CSC 411

Computer Organization (Spring 2022)
Lecture 16: Basic CPU design, Pipelining

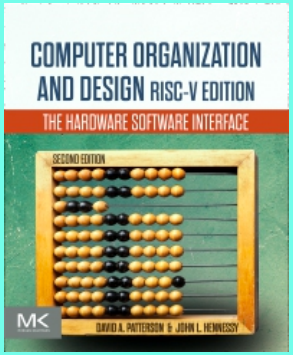
Prof. Marco Alvarez, University of Rhode Island

Disclaimer

Some of the following slides are adapted from:

Computer Organization and Design (Patterson and Hennessy)

The Hardware/Software Interface



Introduction

- CPU performance factors
  - instruction count (determined by ISA and compiler)
  - CPI and cycle time (determined by CPU hardware)
- We will examine two RISC-V implementations
  - a simplified version
  - a more realistic **pipelined** version
- Simple instruction subset, shows most aspects
  - memory reference: `ld, sd`
  - arithmetic/logical: `add, sub, and, or`
  - control transfer: `beq`

Instruction execution

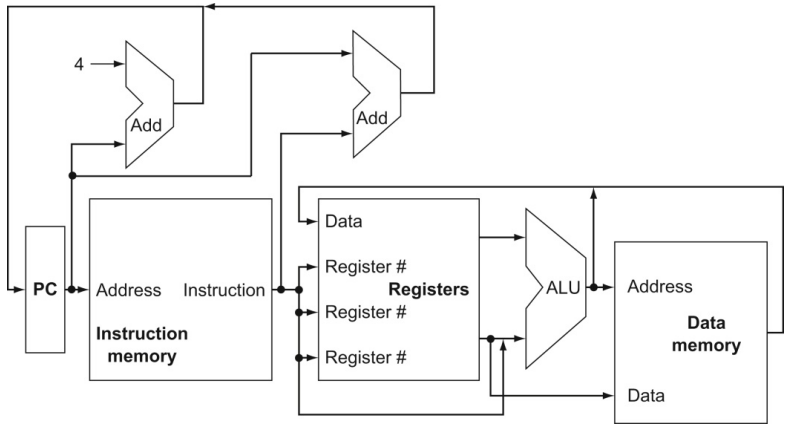
- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
  - use ALU to calculate
    - arithmetic result
    - memory address for load/store
    - branch comparison
  - access data memory for load/store
  - PC ← target address or PC + 4

Single-cycle CPU

Implementation overview

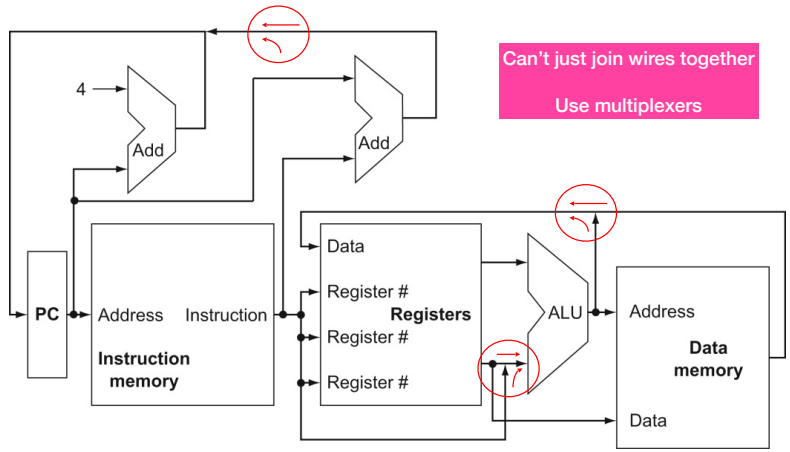
- CPU operations common to all instructions
  - use program counter (PC)
  - read register values
- Need memory
  - to store instructions
  - to store data
- Need registers, ALU, and control logic

CPU Overview



Major functional units and major connections between them. The thick lines interconnecting the functional units represent buses, which consist of multiple signals. The arrows are used to guide the reader in knowing how information flows. Since signal lines may cross, we explicitly show when crossing lines are connected by the presence of a dot where the lines cross.

