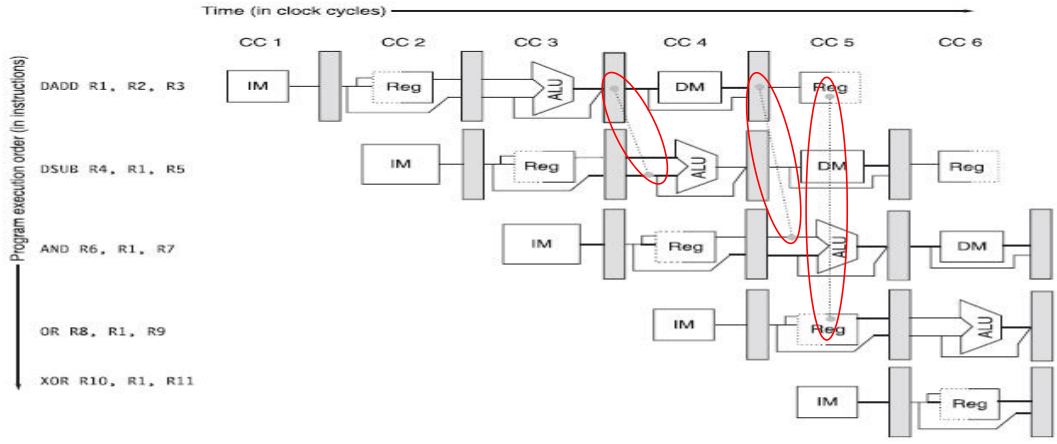

Figure C.5 A set of instructions that depends on the `add` **result uses forwarding paths to avoid the data hazard.** The inputs for the `sub` and `and` instructions forward from the pipeline registers to the first ALU input. The `or` receives its result by forwarding through the register file, which is easily accomplished by reading the registers in the second half of the cycle and writing in the first half, as the dashed lines on the registers indicate. Notice that the forwarded result can go to either ALU input; in fact, both ALU inputs could use forwarded inputs from either the same pipeline register or from different pipeline registers. This would occur, for example, if the `and` instruction was `and x6,x1,x4`.

Minimizing Data Hazard Stalls by Forwarding

For example, the following sequence:

add x1,x2,x3

ld x4,0(x1)

sd x4,12(x1)

To prevent a stall, we need to forward the values of ALU output and memory unit output from pipeline registers to ALU and data memory inputs.

Figure C.6 shows all the forwarding paths for this example.

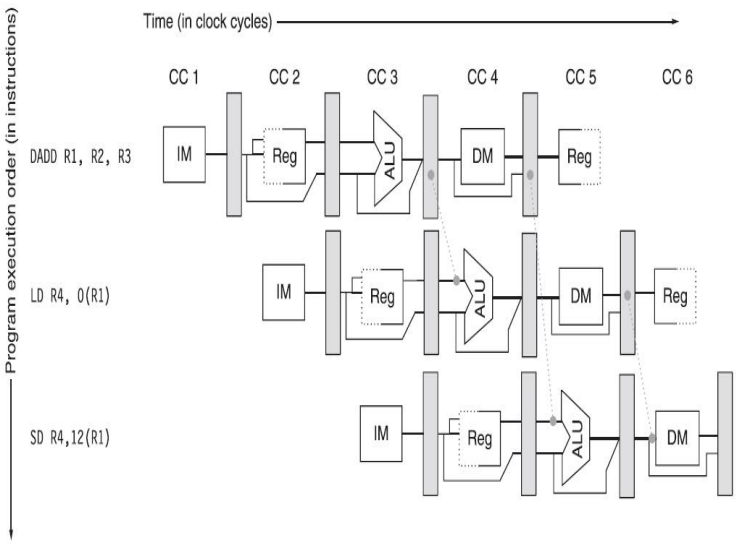

Figure C.6 Forwarding of operand required by stores during MEM. The result of the load is forwarded from the memory output to the memory input to be stored. In addition, the ALU output is forwarded to the ALU input for the address calculation of both the load and the store (this is no different than forwarding to another ALU operation). If the store depended on an immediately preceding ALU operation (not shown herein), the result would need to be forwarded to prevent a stall.

Data Hazard Requiring Stalls

Not all potential data hazards can be handled by bypassing. Consider the following sequence of instructions:

LD x1,0(x2)

SUB x4,x1,x5

AND x6,x1,x7

OR x8,x1,x9

Pipelined data path with the bypass paths is shown in Fig C.7.

LD instruction does not have the data until the end of clock cycle 4 (MEM cycle), while SUB instruction needs to have data by beginning of that clock cycle. Data hazard from using the result of a load instruction cannot be eliminated with hardware.

