Common Hardware Architecture

Outline

- Program & Instruction Set
- CPU Architecture
- Vector Instruction
- GPU & Accelerators

Program & Instruction Set

Levels of Program Code

- High-level language (e.g., C, C++ & Python)
- Assembly language
  - instruction
- Binary code
  - representation of instruction in zeros & ones

```
swap(int v[], int k) {
    int tmp = v[k];
    v[k] = v[k+1];
    v[k+1] = tmp;
}
```

Compiler

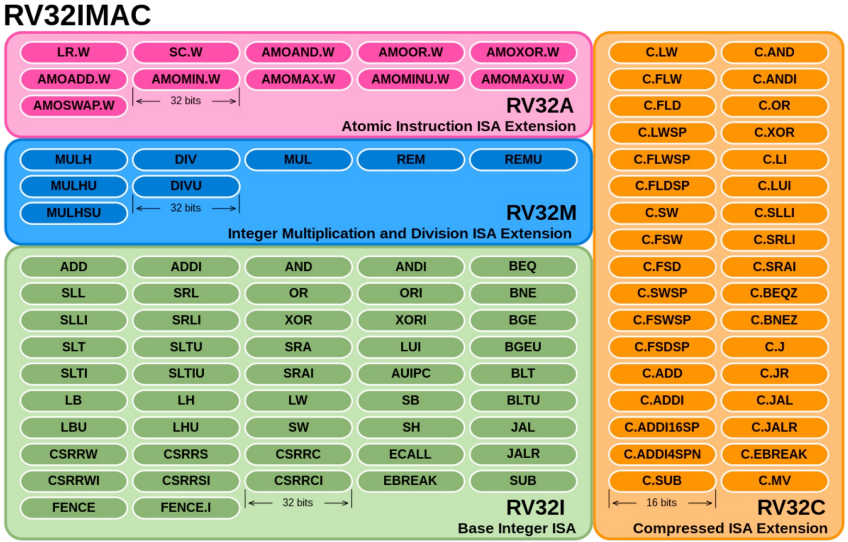
```
swap:
    slli x6, x11, 3
    add  x6, x10, x6
    ld   x5, 0(x6)
    ld   x7, 8(x6)
    sd   x7, 0(x6)
    sd   x5, 8(x6)
    jalr x0, 0(x1)
```

Assembler

```
01010010010010111111000101010
00001010101010100101011010 10
10101001101111111000000000010
```

Instruction Set Architecture (ISA)

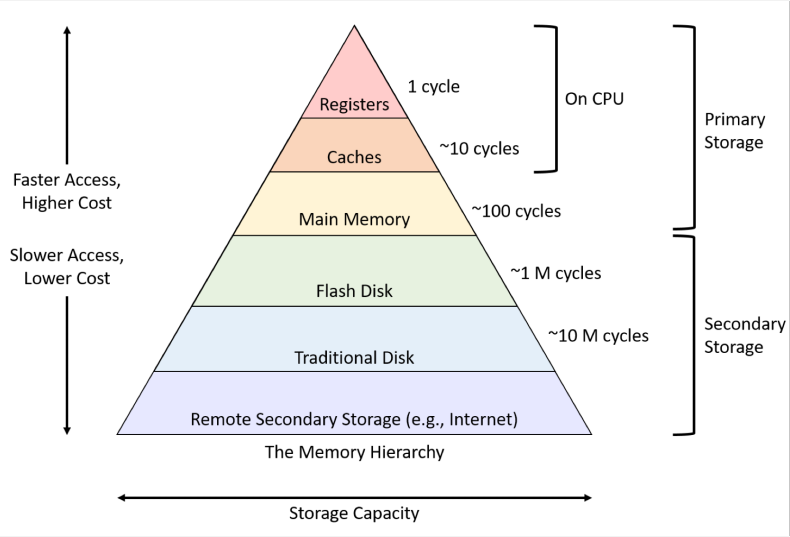
- Intel & AMD:
  x86
- Apple & Android phones:
  arm
- RISC-V
- Nvidia GPU:
  PTX (Parallel Thread eXecution)