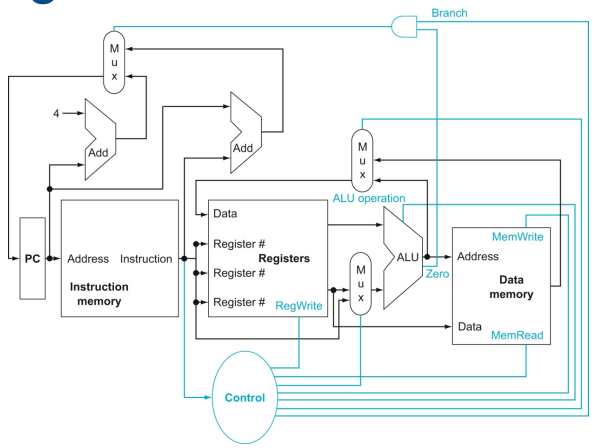
Multiplexers



Can't just join wires together

Use multiplexers

Adding control

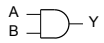


The basic implementation of the RISC-V subset, including the necessary multiplexors and control lines. The added control lines are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation.

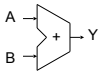
From logic design basics

- Information encoded in binary
  - low voltage = 0, high voltage = 1
  - one wire per bit
  - multi-bit data encoded on multi-wire buses
- **Combinational** element
  - operate on data
  - output is a function of input
- State (**sequential**) elements
  - store information

Combinational elements

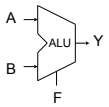


AND gate



Adder

Multiplexer

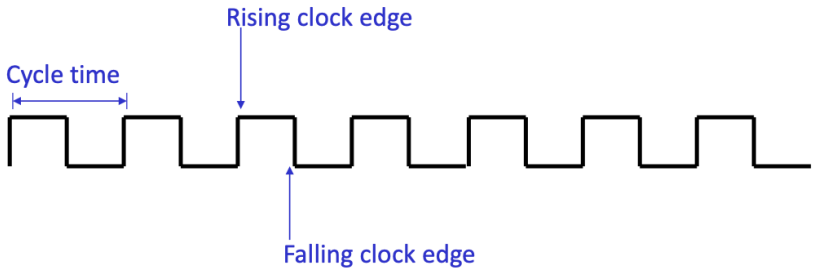


ALU

State elements

- Program counter
- Instruction memory
- Register file
- Data memory

Clock cycles

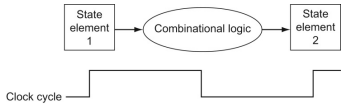


$$4\ \text{GHz} = \text{clock speed} = \frac{1}{\text{cycle time}} = \frac{1}{250\ \text{ps}}.$$

https://www.cs.utah.edu/~rajeev/cs3810/slides/3810-22-14.pdf

Clocking methodology

- Combinatorial logic transforms data during clock cycles
  - between clock edges
  - input from state elements, output to state element
  - longest delay determines clock period

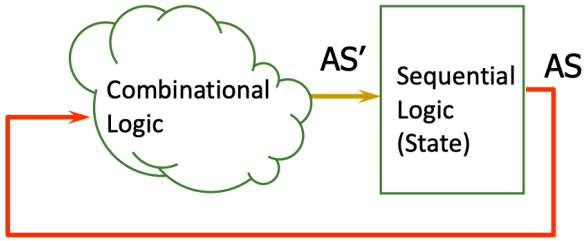


Any inputs to a state element must reach a stable value (that is, have reached a value from which they will not change until after the clock edge) before the active clock edge causes the state to be updated. All state elements in this chapter, including memory, are assumed positive edge-triggered; that is, they change on the rising clock edge.

An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could lead to indeterminate data values. Of course, the clock cycle still must be long enough so that the input values are stable when the active clock edge occurs.

