Modern Processor Architecture

Lecture Goals

- Learn about the key techniques that modern processors use to achieve high performance

- Emphasize the techniques that may help you in the design project (e.g., vector/SIMD instructions)

Reminder: Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \underbrace{\frac{\text{Cycles}}{\text{Instruction}}}_{\text{CPI}} \cdot \underbrace{\frac{\text{Time}}{\text{Cycle}}}_{t_{CLK}}$$

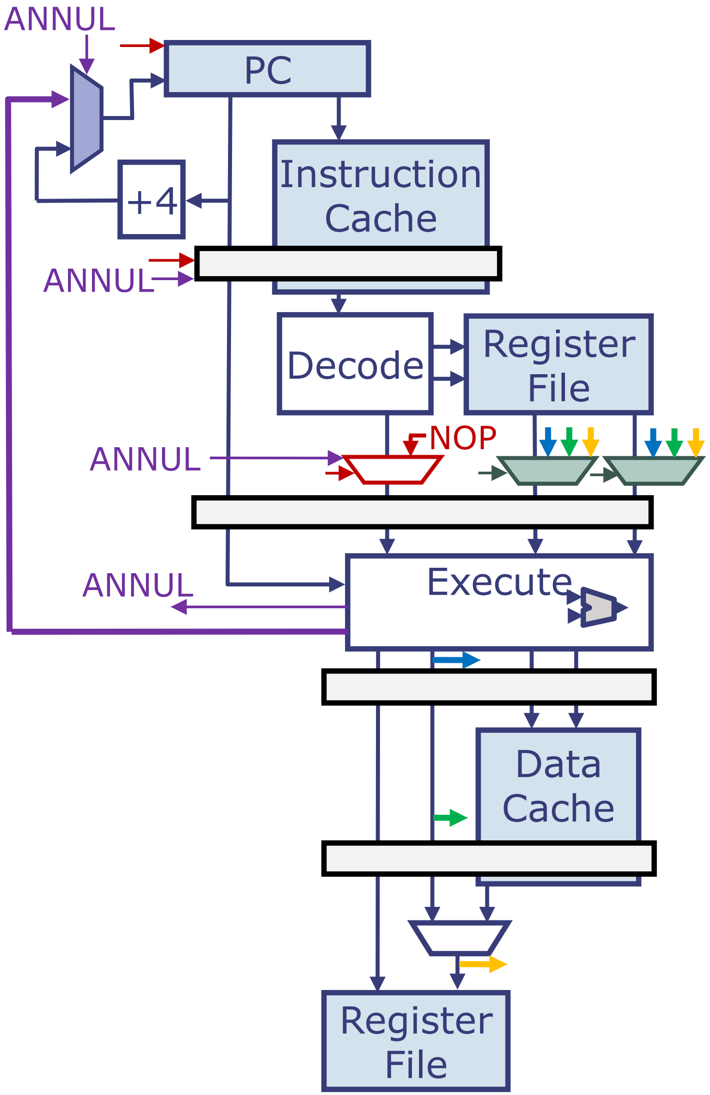
- Pipelining lowers t_{CLK}. What about CPI?

- CPI = CPI_{ideal} + CPI_{hazard}
  - CPI_{ideal}: cycles per instruction if no stalls

- CPI_{hazard} contributors
  - Data hazards: long operations, cache misses
  - Control hazards: branches, jumps, exceptions

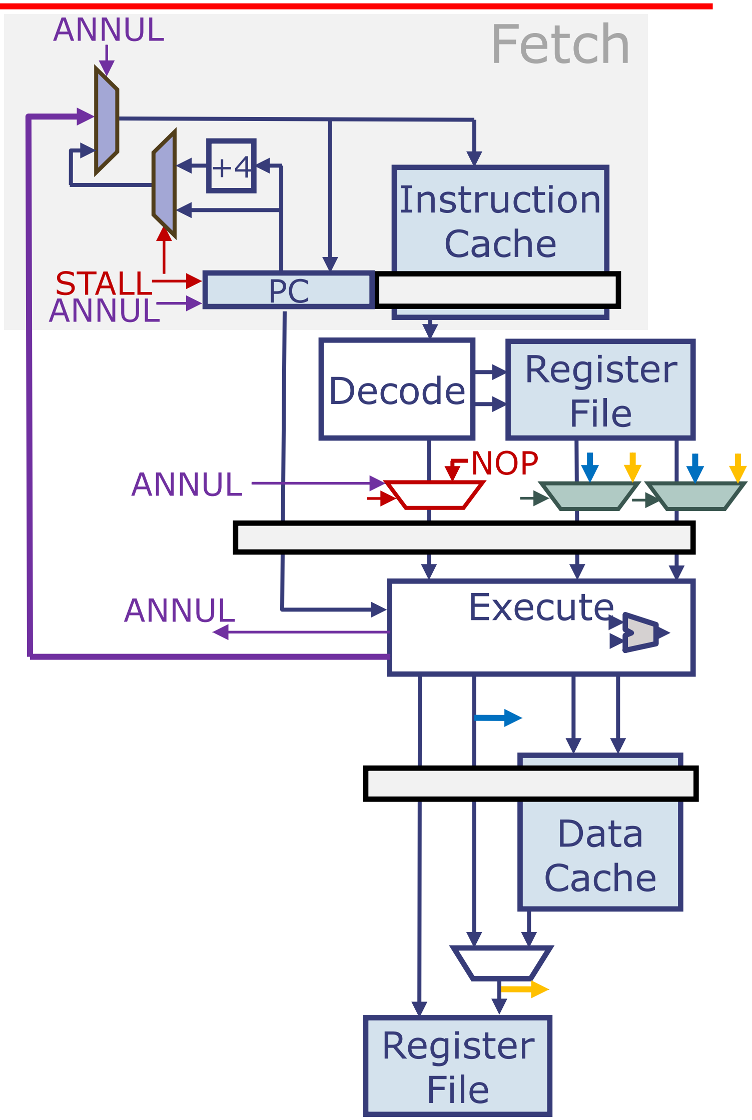
Standard 5-Stage Pipeline

- Assume full bypassing
- $\text{CPI}_{\text{ideal}}=1.0$
- $\text{CPI}_{\text{hazard}}$ due to data hazards:
  Up to how many cycles lost to each load-to-use hazard? 2
- $\text{CPI}_{\text{hazard}}$ due to control hazards:
  How many cycles lost to each jump and taken branch? 2



