What makes a fast processor?

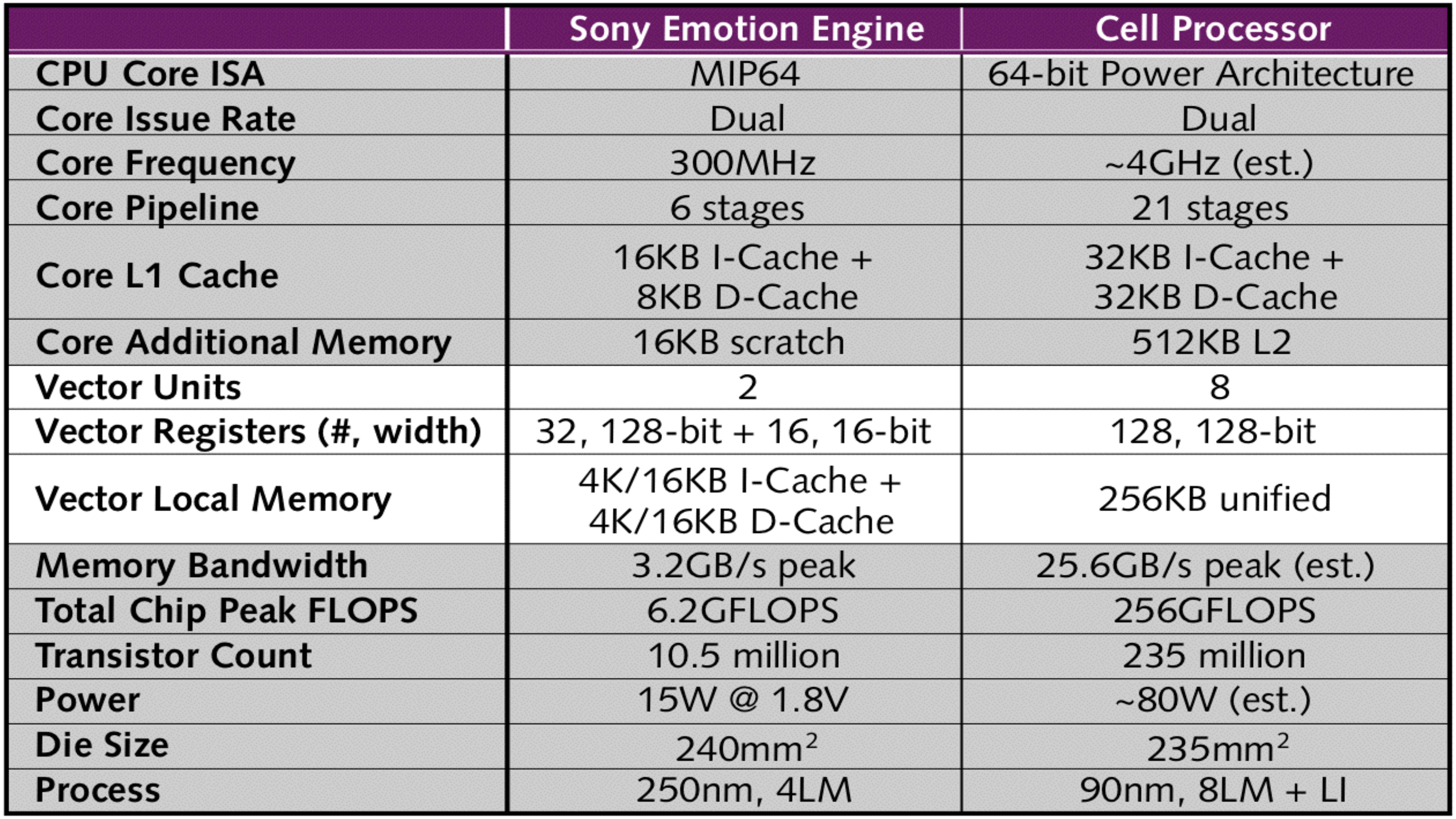
1. Instructions required per program
   - ISA design: RISC vs. CISC
2. Memory bandwidth and latency
   - Memory hierarchy
   - Cache parameterisation
3. Instructions executed per second
   - Internal CPU micro-architecture
   - De-coupled from memory and ISA
   - How clever can the designer get?

http://www.visual6502.org/JSSim/index.html

Pipelining: The search for GHz

- Early CPUs: single-cycle
  - *Lets just make it work; who cares about fast?*
  - Entire fetch-execute-retire process = 1 cycle/instruction
  - Built from discrete components or drawn by hand
- Micro-processors: multiple cycles per instruction
  - *Can we make this run faster than 1MHz?*
  - Fetch; then execute; then retire = 3+ cycles/instruction
  - Designed using first Electronic Design Automation tools
- 1990s: the pipeline is king
  - *We expect to be running at 10GHz by 2000...*
  - Multiple execute cycles; 20-30+ cycles/instruction
  - No single person understands the whole CPU...

Example: technology in PS2 and PS3



	Sony Emotion Engine	Cell Processor
CPU Core ISA	MIP64	64-bit Power Architecture
Core Issue Rate	Dual	Dual
Core Frequency	300MHz	~4GHz (est.)
Core Pipeline	6 stages	21 stages
Core L1 Cache	16KB I-Cache + 8KB D-Cache	32KB I-Cache + 32KB D-Cache
Core Additional Memory	16KB scratch	512KB L2
Vector Units	2	8
Vector Registers (#, width)	32, 128-bit + 16, 16-bit	128, 128-bit
Vector Local Memory	4K/16KB I-Cache + 4K/16KB D-Cache	256KB unified
Memory Bandwidth	3.2GB/s peak	25.6GB/s peak (est.)
Total Chip Peak FLOPS	6.2GFLOPS	256GFLOPS
Transistor Count	10.5 million	235 million
Power	15W @ 1.8V	~80W (est.)
Die Size	240mm^2	235mm^2
Process	250nm, 4LM	90nm, 8LM + LI

Source: Microprocessor Report: Feb 14, 2005

So is pipelining worth it?

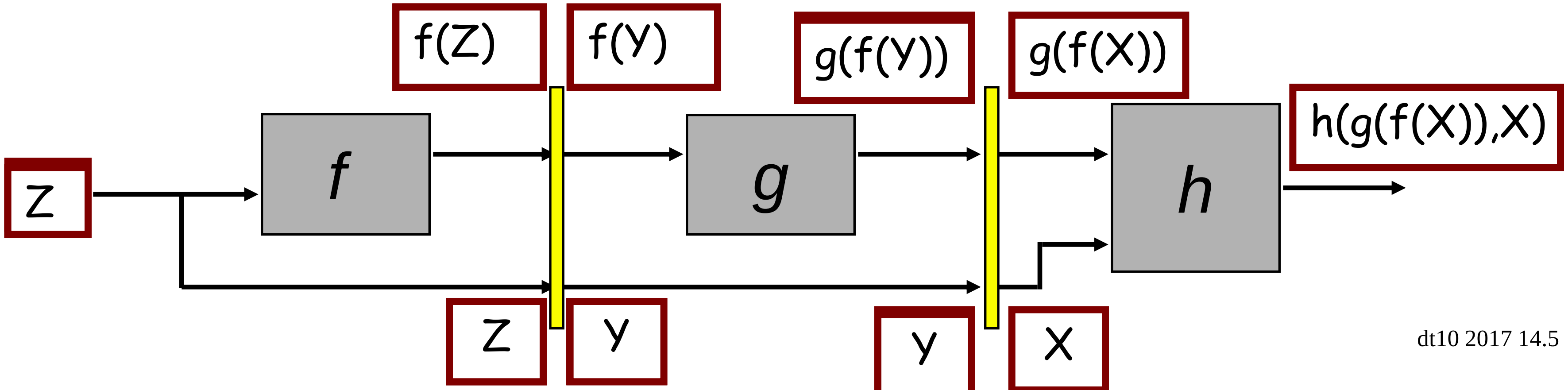
- Yes! Just don’t go overboard
  - All processors in use today are pipelined
  - What clock rate is the CPU in your phone?
- Pipelining is not just for performance
  - Power advantages due to reduced glitches
- Two main difficulties associated with pipelining
  1. MUST: Make sure processor still operates correctly
  2. TRY TO: Balance increased clock rate vs CPU stalls