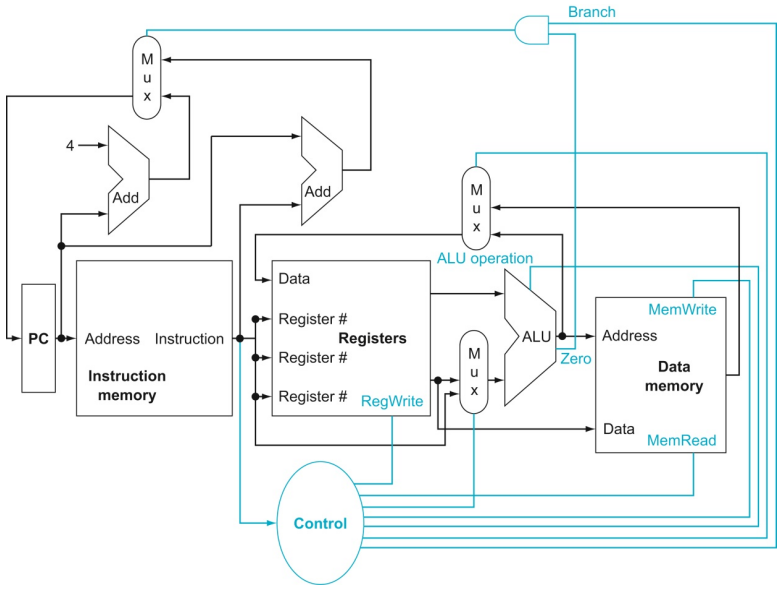
Basic instruction processing

- Each instruction takes a single clock cycle
  - only combinational logic is used to implement execution
- Clock cycle time determined by the **critical path** (longest delay path)

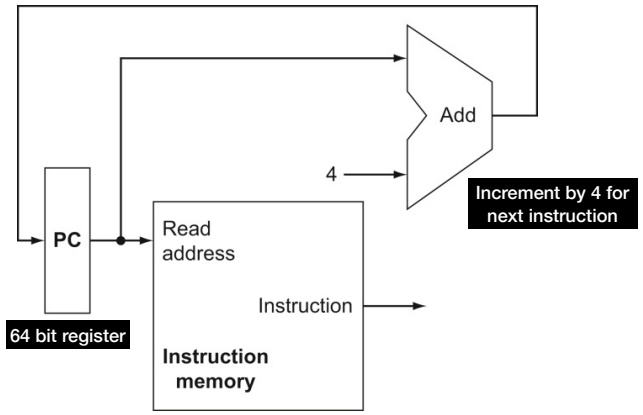


https://safari.ethz.ch/digitaltechnik/spring2022/lib/exe/fetch.php?media=onur-digitaldesign_comparch-2022-lecture11-microarchitecture-fundamentals-afterelecture.pdf

Looking into the datapath



Instruction fetch

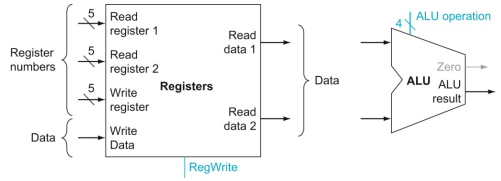


A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

R-format instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

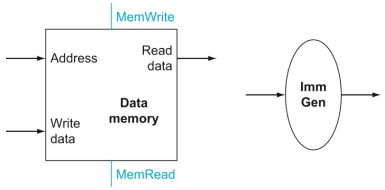


a. Registers b. ALU

The two elements needed to implement R-format ALU operations are the register file and the ALU. The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. In contrast, a register write must be explicitly indicated by asserting the write control signal. The inputs carrying the register number to the register file are all 5 bits wide, whereas the lines carrying data values are 32 bits wide. The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4 bits wide. We will use the Zero detection output of the ALU shortly to implement conditional branches.