Lab 6 Pipeline

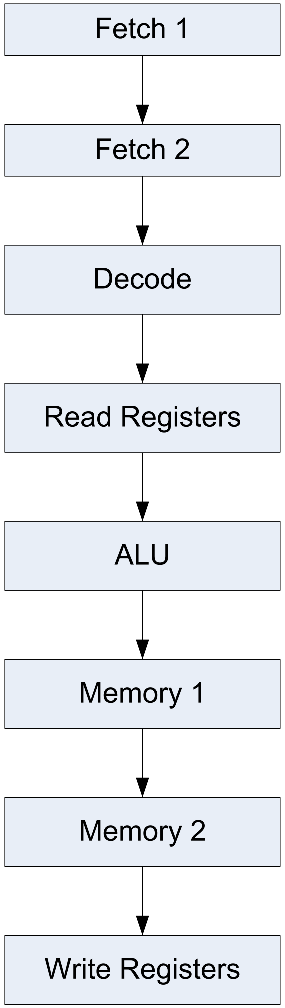
- 4 stages: IF, DEC, EXE, WB
  - No MEM stage
- $\text{CPI}_{\text{hazard}}$ due to data hazards: *Up to how many cycles lost to each load-to-use hazard?* ___1___
- IF uses *PC bypassing*: On annulment, IF starts fetching at the jump/branch target on the same cycle
- $\text{CPI}_{\text{hazard}}$ due to control hazards: *How many cycles lost to each jump and taken branch?* ___1___



Improving Processor Performance

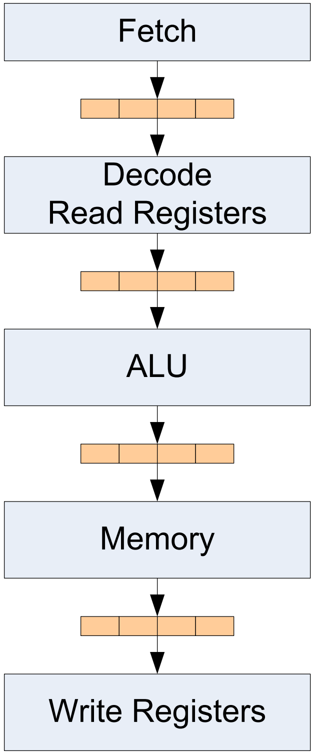
- Increase clock frequency: deeper pipelines
  - Overlap more instructions
- Reduce $\text{CPI}_{\text{ideal}}$: wider pipelines
  - Each pipeline stage processes multiple instructions
- Reduce impact of data hazards: out-of-order execution
  - Execute each instruction as soon as its source operands are available
- Reduce impact of control hazards: branch prediction
  - Predict both direction and target of branches and jumps
- Reduce executed instructions: ISA extensions
  - Add new instructions that perform more work

Deeper Pipelines



- Break up datapath into N pipeline stages
  - Ideal t_{CLK} = 1/N compared to non-pipelined
  - So let’s use a large N!
- Advantage: Higher clock frequency
  - The workhorse behind multi-GHz processors
  - Intel Skylake, AMD Zen2: 19 stages, 4-5 GHz
- Disadvantages
  - More overlapping ⇒ more dependencies
    - CPI_{hazard} grows due to data and control hazards
  - Pipeline registers add area & power

Wider (aka Superscalar) Pipelines



- Each stage operates on up to W instructions each clock cycle
- Advantage: Lower $\text{CPI}_{\text{ideal}}$ (1/W)
  - Skylake & Zen2: 6-wide, Apple M1: 8-wide
- Disadvantages
  - Parallel execution ⇒ more dependencies
    - $\text{CPI}_{\text{hazard}}$ grows due to data and control hazards
  - Much higher cost & complexity
    - More ALUs, register file ports, …
    - Many bypass & stall cases to check

Resolving Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages

- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated

- Strategy 3: Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value

- Strategy 4: Find something else to do

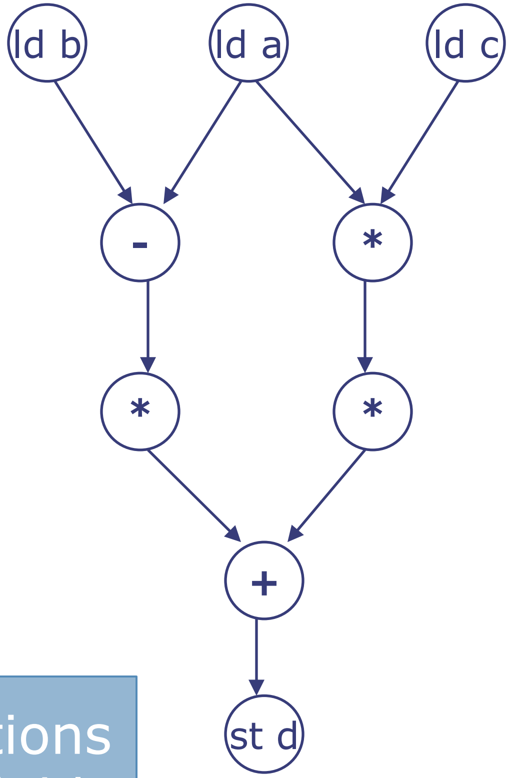
Out-of-Order Execution

- Consider the expression $D = 3(a-b) + 7ac$

Sequential code

```
ld a
ld b
sub a-b
mul 3(a-b)
ld c
mul ac
mul 7ac
add 3(a-b)+7ac
st d
```

Dataflow graph



Out-of-order execution runs instructions as soon as their inputs become available