– Theme of remaining lectures: pipelining & hazards

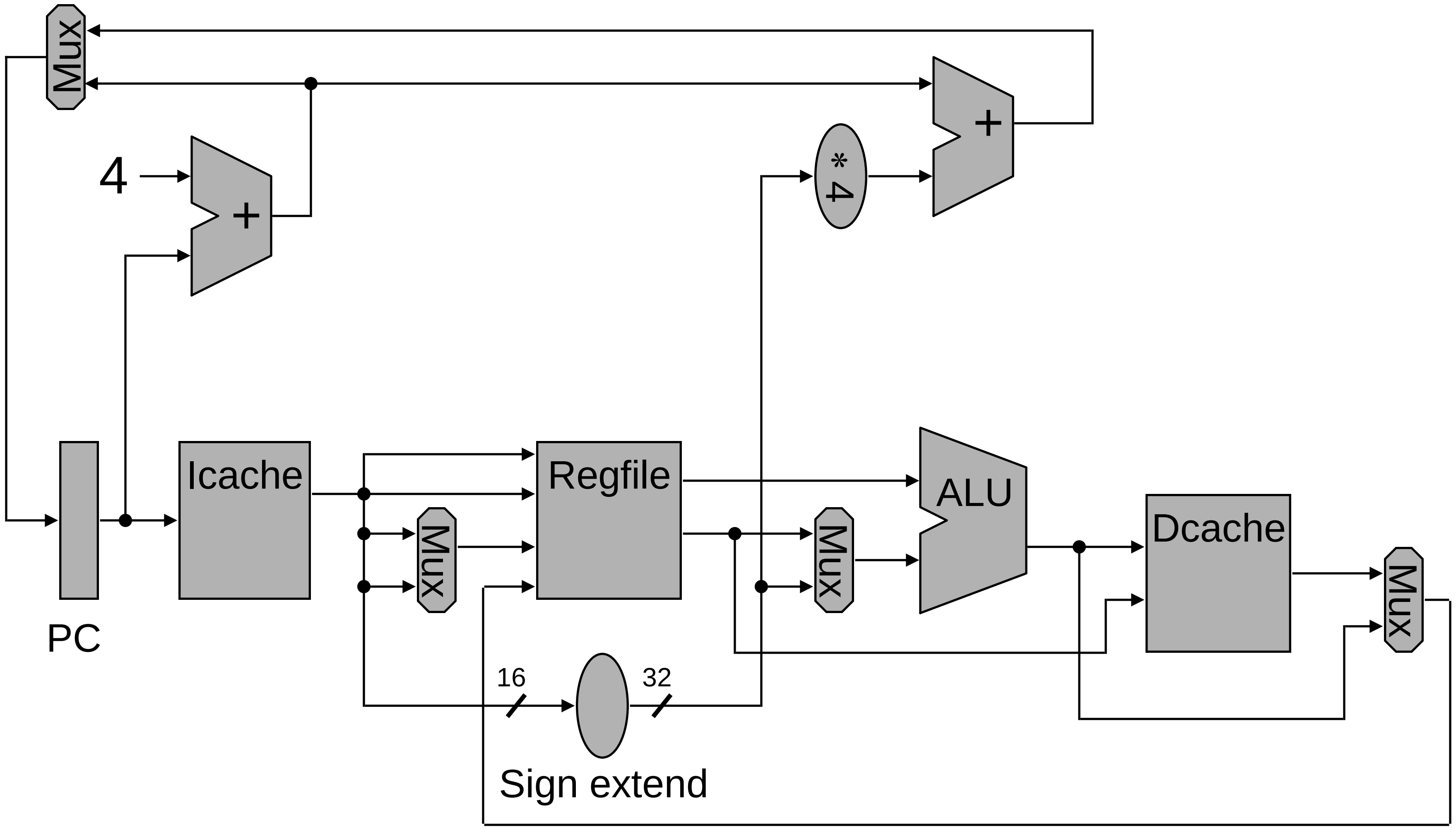
Pipelining

(3rd Ed: p.370-454, 4th Ed: p.330-409)

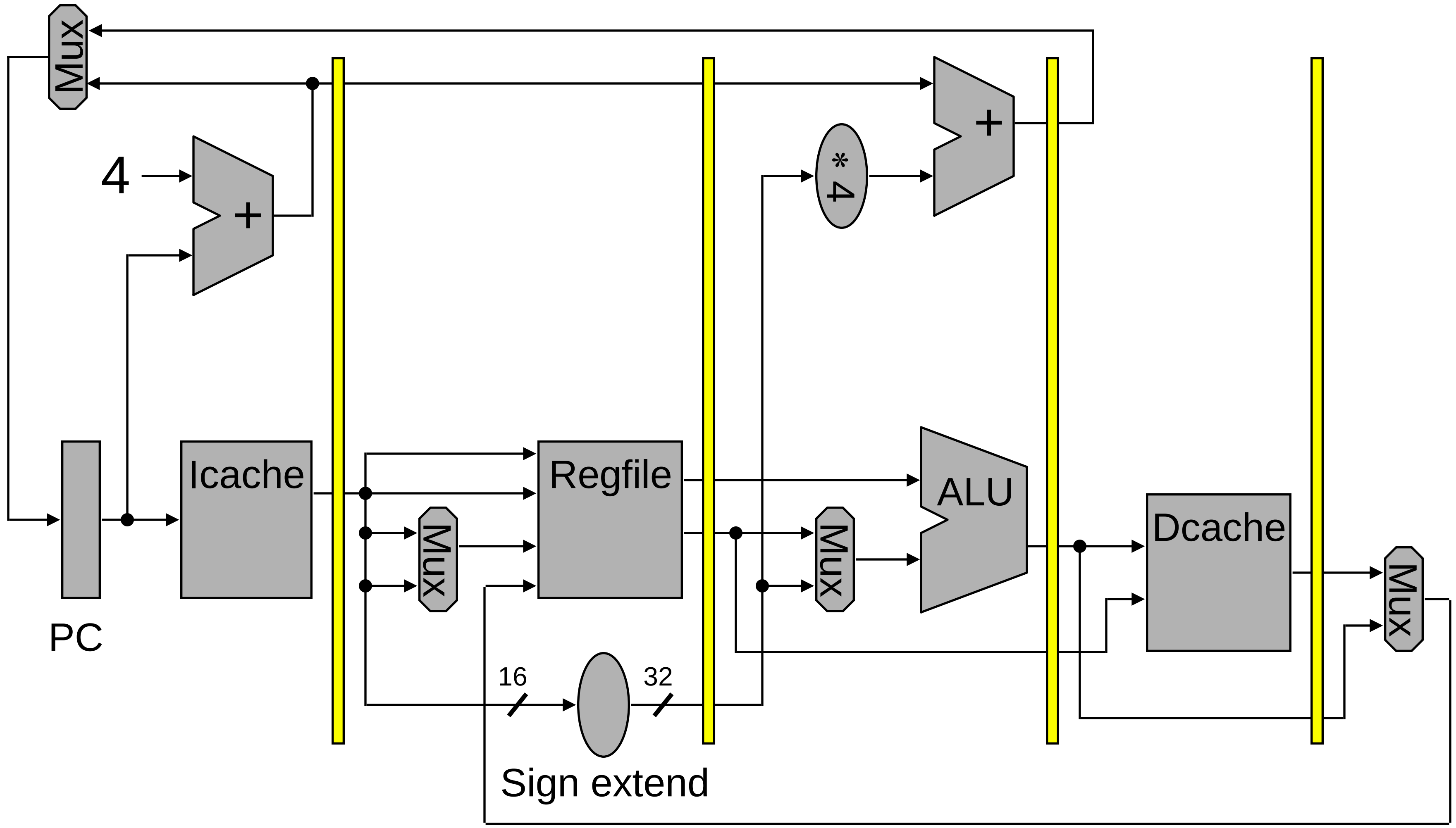
- split up combinational circuit by pipeline registers
- benefits
  - shorter cycle time, assembly-line parallelism
  - reduce power consumption by reducing glitches
- pipelined processor design
  - balance delay in different stages
  - resolve data and control dependencies