Load/store instructions

- Read register operands
- Calculate address using 12-bit offset
  - use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



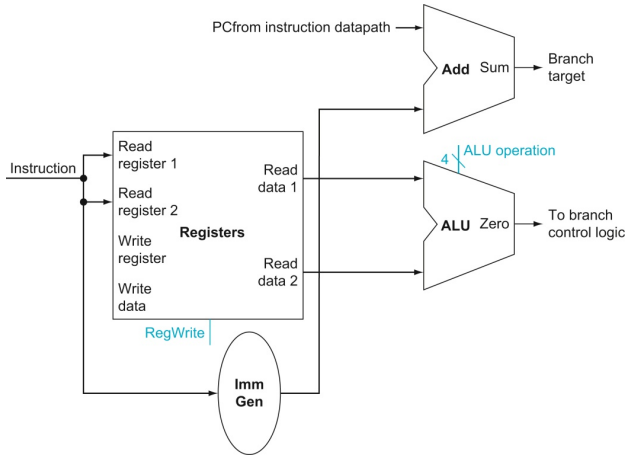
a. Data memory unit b. Immediate generation unit

The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, although only one of these may be asserted on any given clock.

Branch instructions

- Read register operands
- Compare operands
  - use ALU, subtract and check Zero output
- Calculate target address
  - sign-extend displacement
  - shift left 1 place (halfword displacement)
  - add to PC value

Branch instructions

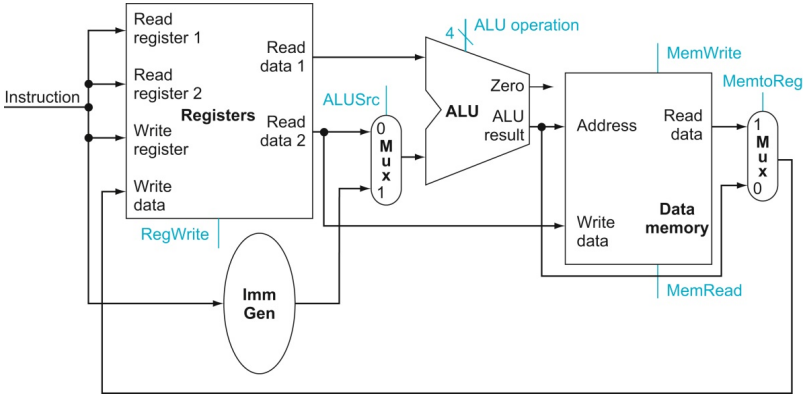


The portion of a datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the PC and immediate. Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

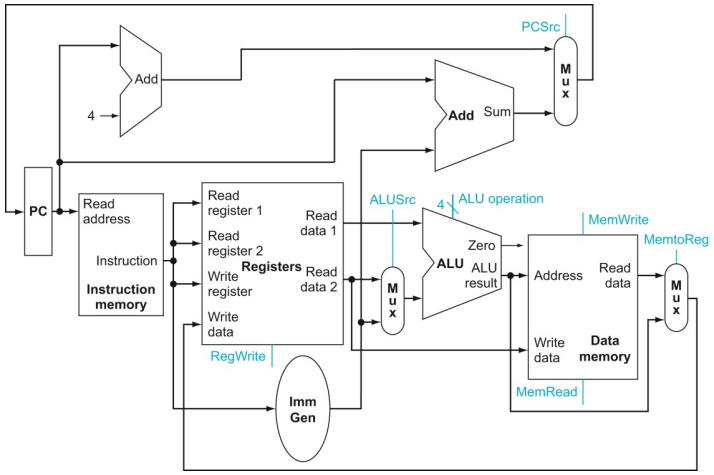
Composing the elements

- First-cut data path does an instruction in one clock cycle
  - each datapath element can only do one function at a time
  - hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