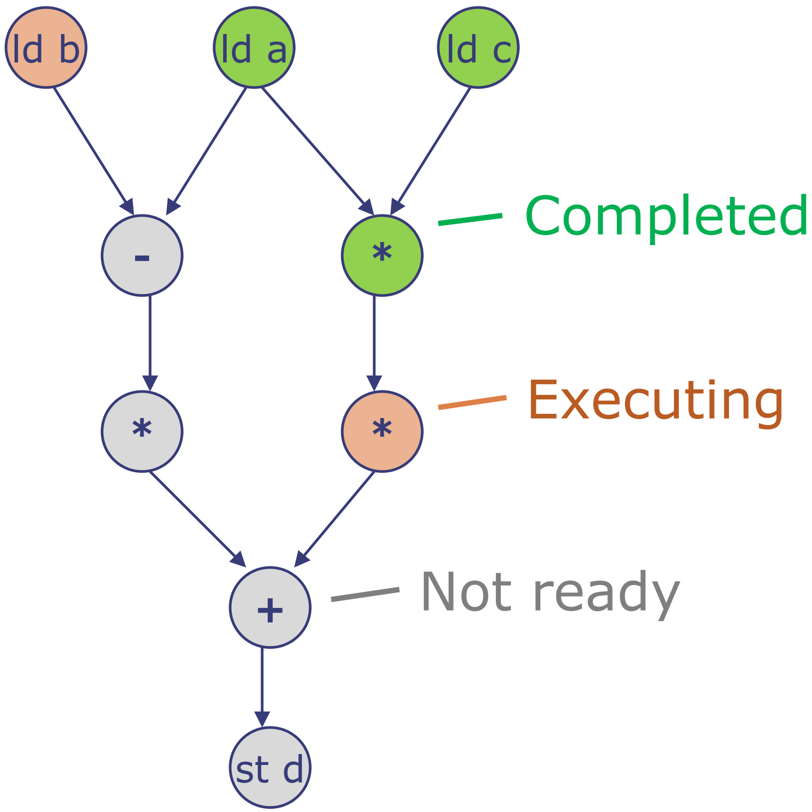
Out-of-Order Execution Example

- If `ld b` takes a few cycles (e.g., cache miss), can execute instructions that do not depend on b

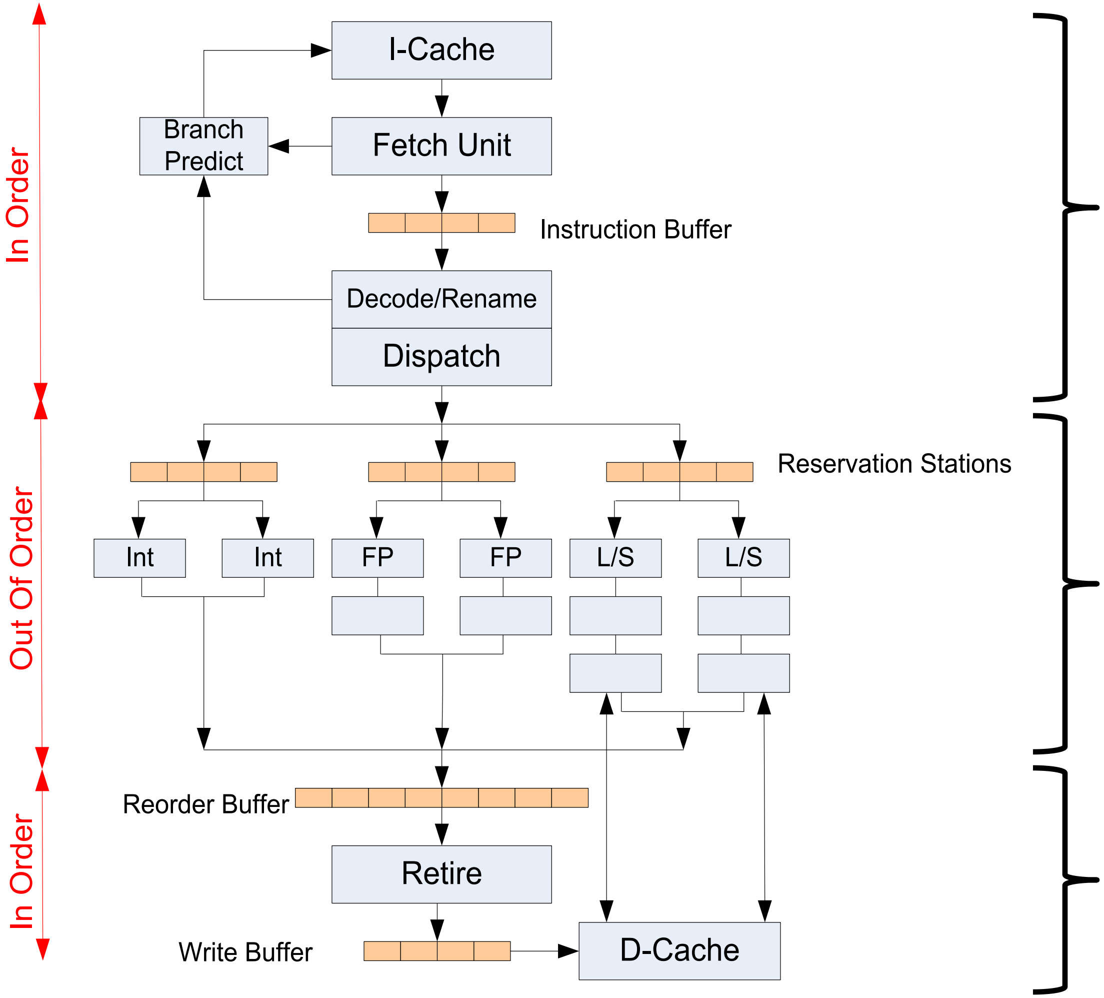
Sequential code

ld a
➡ ld b
sub a-b
mul 3(a-b)
ld c
mul ac
mul 7ac
add 3(a-b)+7ac
st d

Dataflow graph



A Modern Out-of-Order Superscalar Processor



Reconstruct dataflow graph

Execute each instruction as soon as its source operands are available

Write back results in program order

Why is this needed?