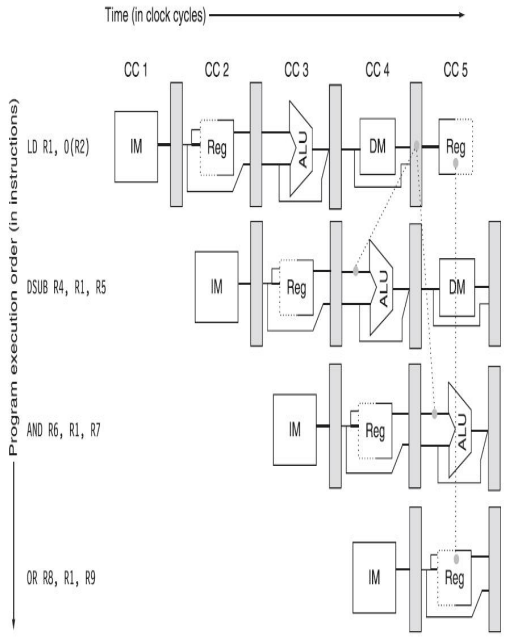

Figure C.7 The load instruction can bypass its results to the and and or instructions, but not to the sub, because that would mean forwarding the result in "negative time."

Data Hazard Requiring Stalls

Figure C.7 shows a forwarding path would have to operate backward in time—a capability not yet available. We can forward result immediately to ALU from the pipeline registers for use in the AND operation, which begins 2 clock cycles after the LOAD.

OR instruction has no problem because it receives the value through the register file.

For the sub instruction, the forwarded result arrives too late—at the end of a clock cycle, when it is needed at the beginning.

LOAD instruction has a delay or latency that cannot be eliminated by forwarding alone.

Pipeline Interlock Hardware: we need to add hardware called a pipeline interlock, to preserve the correct execution pattern. Pipeline interlock detects a hazard and stalls the pipeline until the hazard is cleared.

The interlock stalls the pipeline, beginning with the instruction that wants to use the data until the source instruction produces it. This pipeline interlock introduces a stall or bubble. CPI for the stalled instruction increases by the length of the stall (1 clock cycle).

Data Hazard Requiring Stalls

Figure C.8 shows pipeline before and after the stall using the names of the pipeline stages. Stall causes the instructions starting with the SUB to move one cycle later in time, the forwarding to the AND instruction now goes through the register file, and no forwarding at all is needed for OR instruction.

The insertion of the bubble causes the number of cycles to complete this sequence to grow by one. No instruction is started during clock cycle 4 (and none finishes during cycle 6)

ld x1,0(x2)	IF	ID	EX	MEM	WB				
sub x4,x1,x5		IF	ID	EX	MEM	WB			
and x6,x1,x7			IF	ID	EX	MEM	WB		
or x8,x1,x9				IF	ID	EX	MEM	WB	
ld x1,0(x2)	IF	ID	EX	MEM	WB				
sub x4,x1,x5		IF	ID	Stall	EX	MEM	WB		
and x6,x1,x7			IF	Stall	ID	EX	MEM	WB	
or x8,x1,x9				Stall	IF	ID	EX	MEM	WB

Figure C.8 In the top half, we can see why a stall is needed: the MEM cycle of the load produces a value that is needed in the EX cycle of the sub, which occurs at the same time. This problem is solved by inserting a stall, as shown in the bottom half.

Branch Hazards

Control hazards or Branch Hazards: Control hazard occurs when the pipeline makes wrong decisions on branch prediction and therefore brings instructions into the pipeline that must subsequently be discarded.

Control Hazards can cause a greater performance loss for our RISC V pipeline than data hazards. When a branch is executed, it may or may not change the PC to its current value plus 4.

If a branch changes the PC to its target address, it is a taken branch; if it falls through, it is not taken, or untaken.

If instruction i is a taken branch, then the PC is usually not changed until the end of ID, after the completion of the address calculation and comparison.

Figure C.9 shows that the simplest method of dealing with branches is to redo the fetch of the instruction following a branch, once we detect the branch during ID (when instructions are decoded).

Branch Hazards

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor+1				IF	ID	EX	MEM
Branch successor+2					IF	ID	EX

Figure C.9 A branch causes a one-cycle stall in the five-stage pipeline. The instruction after the branch is fetched, but the instruction is ignored, and the fetch is restarted once the branch target is known. It is probably obvious that if the branch is not taken, the second IF for branch successor is redundant. This will be addressed shortly.

The first IF cycle is essentially a stall because it never performs useful work.

If the branch is untaken then the repetition of the IF stage is unnecessary because the correct instruction was indeed fetched.

One stall cycle for every branch will yield a performance loss of 10% to 30% depending on the branch frequency, so we will examine some techniques to deal with this loss.