R-type/load/store datapath



Full datapath



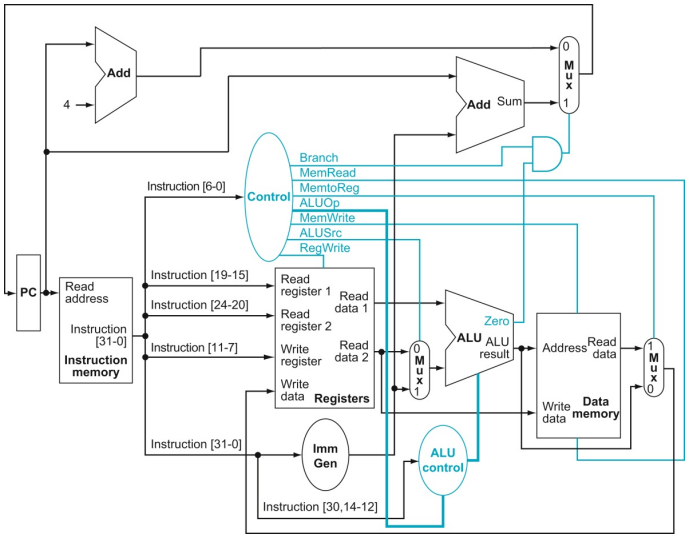
This datapath can execute the basic instructions (load-store register, ALU operations, and branches) in a single clock cycle. Just one additional multiplexor is needed to integrate branches.

Control signals from instruction

Name (Bit position)	Fields 31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

Datapath with control



Control lines

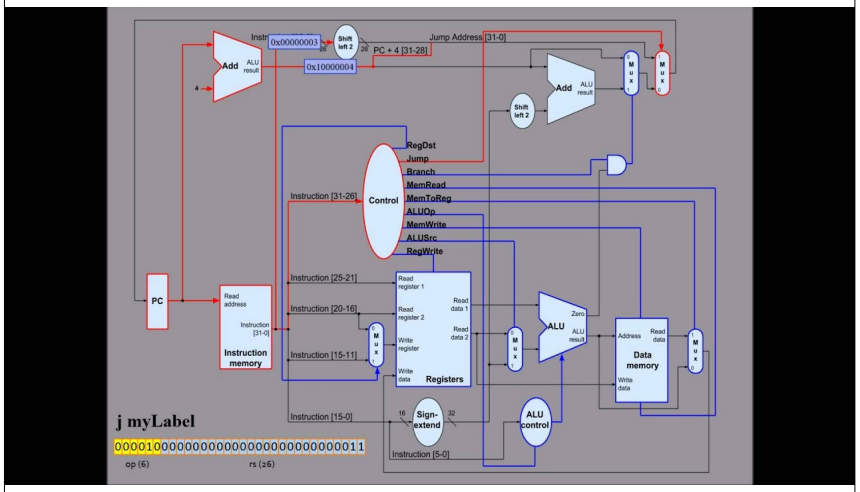
Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
lw	1	1	1	1	0	0	0	0
sw	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

The setting of the control lines is completely determined by the opcode fields of the instruction.

Performance issues

- Longest delay determines clock period
  - critical path: load instruction
  - instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates design principle
  - making the common case fast
- We will improve performance by **pipelining**

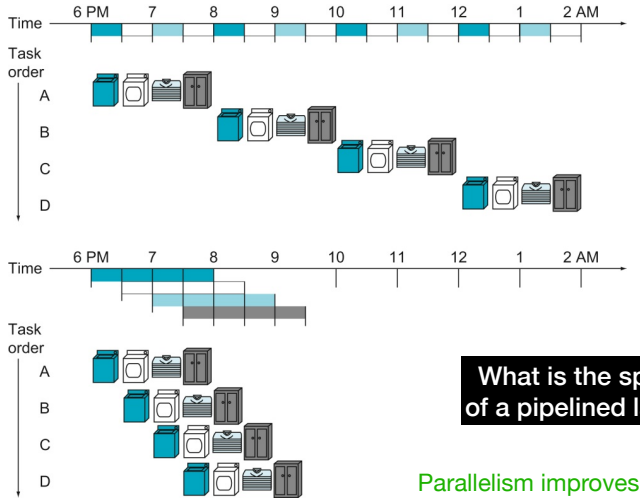
Animation on Youtube (MIPS)