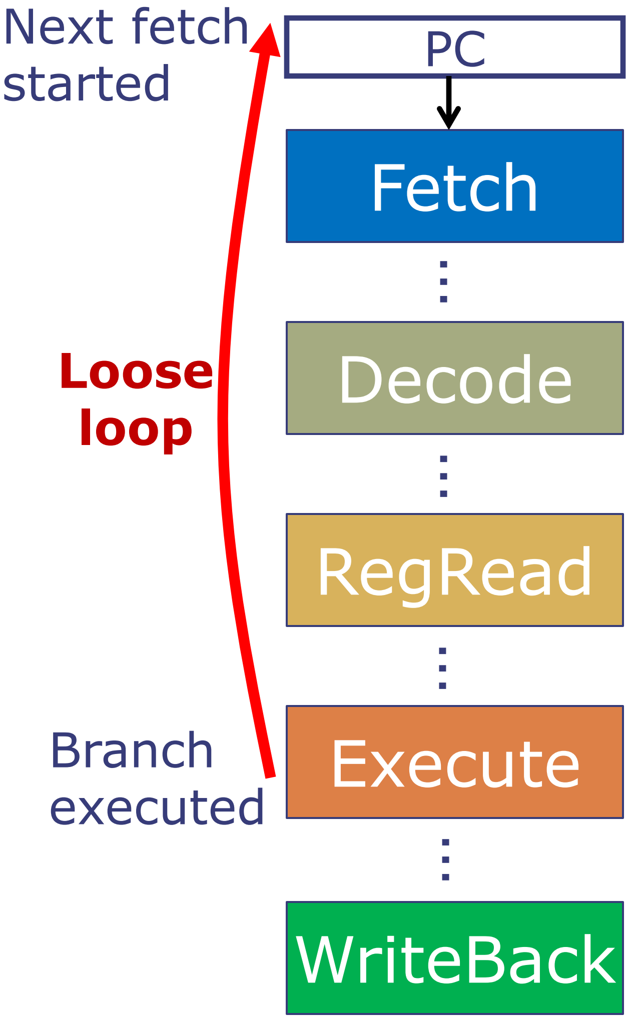
Control Hazard Penalty

- Modern processors have >10 pipeline stages between next PC calculation and branch resolution!

- How much work is lost every time pipeline does not follow correct instruction flow?

  Loop length x Pipeline width

- One branch every 5-20 instructions… performance impact of mispredictions?



RISC-V Branches and Jumps

- Each instruction fetch depends on information from the preceding instruction:
  1) Is the preceding instruction a taken branch or jump?
  2) If so, what is the target address?

Instruction	*Taken known?*	*Target known?*
JAL	After Inst. Decode	After Inst. Decode
JALR	After Inst. Decode	After Inst. Execute
Branches	After Inst. Execute	After Inst. Decode

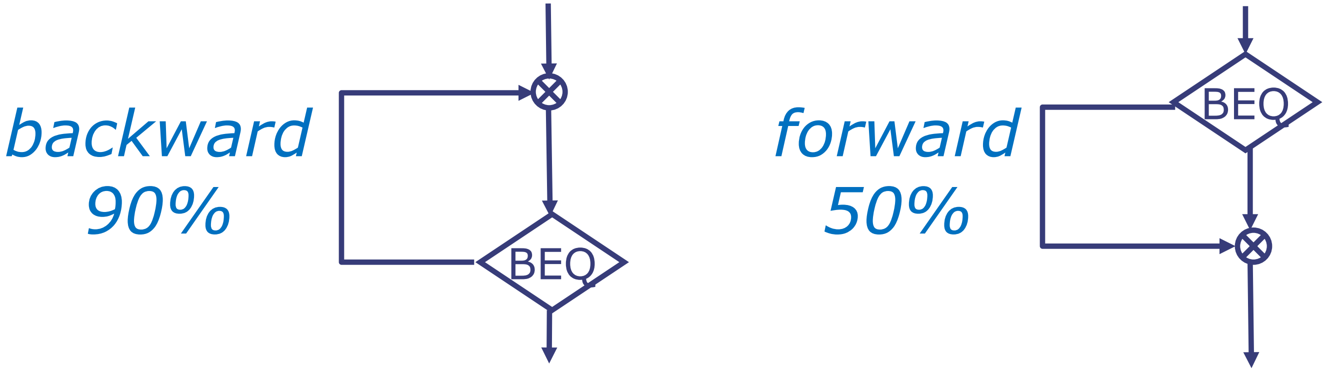
Resolving Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages
- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated
- Strategy 3: Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value
- Strategy 4: Find something else to do

Predict jump/branch target and direction

Static Branch Prediction

- Probability a branch is taken is ~60-70%, but:



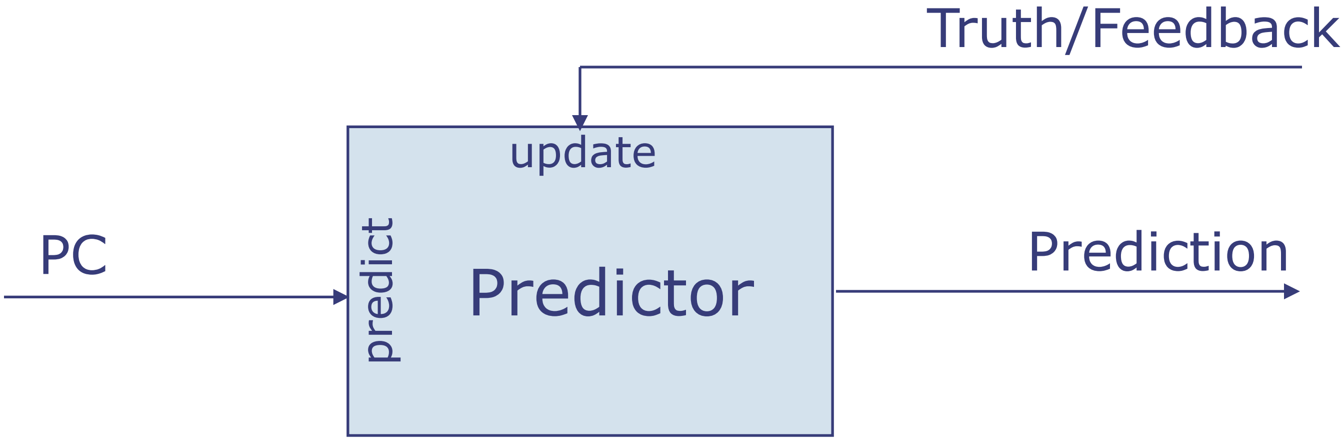
backward
90%

forward
50%

- Some ISAs attach preferred direction hints to branches, e.g., Motorola MC88110
  - bne0 *(preferred taken)* beq0 *(not taken)*
- Achieves ~80% accuracy

