CSC 411

Computer Organization (Fall 2024)
Lecture 22: Basic CPU design

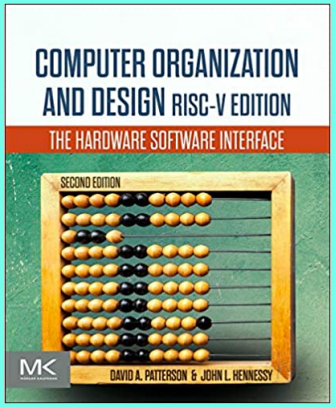
Prof. Marco Alvarez, University of Rhode Island

Disclaimer

Some figures and slides are adapted from:

Computer Organization and Design (Patterson and Hennessy)

The Hardware/Software Interface



Introduction

- We will study two RISC-V implementations
  - a simplified version (**single-cycle**)
  - a more realistic **pipelined** version
- Focusing on a small instruction set (subset of an actual RISC-V implementation)
  - memory access: `lw, sw`
  - arithmetic/logical: `add, sub, and, or`
  - control transfer: `beq, j`

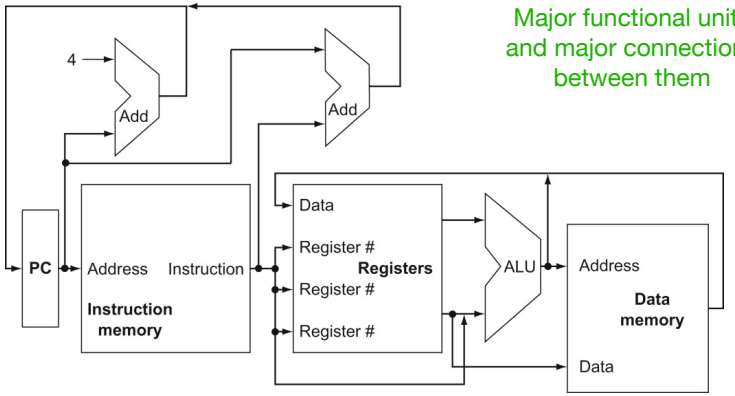
Single-cycle CPU

Instruction execution

- Program counter (PC)
  - requires access to instruction memory
  - PC determines address for instruction **“fetching”**
  - must be updated accordingly (PC+4 or target address)
- Register file
  - after **“decoding”** instruction, operands can be read from the register file
- ALU
  - depending on the type of instruction, use the ALU for **“executing”** the instruction
  - may require accessing data memory for load/stores
  - may write back to the register file
- Control logic
  - outputs values to control lines from instruction encoding

The big picture

add x3, x4, x5

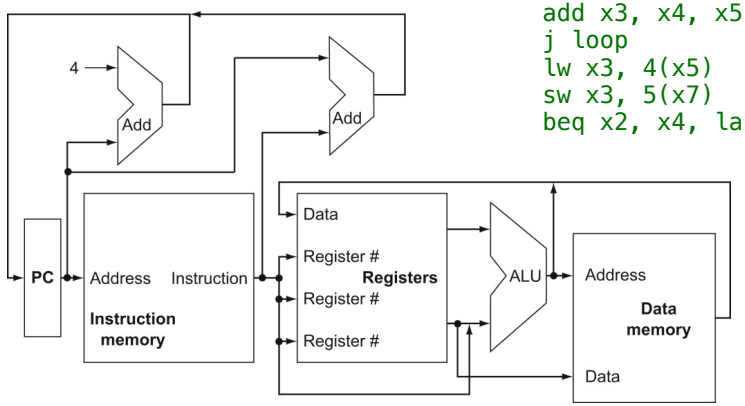


Major functional units and major connections between them

The thick lines interconnecting the functional units represent buses, which consist of multiple signals. The arrows are used to show information flows. A black dot indicates when crossing lines are connected.