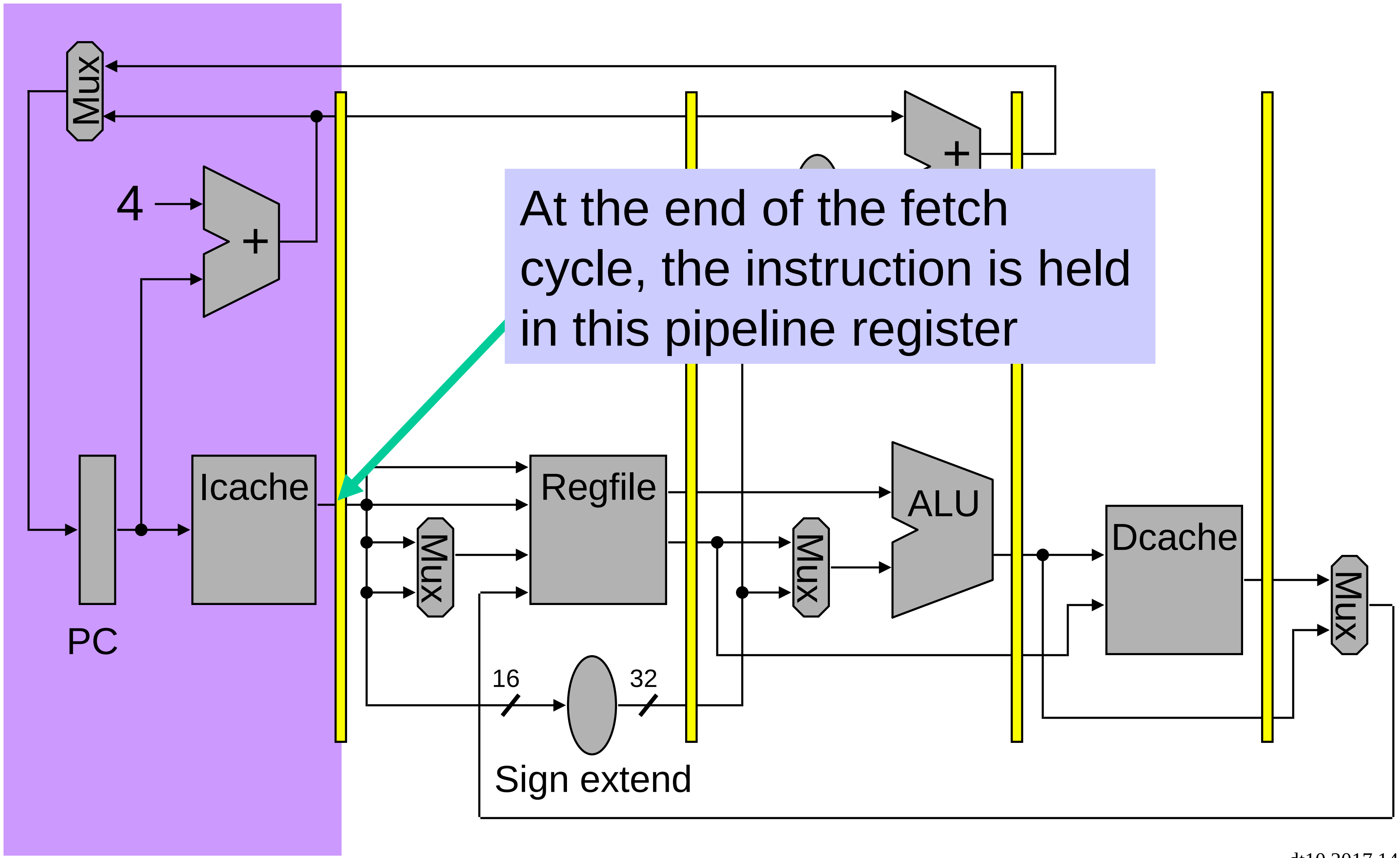
Single-cycle datapath



Pipelined datapath

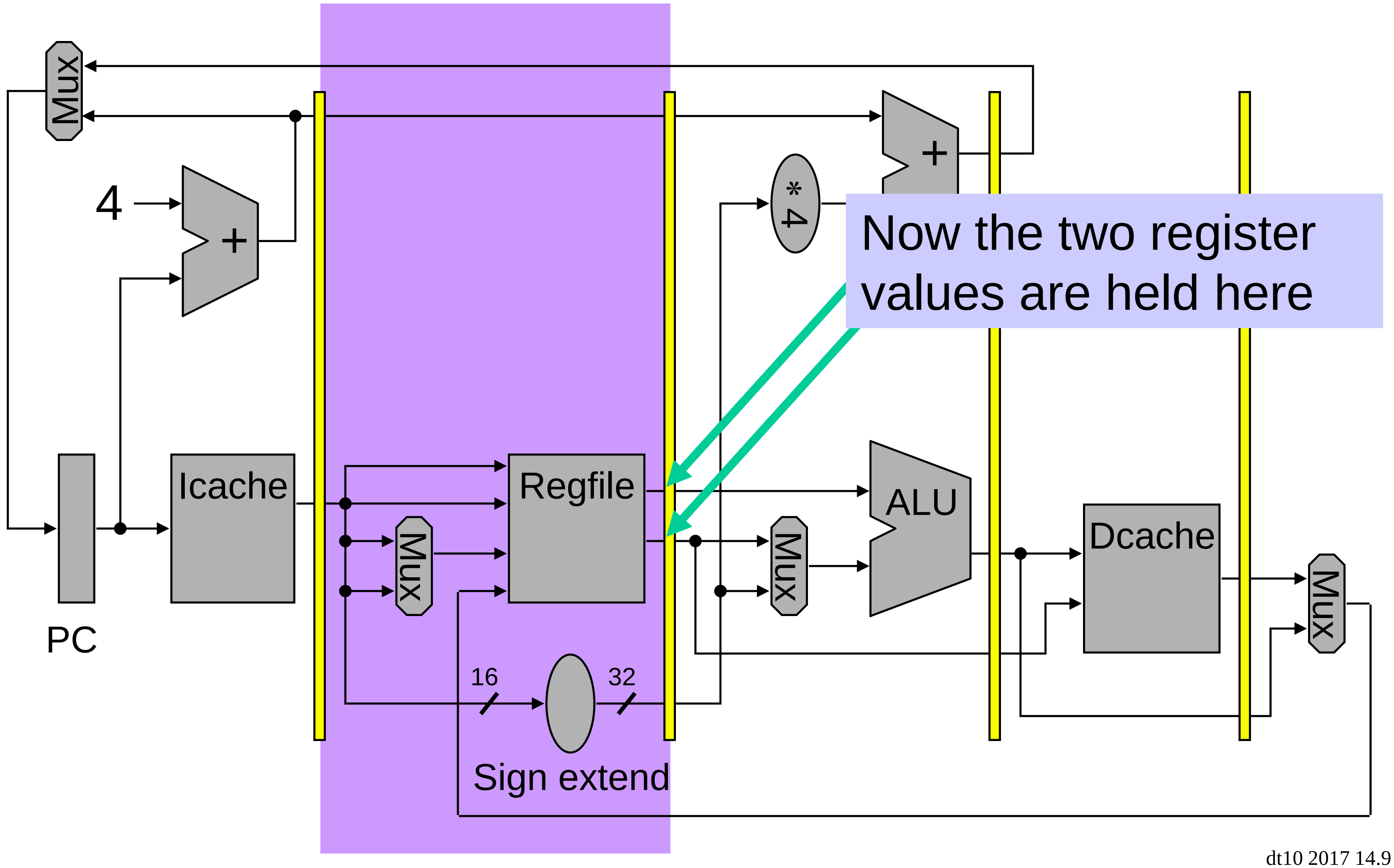


R-type instruction: fetch



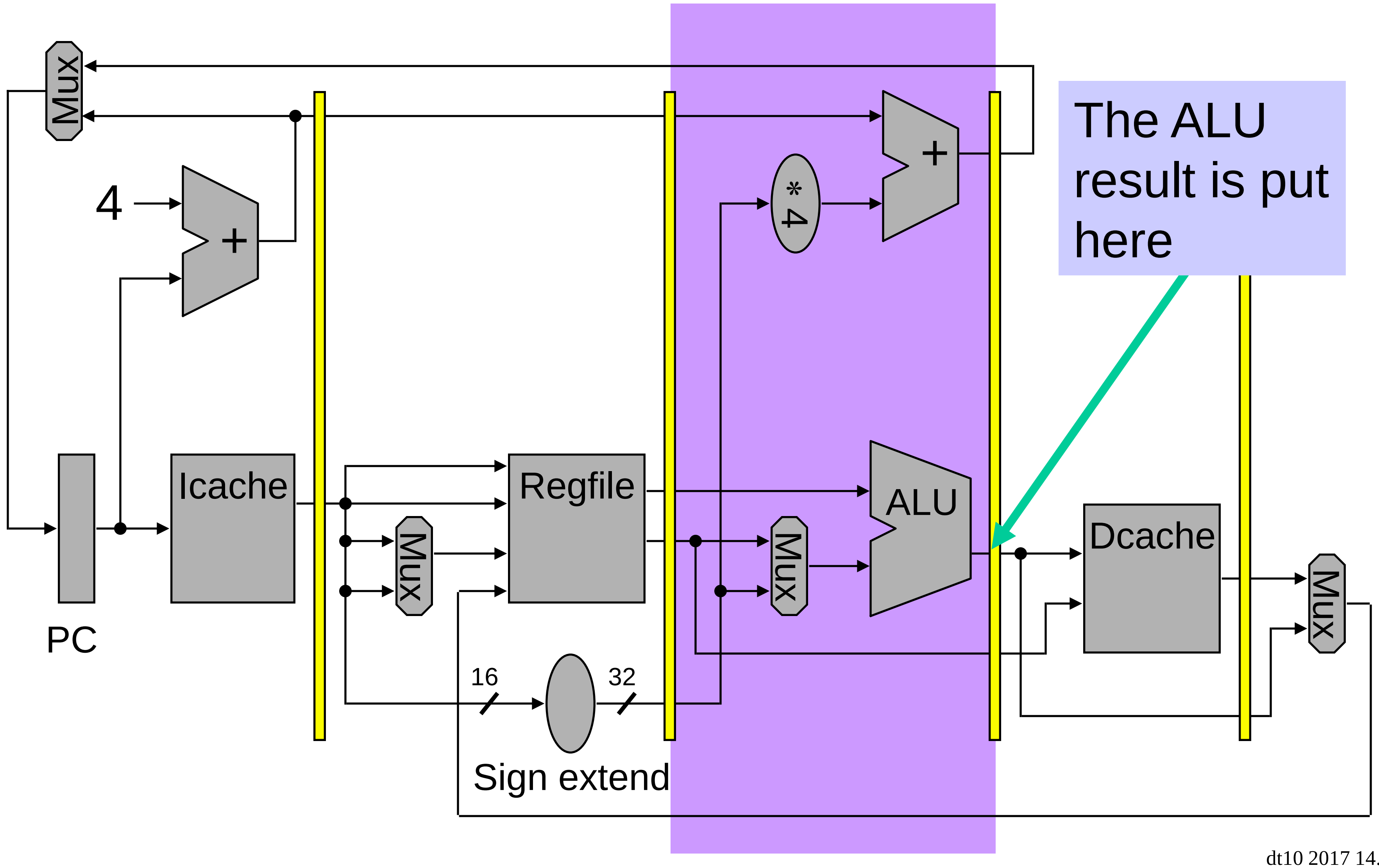
At the end of the fetch cycle, the instruction is held in this pipeline register