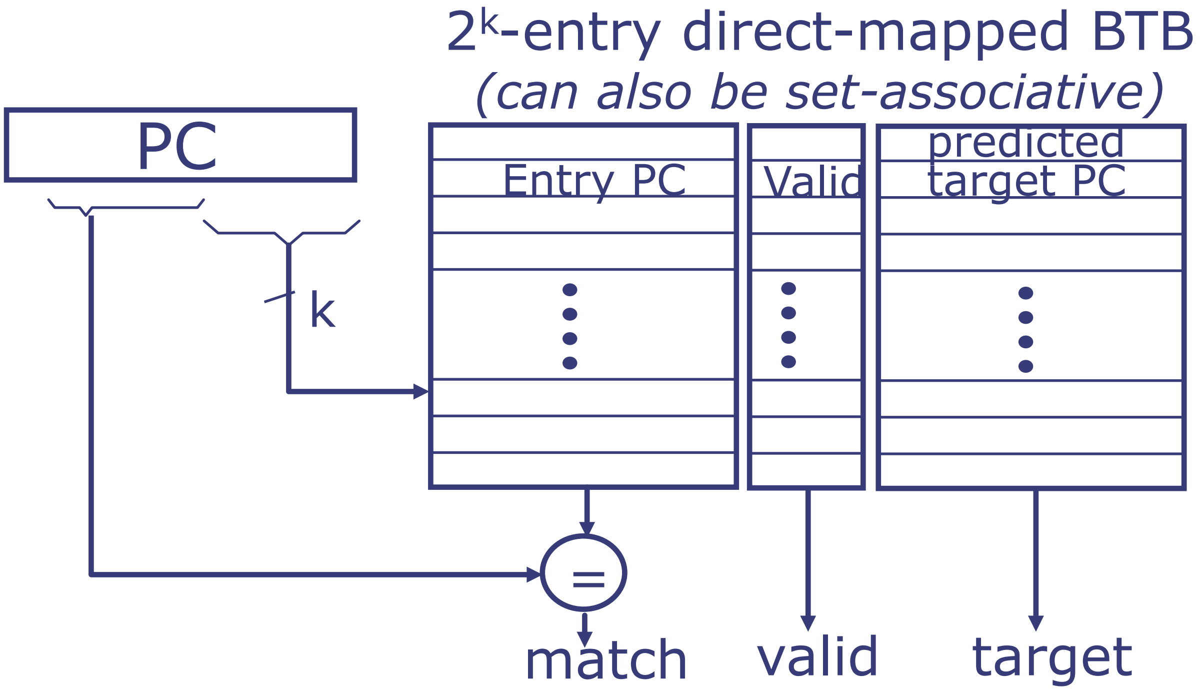
Dynamic Branch Prediction

Learning from past behavior



- Temporal correlation
  - The way a branch resolves may be a good predictor of the way it will resolve at the next execution
- Spatial correlation
  - Several branches may resolve in a highly correlated manner (a preferred path of execution)

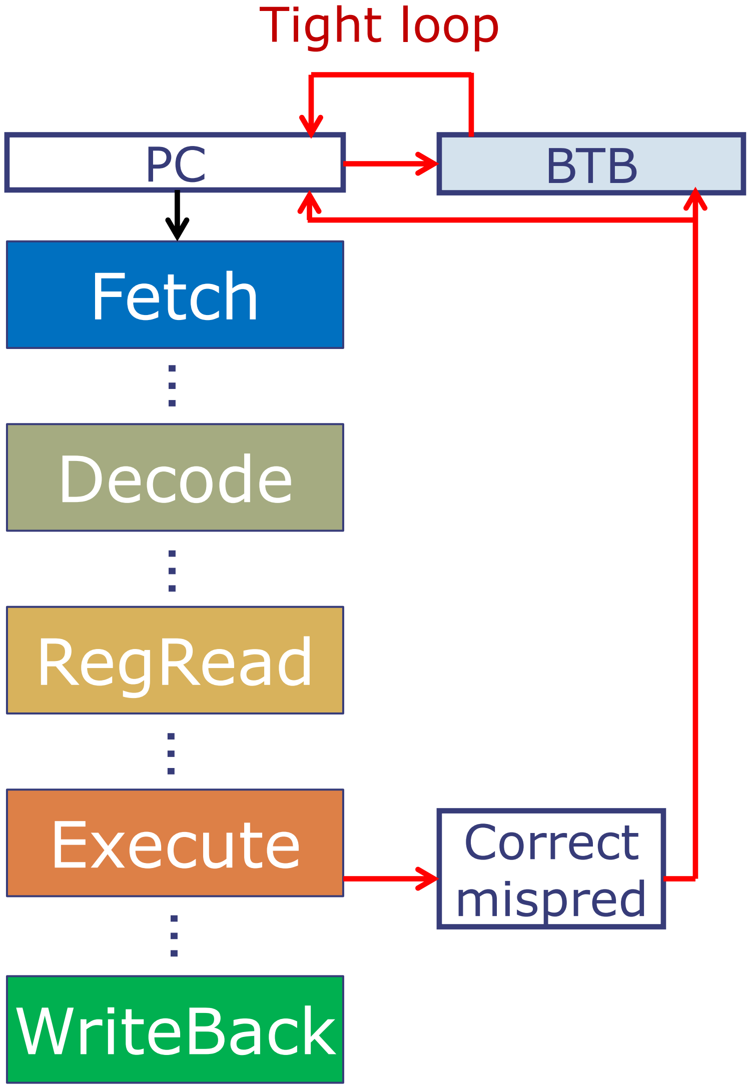
Predicting the Target Address: Branch Target Buffer (BTB)



- BTB is a cache for targets: Remembers last target PC *for taken branches and jumps*
    - If hit, use stored target as predicted next PC
    - If miss, use PC+4 as predicted next PC
    - After target is known, update if prediction is wrong