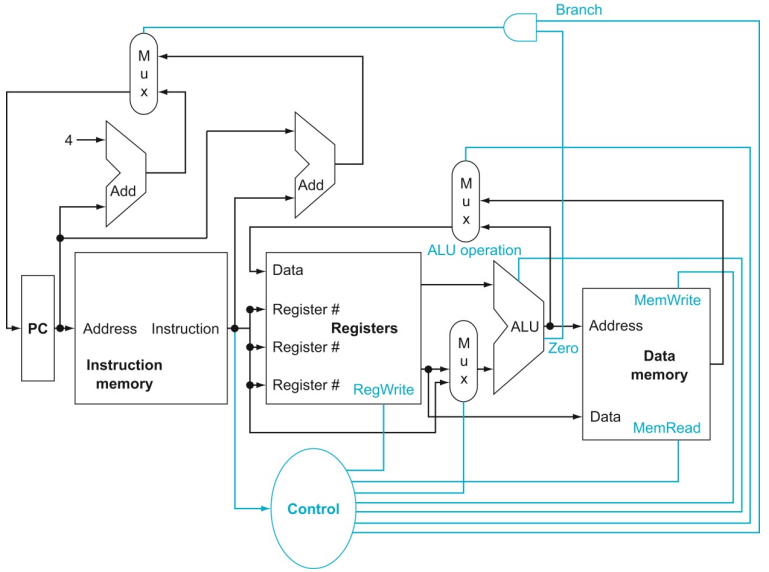
Practice

- Trace the flow of the following instructions




```
add x3, x4, x5
j loop
lw x3, 4(x5)
sw x3, 5(x7)
beq x2, x4, label
```

Adding multiplexors and control lines

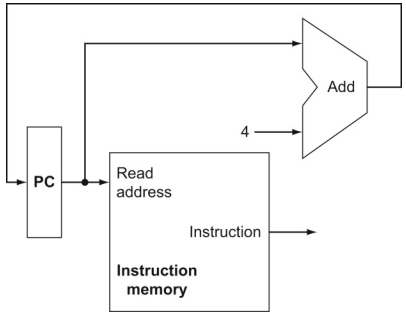


Building a datapath

- Datapath
  - elements that process data and addresses in the CPU
    - registers, ALUs, multiplexors, memories, …
- In the next slides, we will build a (toy) RISC-V datapath incrementally
  - leading to the “big-picture” figure from the previous slide

Instruction fetch

- State elements
  - PC register
    - 32-bit register written at the end of every clock-cycle
  - instruction memory
    - only read access, thus it can be treated as combinational logic
    - no control signal is needed
    - (when program is loaded, memory is written — ignored for simplicity)

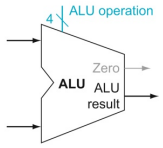


- Adder
  - always adds two 32-bit inputs

Implementing R-type instructions

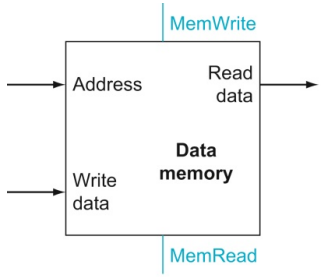
- Register file (state element)
  - always outputs the contents of the registers corresponding to the Read register inputs, no other control inputs are needed
  - a register write must be indicated to write the value to a register
  - writes are (falling) edge-triggered, therefore the design can legally read and write the same register within a clock cycle: the read gets the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle

- ALU
  - controlled with the ALU operation signal, which will be 4 bits wide
  - the Zero detection output is used to implement conditional branches

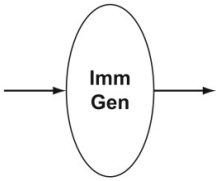


Implementing loads/stores

- Data memory unit (state element)
  - inputs: the memory address and the write data
  - output: the read result
  - edge-triggered for writes
  - separate read and write controls, although only one of these may be asserted on any given clock
    - a read signal is required, since, unlike the register file, reading the value of an invalid address can cause problems

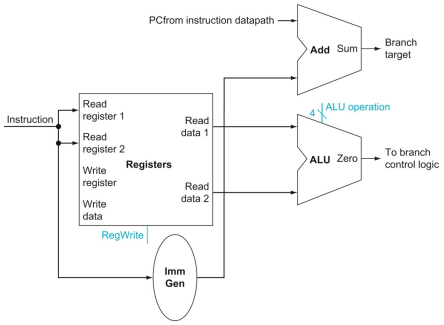


- Immediate generation unit (ImmGen)
  - selects a 12-bit field that is sign-extended into a 32-bit result