R-type instruction: register read



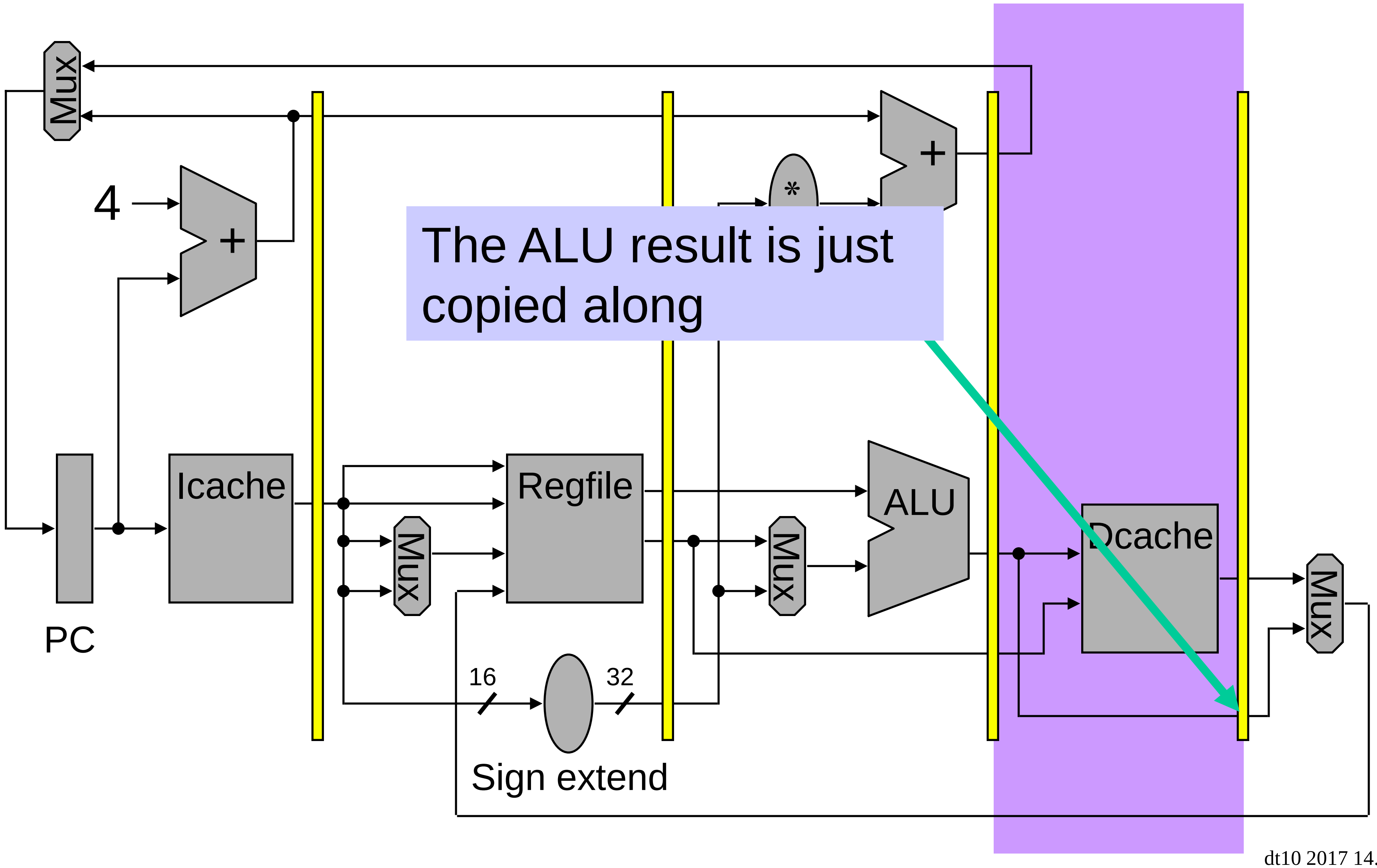
Now the two register values are held here