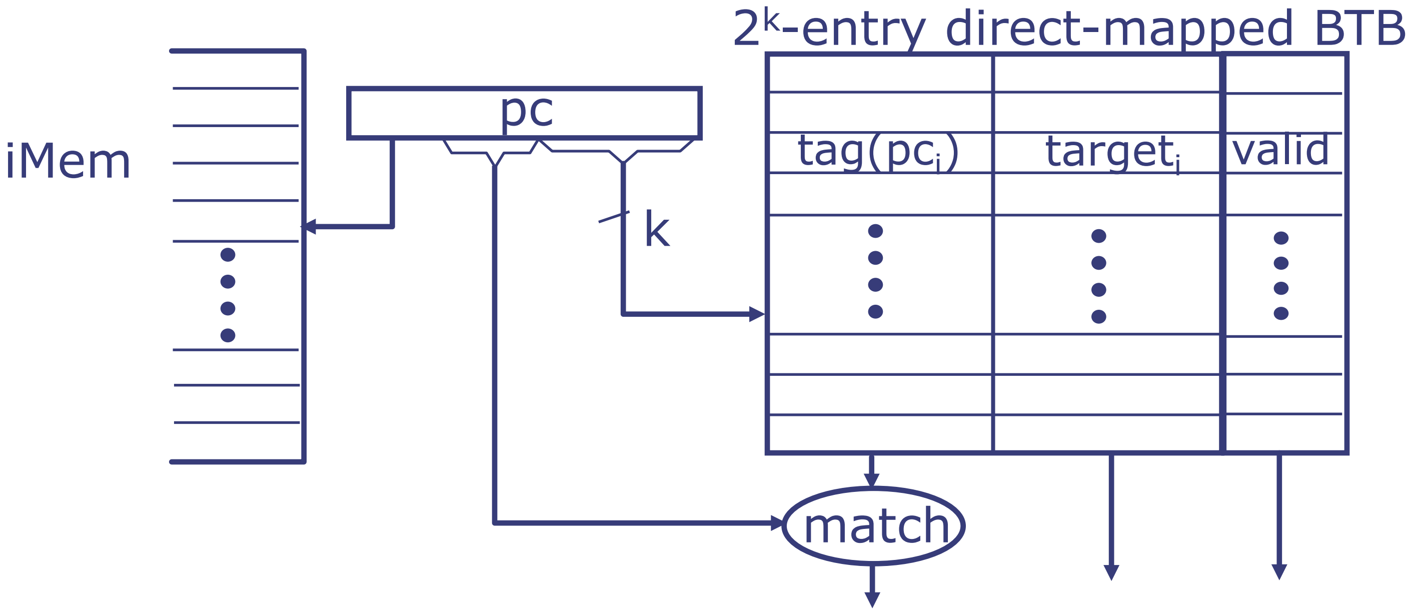
Integrating the BTB in the Pipeline

Predict next PC immediately



Correct misprediction when the right outcome is known

BTB Implementation Details



- Unlike caches, it is fine if the BTB produces an invalid next PC
  - It's just a prediction!
- Therefore, BTB area & delay can be reduced by
  - Making tags arbitrarily small (match with a subset of PC bits)
  - Storing only a subset of target PC bits (fill missing bits from current PC)
  - Not storing valid bits
- Even small BTBs are very effective!

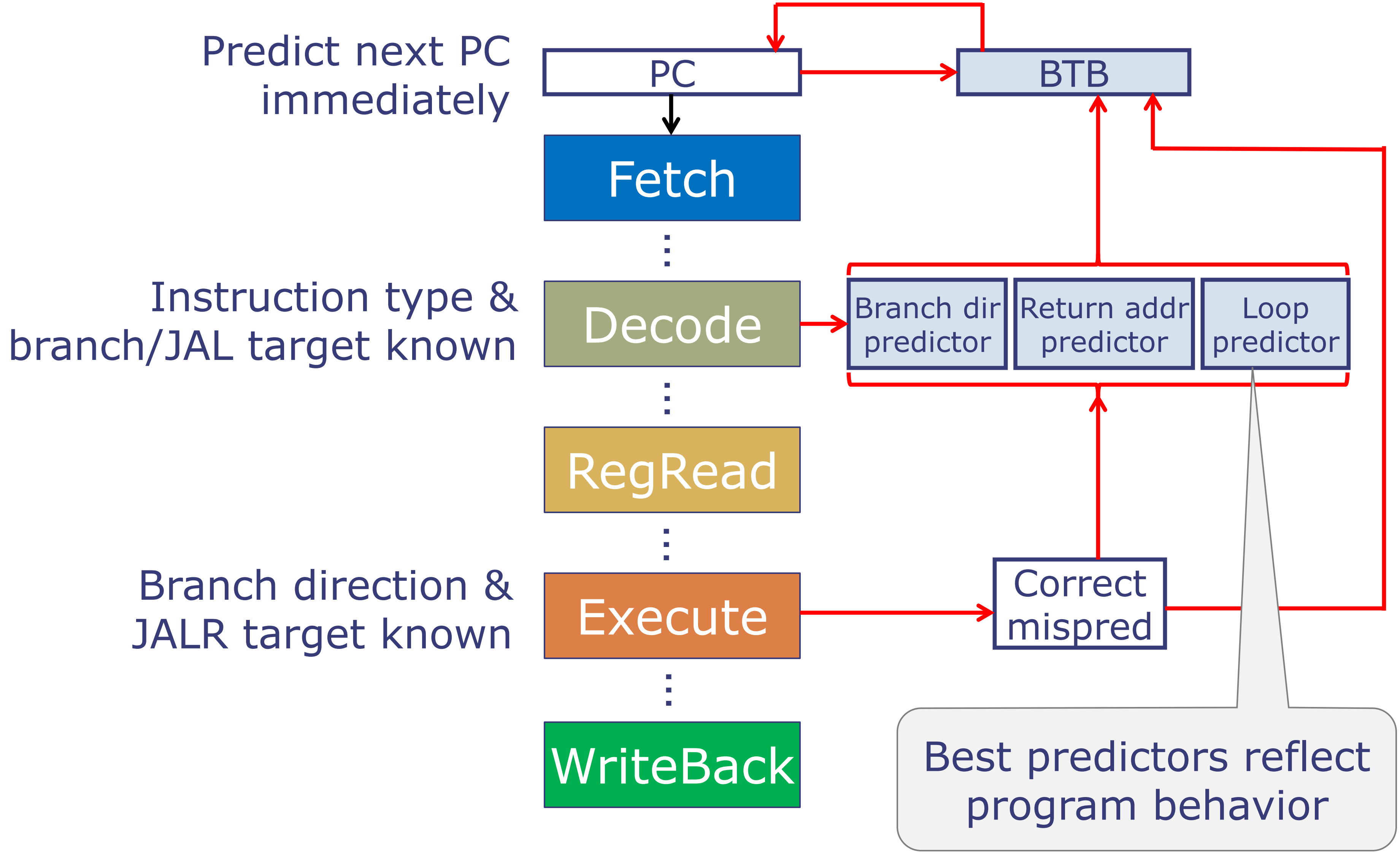
BTB Interface

```
typedef struct
  { Word pc; Word nextPc; Bool taken; } UpdateArgs;
module BTB;
  method Addr predict(Addr pc);
  input Maybe#(UpdateArgs) update default = Invalid;
endmodule
```

- *predict:* Simple lookup to predict nextPC in Fetch stage
- *update:* On a pc misprediction, if the jump or branch at the pc was taken, then the BTB is updated with the new (pc, nextPC). Otherwise, the pc entry is deleted.