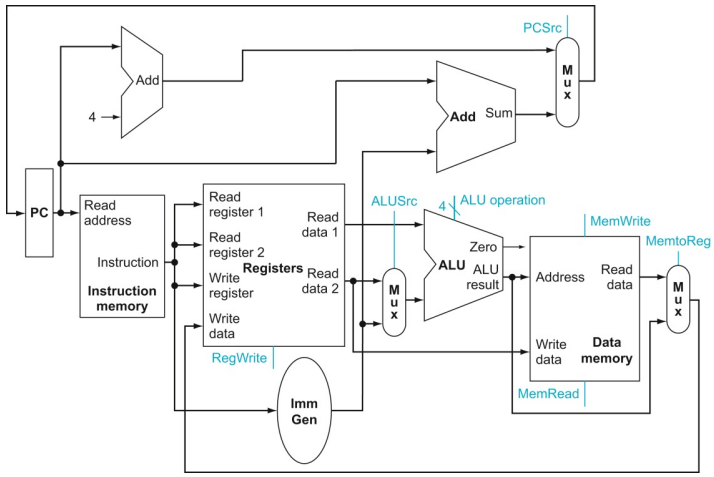


Implementing branch instructions

- Read and compare register operands
  - uses the ALU to evaluate the branch condition
- Calculate target address
  - uses a separate adder to compute the sum of the PC and immediate (the branch displacement)
  - 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



First-cut datapath



Executes an instruction in one clock cycle. Each datapath element can only do one function at a time, hence the need for separate instruction and data memories

ALU control

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

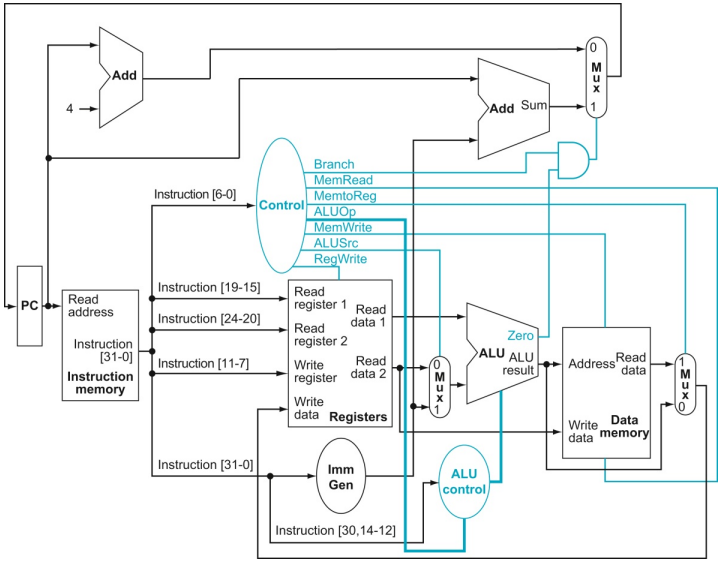
Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
lw	00	load word	XXXXXXX	XXX	add	0010
sw	00	store word	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000		AND	0000
R-type	10	or	0000000	110	OR	0001

"Don't care" bits are shown as X's

Control signals are derived 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