R-type instruction: execution

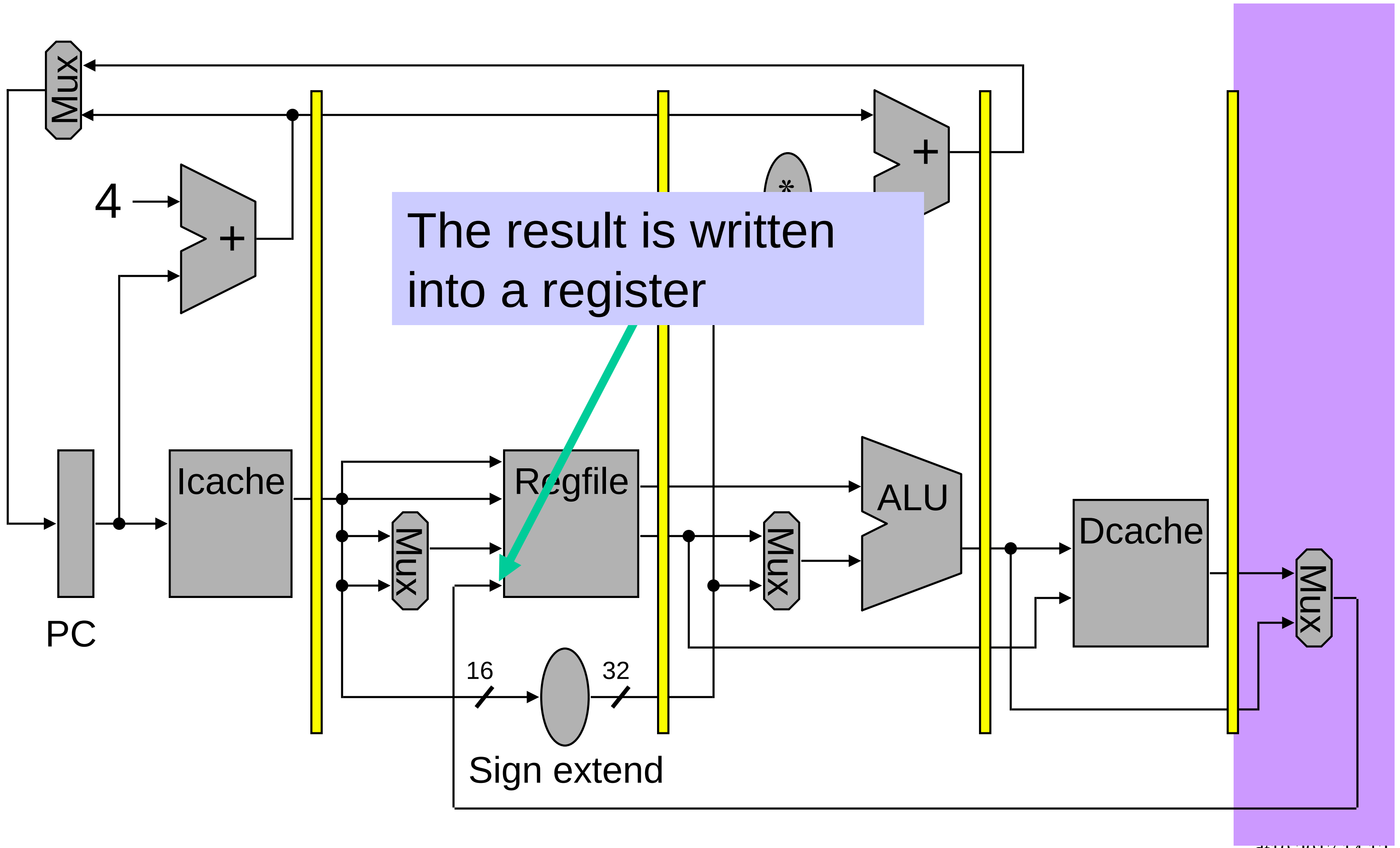


The ALU result is put here

R-type instruction: memory

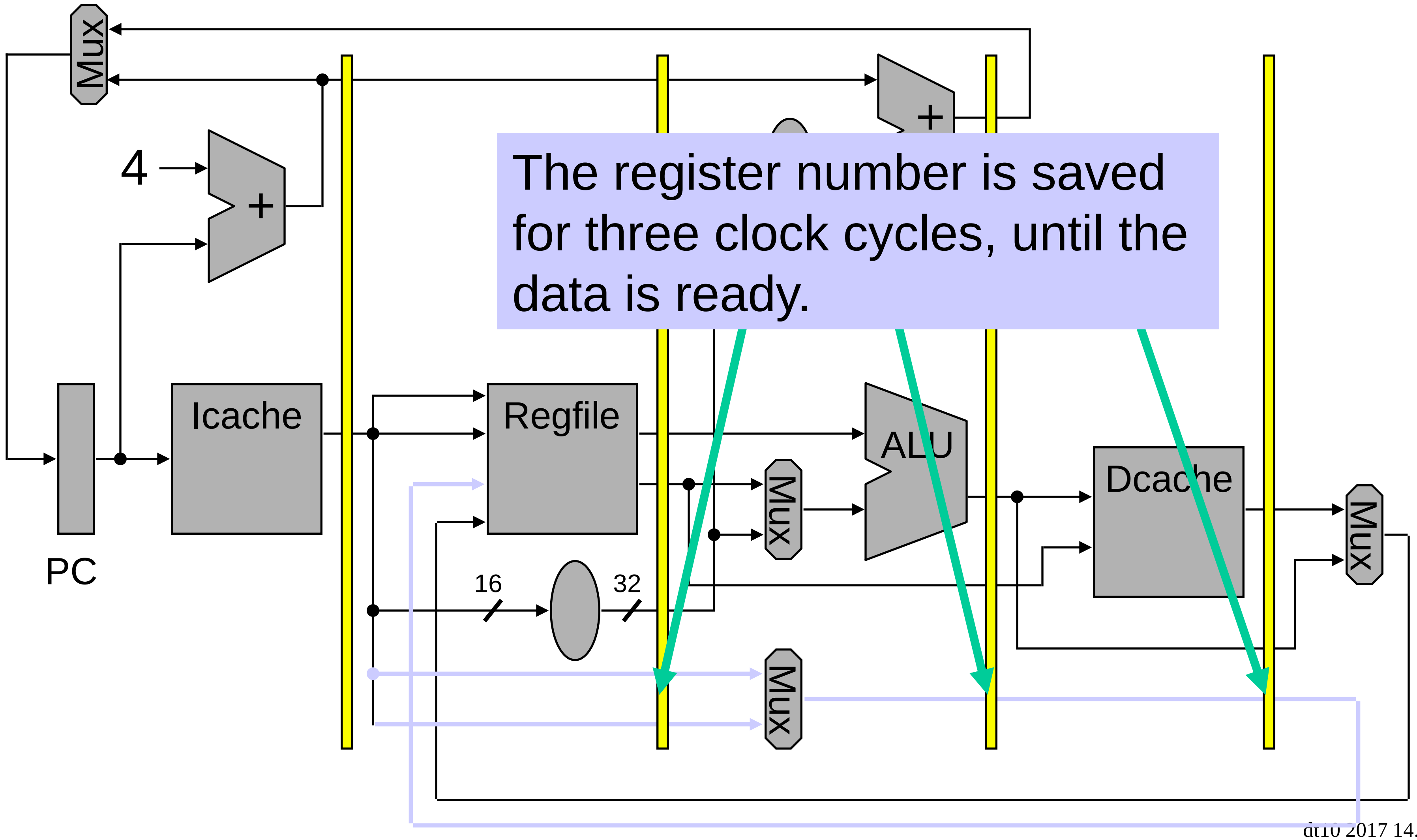


R-type instruction: write-back



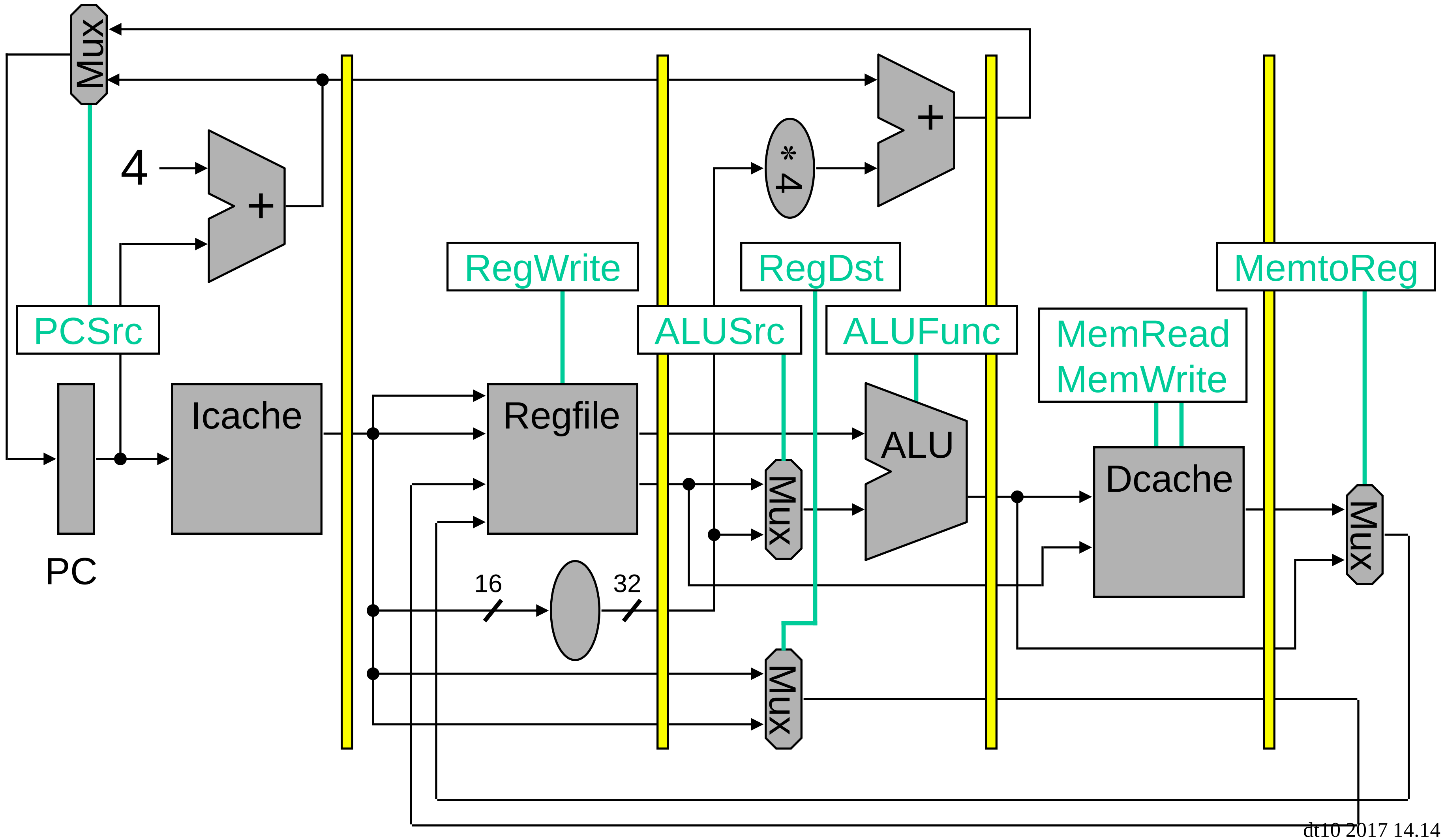
The result is written into a register

Writing the correct register

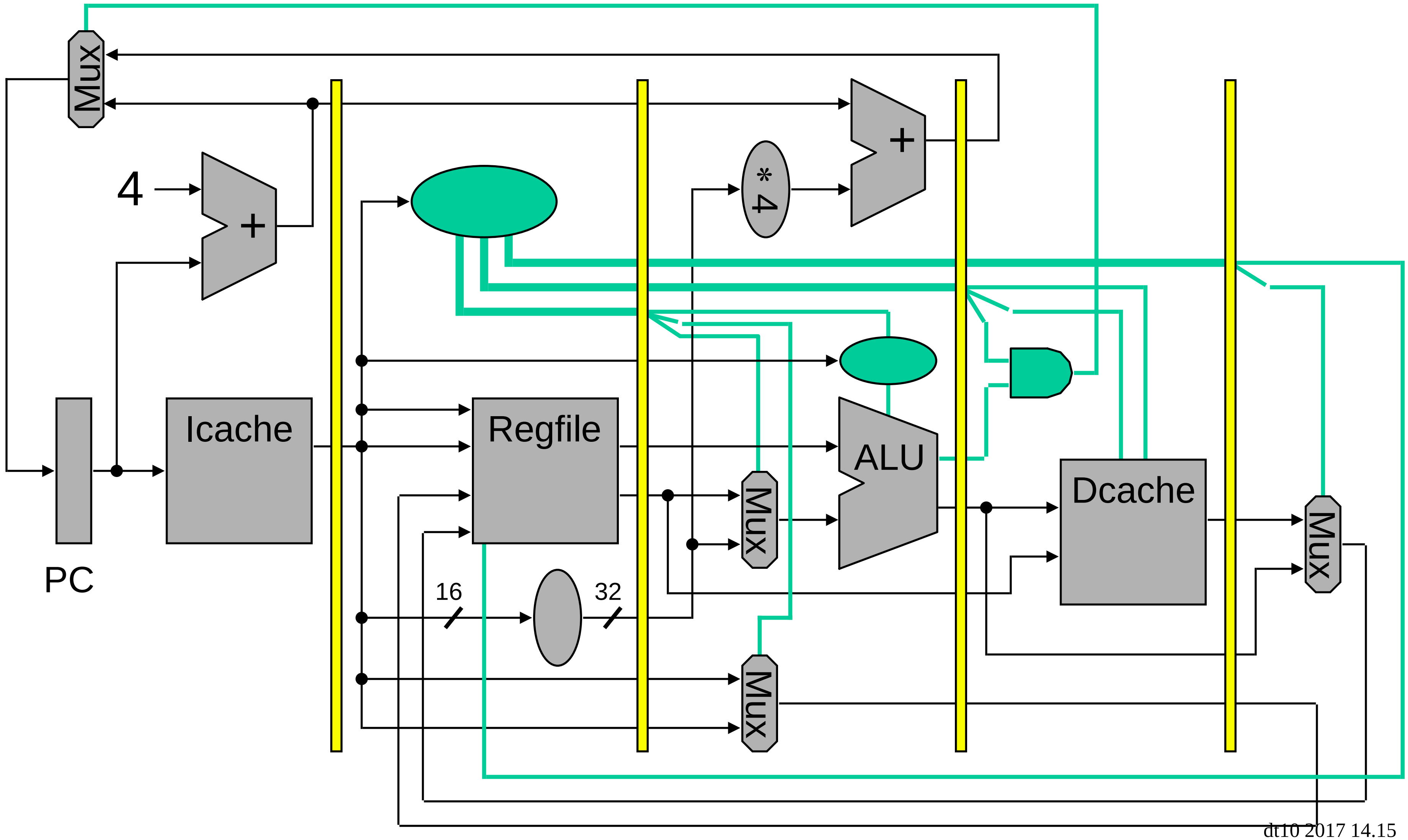