https://youtu.be/oETOwVBzu1s?t=328

Pipelined execution

Pipelining analogy



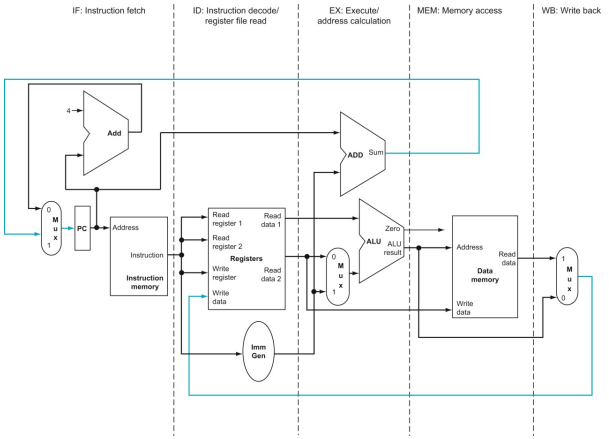
What is the speedup of a pipelined laundry?

Parallelism improves performance

RISC-V pipeline

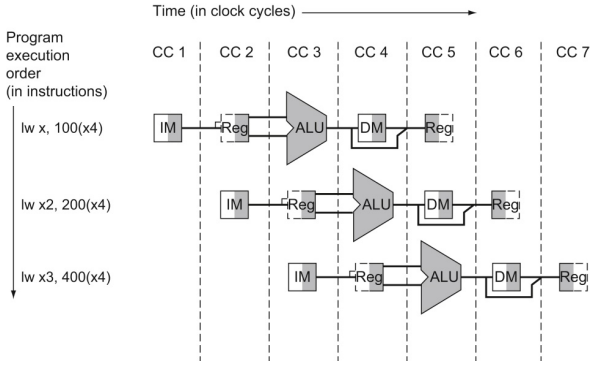
- Five stages, one step per stage
  - **IF**: instruction fetch from memory
  - **ID**: instruction decode & register read
  - **EX**: execute operation or calculate address
  - **MEM**: access memory operand
  - **WB**: write result back to register

Datapath with pipeline stages



Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from memory to the left to be written into the register file. Normally we use color lines for control, but these are data lines.

Pipelined execution



Each stage is labeled by the physical resource used in that stage. To maintain proper time order, this stylized datapath breaks the register file into two logical parts: registers read during register fetch (ID) and registers written during write back (WB). This dual use is represented by drawing the unshaded left half of the register file using dashed lines in the ID stage, when it is not being written, and the unshaded right half in dashed lines in the WB stage, when it is not being read. As before, we assume the register file is written in the first half of the clock cycle and the register file is read during the second half.

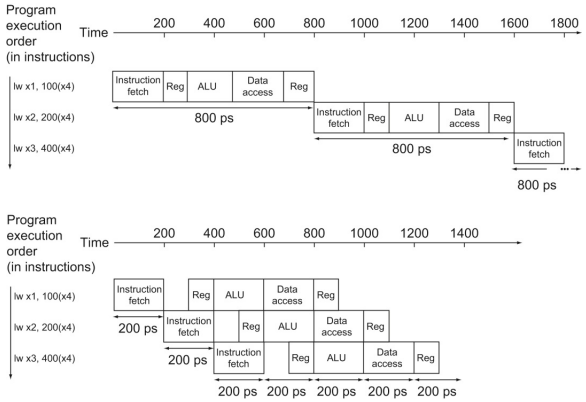
Pipeline performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

Lets compare pipelined datapath with single-cycle datapath

Pipeline performance



Both use the same hardware components, we see a fourfold speed-up on average time between instructions, from 800 ps down to 200 ps. The pipeline stage times of a computer are also limited by the slowest resource, either the ALU operation or the memory access. We assume the write to the register file occurs in the first half of the clock cycle and the read from the register file occurs in the second half. We use this assumption throughout this chapter.

Pipeline speedup

- If all stages are balanced
  - i.e., all take the same time
  - time between pipelined instructions is the time between non-pipelined instructions divided by the number of stages
- If not balanced, speedup is less
- Speedup due to increased **throughput**
  - **Latency** (time for each instruction) does not decrease

Pipelining and ISA design

- RISC-V ISA designed for pipelining
  - all instructions are 32-bits
    - easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - few and regular instruction formats
    - can decode and read registers in one step
  - load/store addressing
    - can calculate address in 3rd stage, access memory in 4th stage