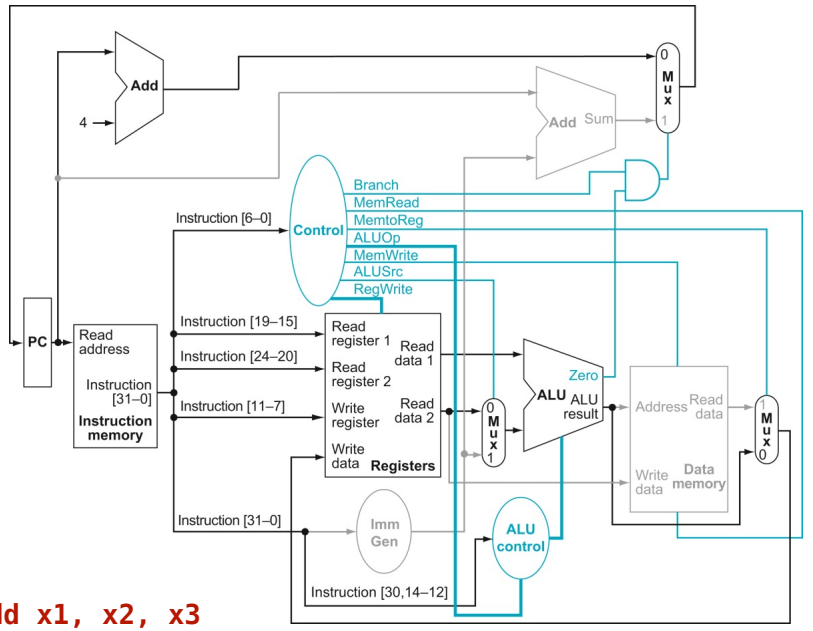
Datapath with control



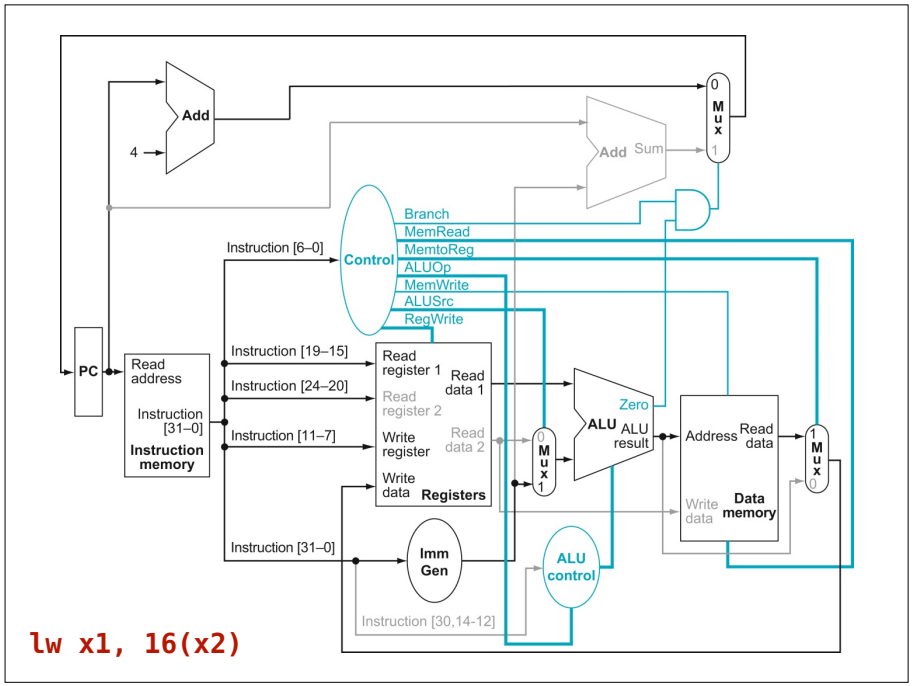
Performance issues

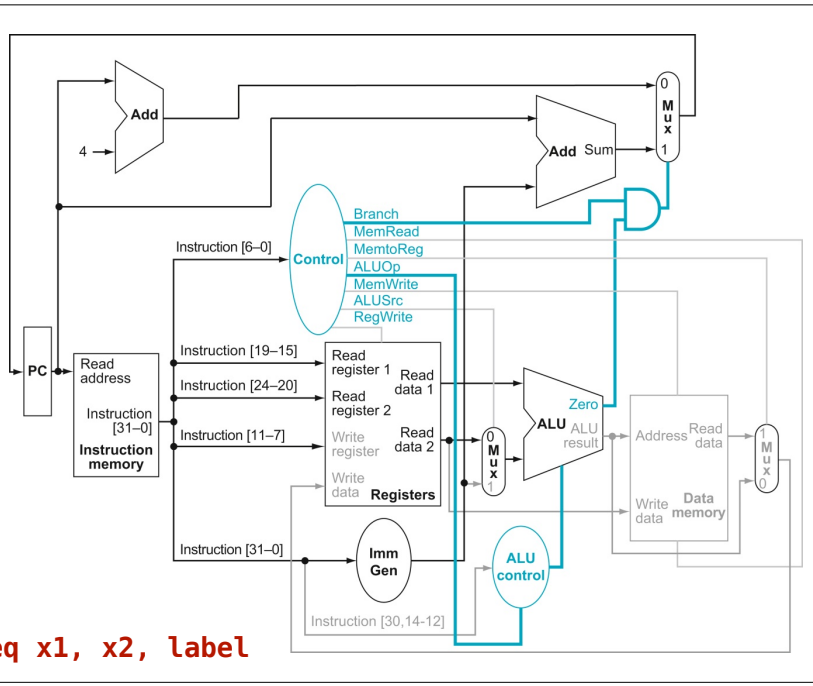
- Longest delay determines **clock period**
  - critical path: load instruction
    - instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- We will improve performance by **pipelining**