The register number is saved for three clock cycles, until the data is ready.

Control signals



Pipelined control



Performance issues

- longest delay determines clock period of processor
  - different instruction types use different sets of stages
  - critical path is load instruction: uses all stages

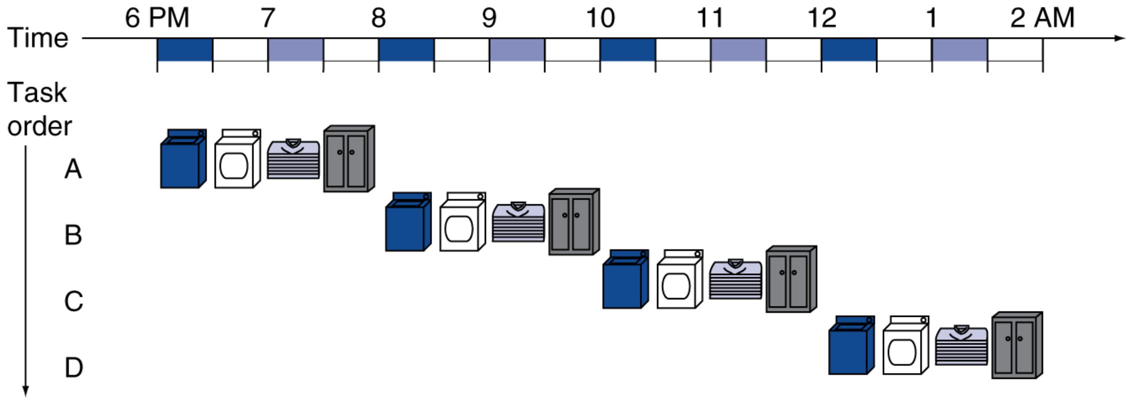
load = instr. mem. ► reg. file ► ALU ► data mem. ► reg. file