Reducing Pipeline Branch Penalties

There are many methods for dealing with the pipeline stalls caused by branch delay.

1. The simplest scheme to handle branches is to freeze or flush the pipeline, holding or deleting any instructions after the branch until the branch destination is known.

The attractiveness of this solution lies primarily in its simplicity both for hardware and software. It is the solution used earlier in the pipeline shown in Figure C.9. In this case, the branch penalty is fixed and cannot be reduced by software.

2. A higher-performance and only slightly more complex, scheme is to treat every branch as not taken, simply allowing the hardware to continue as if the branch were not executed.

Care must be taken not to change the processor state until the branch outcome is known. The complexity of this scheme arises when the state might be changed by an instruction and how to “back out” such a change.

Reducing Pipeline Branch Penalties

3. In the simple five-stage pipeline, this predicted-not-taken or predicted-untaken scheme is implemented by continuing to fetch instructions as if the branch were a normal instruction.

If the branch is taken, we need to turn the fetched instruction into a no-op and restart the fetch at the target address. Figure C.10 shows both situations.

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Figure C.10 The predicted-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom). When the branch is untaken, determined during ID, we fetch the fall-through and just continue. If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following the branch to stall 1 clock cycle.

Reducing Pipeline Branch Penalties

An alternative scheme is to treat every branch as taken. As soon as the branch is decoded and the target address is computed, assume the branch to be taken and begin fetching and executing at the target.

This buys us a one-cycle improvement when the branch is actually taken, because we know the target address at the end of ID, one cycle before we know whether the branch condition is satisfied in the ALU stage.

In either a predicted-taken or predicted-not-taken scheme, the compiler can improve performance by organizing the code so that the most frequent path matches the hardware's choice.

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Figure C.10 The predicted-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom). When the branch is untaken, determined during ID, we fetch the fall-through and just continue. If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following the branch to stall 1 clock cycle.

Reducing Pipeline Branch Penalties

4. Scheme used in early RISC processors is called delayed branch. In a delayed branch, the execution cycle with a branch delay of one is branch instruction ,sequential successor, branch target if taken.

The sequential successor is in the branch delay slot. This instruction is executed whether or not the branch is taken. The pipeline behavior of five-stage pipeline with a branch delay is shown in Figure C.11.

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction ($i+1$)		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction ($i+1$)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Figure C.11 The behavior of a delayed branch is the same whether or not the branch is taken. The instructions in the delay slot (there was only one delay slot for most RISC architectures that incorporated them) are executed. If the branch is untaken, execution continues with the instruction after the branch delay instruction; if the branch is taken, execution continues at the branch target. When the instruction in the branch delay slot is also a branch, the meaning is unclear: if the branch is not taken, what should happen to the branch in the branch delay slot? Because of this confusion, architectures with delay branches often disallow putting a branch in the delay slot.

Reducing Pipeline Branch Penalties

Although it is possible to have a branch delay longer than one, in practice almost all processors with delayed branch have a single instruction delay.

Other techniques are used if the pipeline has a longer potential branch penalty.The job of the compiler is to make the successor instructions valid and useful.

Although the delayed branch was useful for short simple pipelines at a time when hardware prediction was too expensive, the technique complicates implementation when there is dynamic branch prediction. For this reason, RISC V appropriately omitted delayed branches.

Performance of Branch Schemes

Performance of Branch Schemes

What is the effective performance of each of these schemes? The effective pipeline speedup with branch penalties, assuming an ideal CPI of 1, is

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles from branches}}$$

Because of the following:

$$\text{Pipeline stall cycles from branches} = \text{Branch frequency} \times \text{Branch penalty}$$

we obtain:

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

The branch frequency and branch penalty can have a component from both unconditional and conditional branches. However, the latter dominate because they are more frequent.

Performance of Branch Schemes

Example For a deeper pipeline, such as that in a MIPS R4000 and later RISC processors, it takes at least three pipeline stages before the branch-target address is known and an additional cycle before the branch condition is evaluated, assuming no stalls on the registers in the conditional comparison. A three-stage delay leads to the branch penalties for the three simplest prediction schemes listed in Figure C.12.

Find the effective addition to the CPI arising from branches for this pipeline, assuming the following frequencies:

Unconditional branch	4%
Conditional branch, untaken	6%
Conditional branch, taken	10%

Answer We find the CPIs by multiplying the relative frequency of unconditional, conditional untaken, and conditional taken branches by the respective penalties. The results are shown in Figure C.13.

Branch scheme	Penalty unconditional	Penalty untaken	Penalty taken
Flush pipeline	2	3	3
Predicted taken	2	3	2
Predicted untaken	2	0	3

Figure C.12 Branch penalties for the three simplest prediction schemes for a deeper pipeline.