Examples



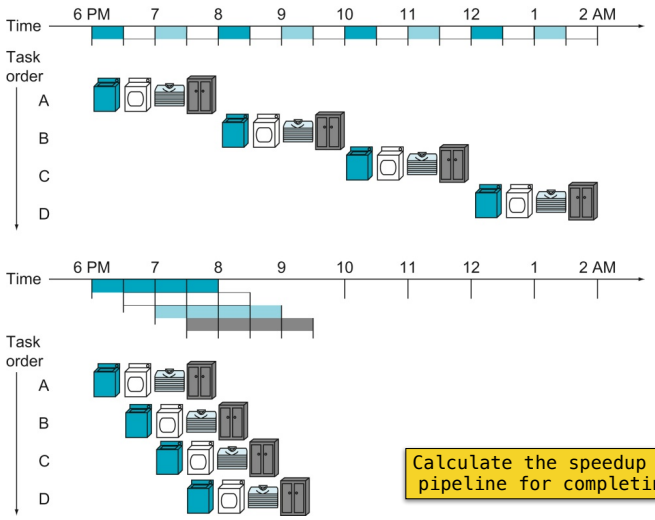
`add x1, x2, x3`





Pipeline execution

Pipelining (analogy)



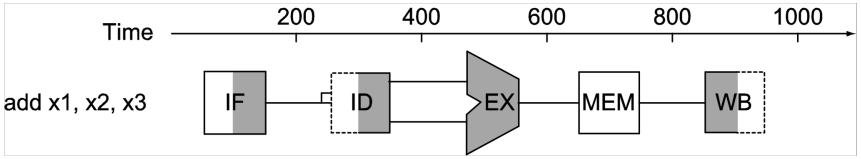
Calculate the speedup of using a pipeline for completing 4 loads

Parallelism improves performance

RISC-V pipeline

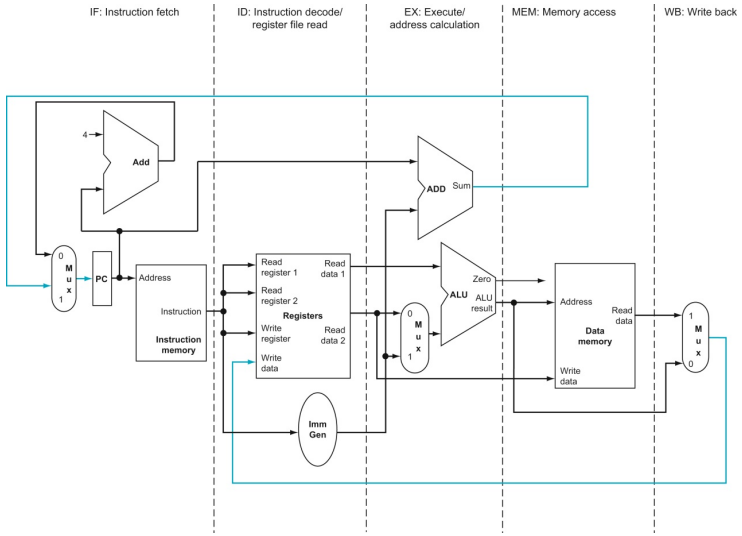
- Consider 5 stages (steps)
  - **IF:** instruction fetch from memory
  - **ID:** instruction decode & register read
  - **EX:** execute operation or calculate address
  - **MEM:** access memory operand
  - **WB:** write result back to register