add = instr. mem. ► reg. file ► ALU ► *data mem.* ► reg. file

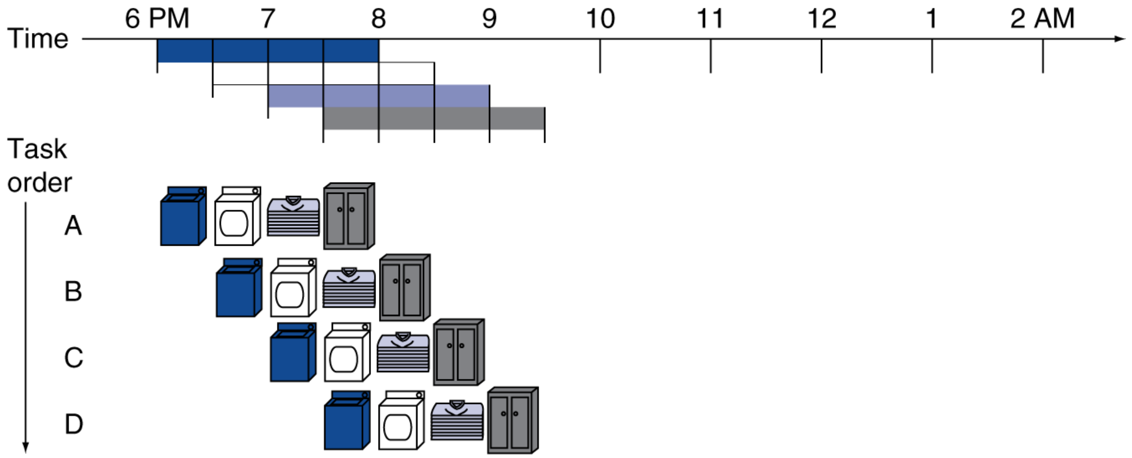
- difficult to vary clock period for each instruction
  - can be done, but quite uncommon
- violates design principle
  - making the common case fast
- most common solution: pipelining
  - other solutions exist: e.g. GALS, self-timed logic