A BTB can improve CPI
in the design project
(and has lower t_{CLK} than static prediction)

Modern Processors Combine Multiple Specialized Predictors



Best predictors reflect program behavior

Improving performance by changing the ISA

$$\frac{\text{Time}}{\text{Program}} = \boxed{\frac{\text{Instructions}}{\text{Program}}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}$$

- Doing more work per instruction is a very effective way to improve performance for some applications
  - Complex operations (integer multiplication and division, floating-point arithmetic, encryption…)
  - Multiple operations per instruction (vector/SIMD, matrix multiplication)

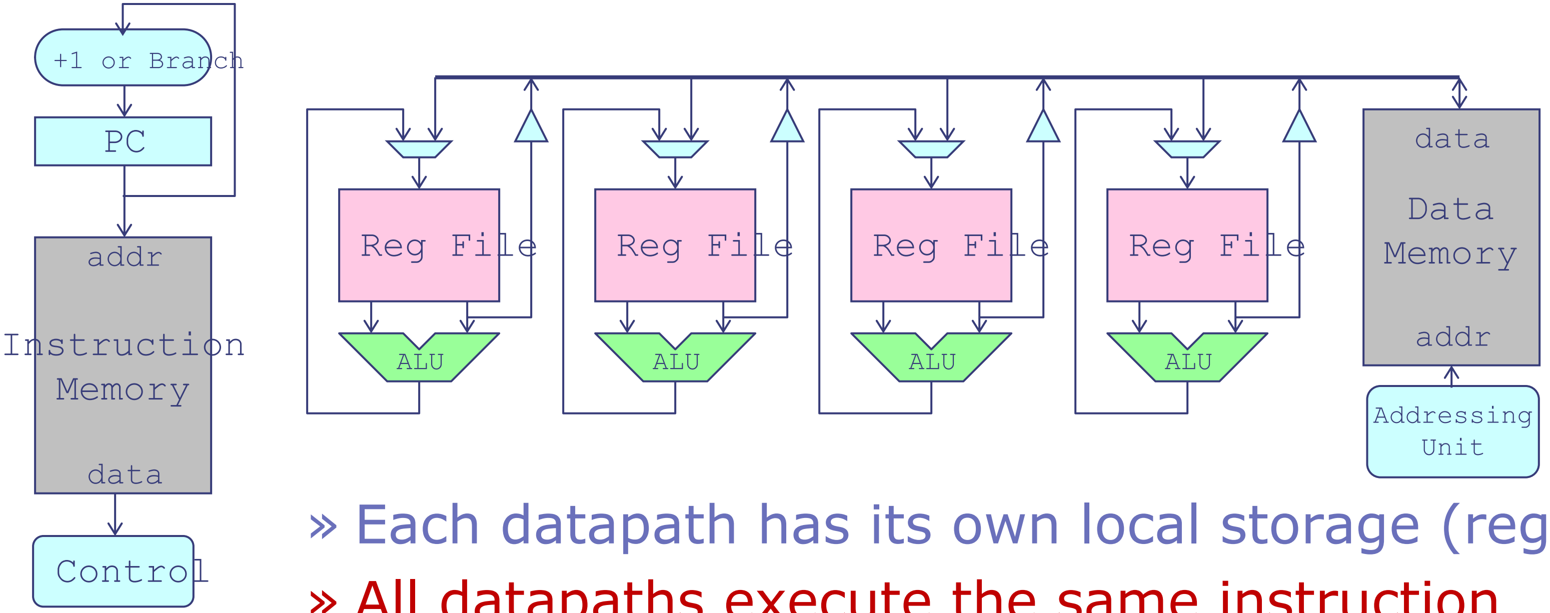
Main avenue to improve performance
on the design project

Vector instructions

- Same operation applied to multiple data elements

  `for (int i = 0; i < 16; i++) x[i] = a*b[i] + c[i];`

- Exploit with vector processors or ISA extensions



» Each datapath has its own local storage (reg file)

» All datapaths execute the same instruction

» Memory access with vector loads and stores + wide memory port

Vector Code Example

```
for (i = 0; i < 16; i++)  x[i] = a[i] + b[i];
```

RISC-V assembly

```
loop: lw a1, 0(a4)
      lw a2, 0(a5)
      add a3, a1, a2
      sw a3, 0(a6)
      addi a4, a4, 4
      addi a5, a5, 4
      addi a6, a6, 4
      blt a6, a7, loop
```

8*16 = 128 instructions

Equivalent vector assembly

```
ld.v v1, 0(a4)
ld.v v2, 0(a5)
add.v v3, v1, v2
st.v v3, 0(a6)
```

4 instructions

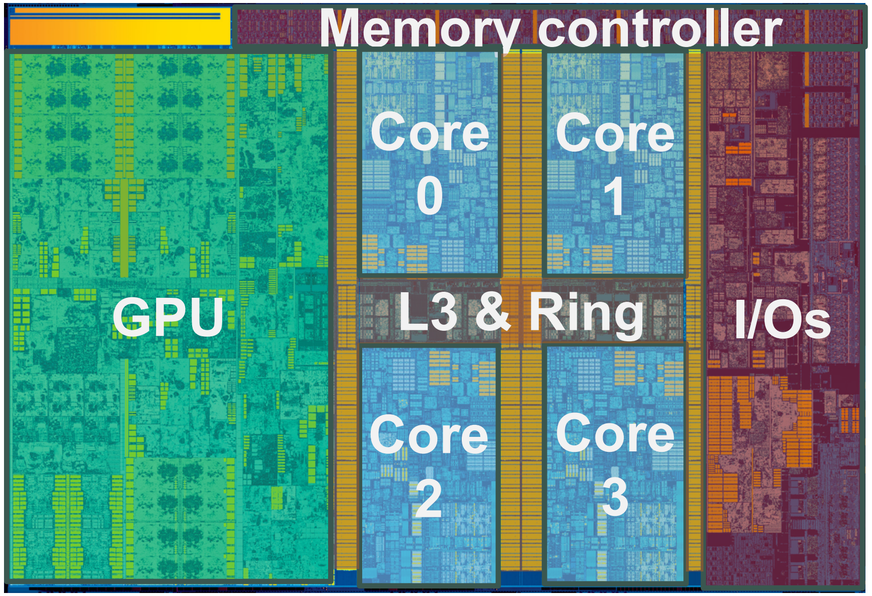
Vector Processing Implementations

- Advantages of vector ISAs:
  - Compact: 1 instruction defines N operations
  - Parallel: N operations are (data) parallel and independent
  - Expressive: Memory operations describe regular patterns