Single-cycle execution

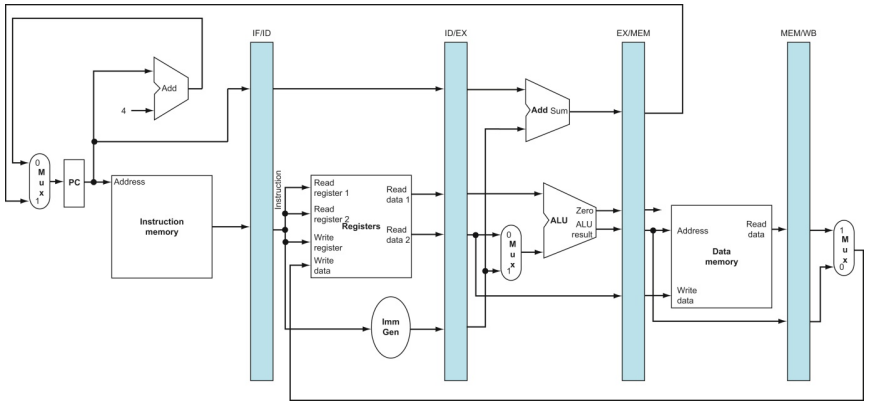


Resources used are shaded. Different instructions use different resources. Note also that that writes to the register file occur in the fist half of the cycle and reads occur in the second half.

Datapath with pipeline stages

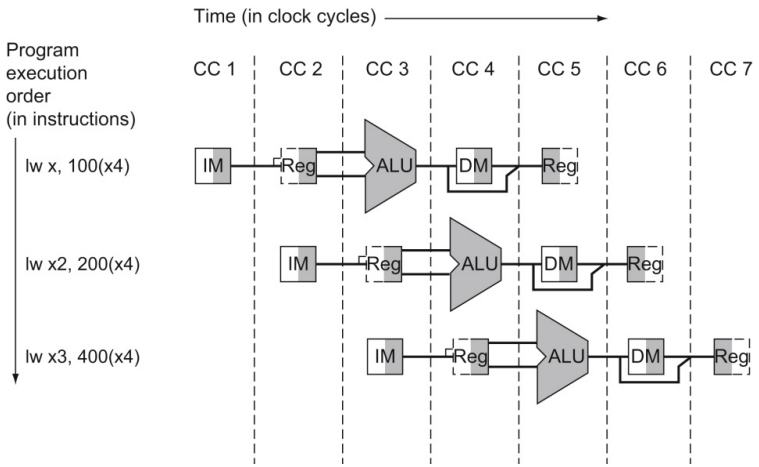


Adding pipeline registers



The pipeline registers, in color, are state elements that separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled IF/ID because it separates the instruction fetch and instruction decode stages.

Pipelined execution



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.

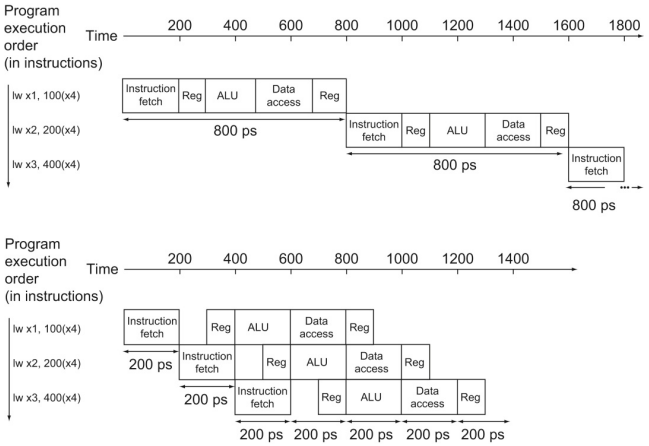
Pipeline performance

- Assume time per stage is:
  - 100ps for register read or write
  - 200ps for all 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

Note this calculation ignores the delays from multiplexers, PC access, control unit, and sign extension

Pipeline performance



Using the same hardware components we see a fourfold speed-up on average time between instructions, from 800 ps down to 200 ps. Stage times of a computer are 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.

Pipeline performance

- If all stages take the same time
  - time between pipelined instructions is the time between non-pipelined instructions divided by the number of stages
  - if stage times are not balanced, speedup is less
- Speedup due to increased **throughput**
  - **latency** (time for each instruction) does not decrease
    - it may go up, due to the need for latches at each stage
- Instructions take more cycles to finish
  - although **clock speed** goes high