Pipelining analogy

- pipelined laundry: overlapping execution
  - parallelism improves performance



- 4 loads:
  - speedup
    = 8/3.5 = 2.3



- non-stop:
  - Speedup
    = 2n/0.5n + 1.5 ≈ 4
    = number of stages

MIPS pipeline

- Five stages, one step per stage
  1. IF: **I**nstruction **F**etch from memory
  2. ID: **I**nstruction **D**ecode & register read
  3. EX: **Ex**ecute operation or calculate address
  4. MEM: Access **mem**ory operand
  5. WB: **W**rite result **B**ack to register

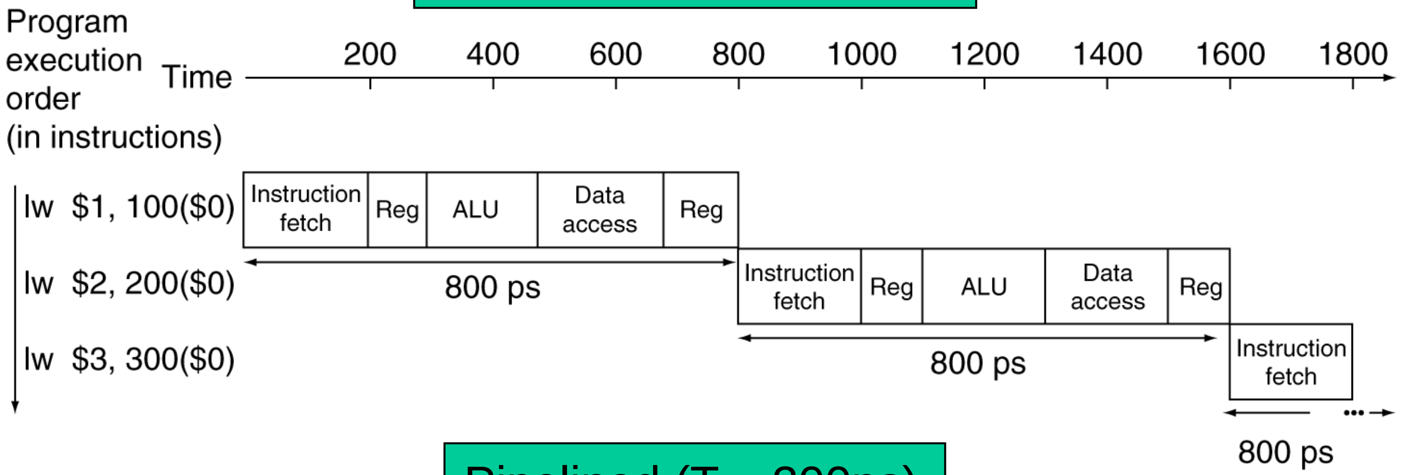
Pipeline performance: analysis

- assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- compare pipelined datapath with single-cycle datapath

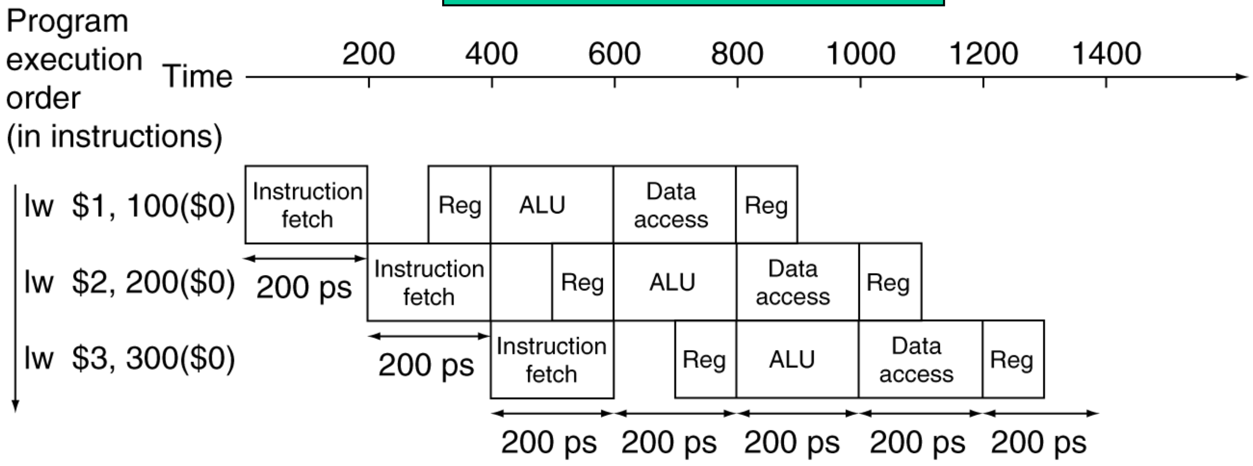
Instr. Type	Instr. fetch (IF)	Reg. read (ID)	ALU op. (EX)	Data mem. (MEM)	Reg. write (WB)	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline performance: comparison

Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)



Pipeline speedup

- assume: all stages are balanced
  - all take the same time
  - time between instructions$_{\text{pipelined}}$

$$= \frac{\text{time between instructions}_{\text{nonpipelined}}}{\text{number of stages}}$$

- if stages are not balanced, speedup is less
- speedup due to increased throughput
  - latency (time for each instruction) does not decrease
  - pipelining almost always increases latency a little...

Pipelining and ISA design

- MIPS ISA is designed for pipelining
- all instructions are 32-bits
  - Easier to fetch and decode in one cycle
  - contrast with x86: 1-byte to 17-byte instructions
- few and regular instruction formats
  - decode and read registers in one step
- load/store addressing
  - calculate address in 3rd stage, access memory in 4th stage
- alignment of memory operands
  - memory access takes only one cycle

Summary

- recap: pipelining
  - split hardware into stages
  - different instructions in different stages: parallelism
  - needs: store partial results, control instruction overlap
- to think about:
  - instruction requires value not yet provided by earlier instruction e.g. get outdated value
  - load the wrong instruction because of branching
  - exceptions
- (somewhat) recent developments
  - VLIW, out-of-order / predicated / speculative execution, MIMD
  - multi-core/multi-threaded processor, reusable custom instructions