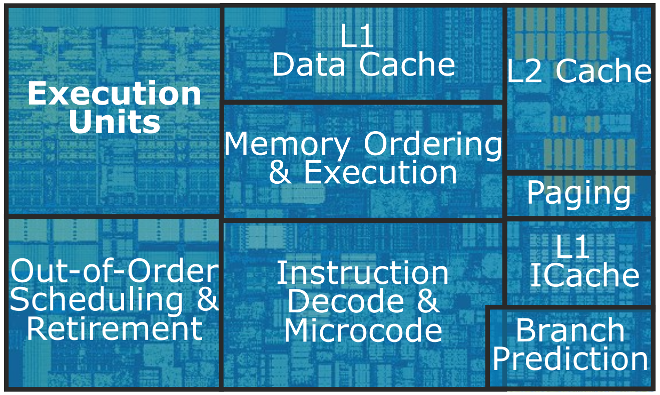
- Modern CPUs: Vector extensions & wider registers
  - SSE (1999): 128-bit operands (4x32-bit or 2x64-bit)
  - AVX (2011): 256-bit operands (8x32-bit or 4x64-bit)
  - AVX-512 (2017): 512-bit operands
  - Explicit parallelism, extracted at compile time (vectorization)

Putting It All Together: Intel Core i7 (Skylake)

- Each core has 19 pipeline stages, ~4GHz
- 6-wide superscalar
- Out of order execution
- Vector ISA extension with 512-bit vector registers and operations (AVX-512)
- Multi-level branch predictors
- Caches:
  - L1: 32KB I + 32KB D
  - L2: 256KB
  - L3: 8MB, shared
- Large overheads vs simple cores!



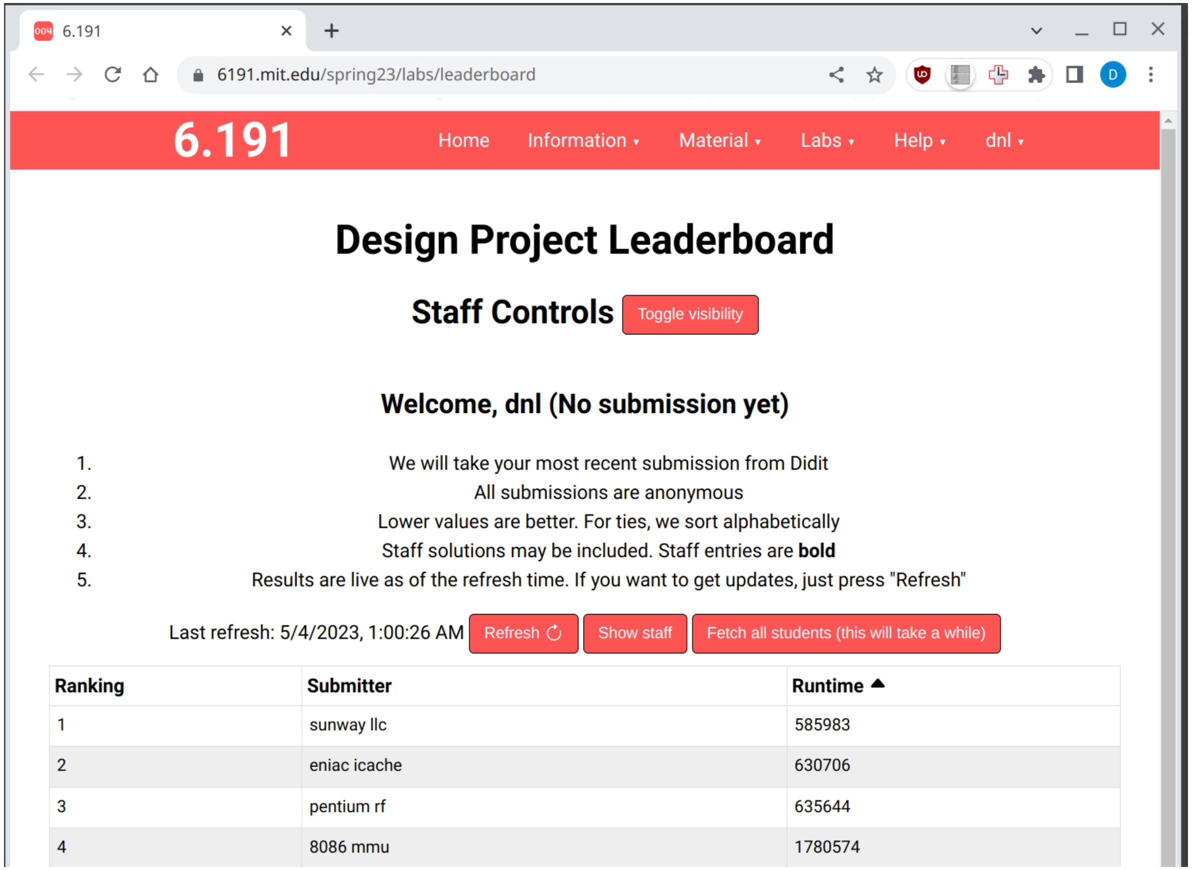
Intel, 2016, 14nm, 1.7B transistors, 122mm^2


Execution Units
L1 Data Cache
L2 Cache
Memory Ordering & Execution
Paging
Out-of-Order Scheduling & Retirement
Instruction Decode & Microcode
L1 ICache
Branch Prediction

▫ Your RISC-V core

Design Project Leaderboard

- Available in Labs > DP > Leaderboard



Design Project Leaderboard

Staff Controls

Welcome, dnl (No submission yet)

1. We will take your most recent submission from Didit
2. All submissions are anonymous
3. Lower values are better. For ties, we sort alphabetically
4. Staff solutions may be included. Staff entries are **bold**
5. Results are live as of the refresh time. If you want to get updates, just press "Refresh"

Last refresh: 5/4/2023, 1:00:26 AM

Ranking	Submitter	Runtime ▲
1	sunway llc	585983
2	eniac icache	630706
3	pentium rf	635644
4	8086 mmu	1780574

Thank you!

Good luck on Quiz 3 ☺

And thanks for taking the class!