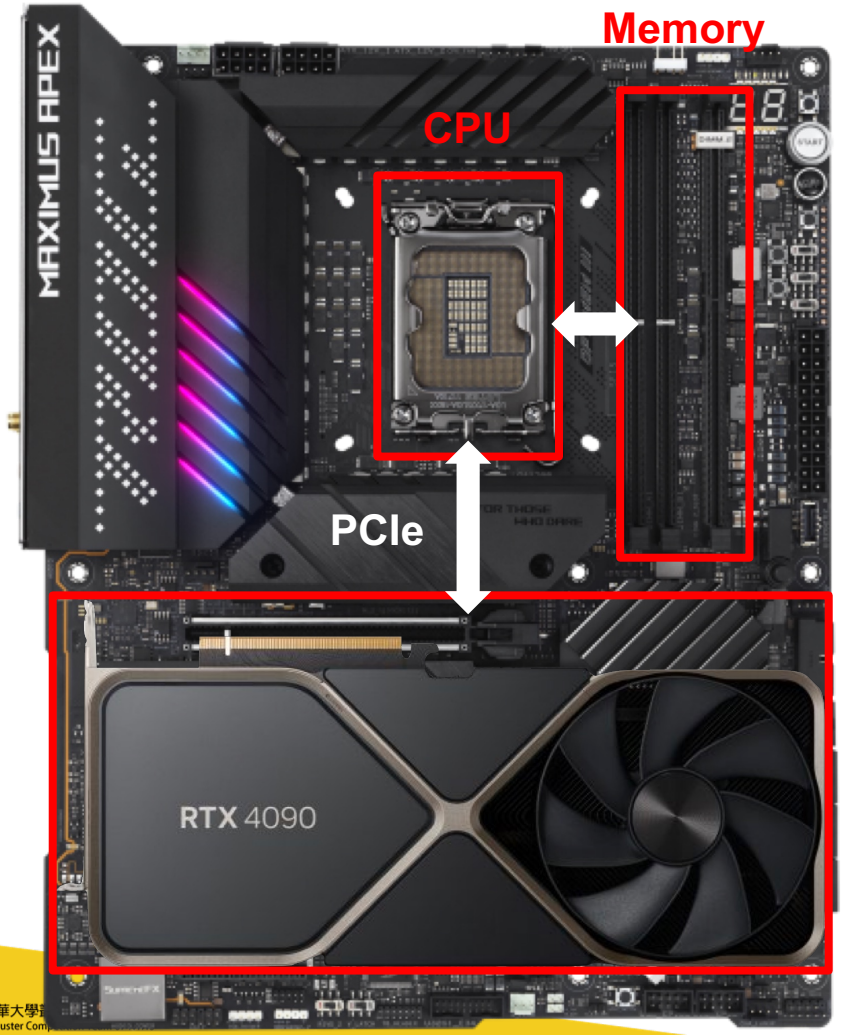


RISC-V instruction set

Memory & Disk

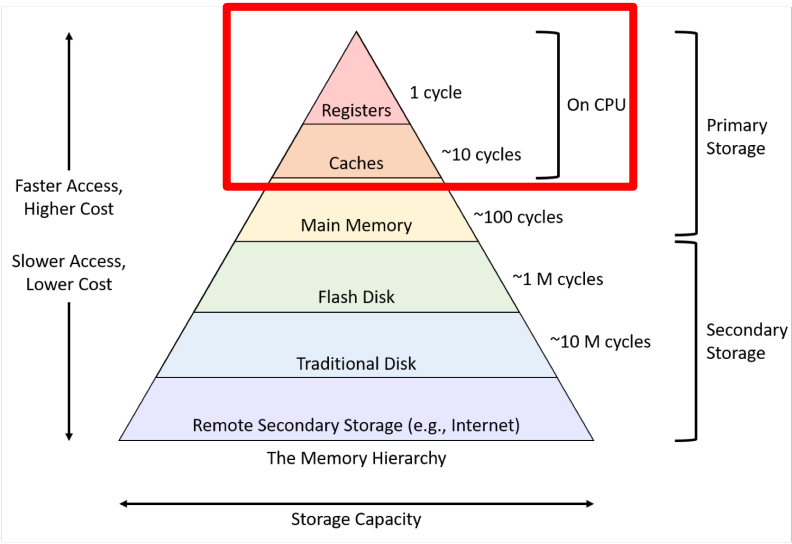
- CPU can only access registers
- Data should be copied from memory to register for computing
- Most of our program is stored in secondary storage, and loaded into main memory during execution
- Cache is designed to reduce number of accesses of main memory




Memory
CPU
PCIe
MAXIMUS APEX
RTX 4090

Memory & Disk

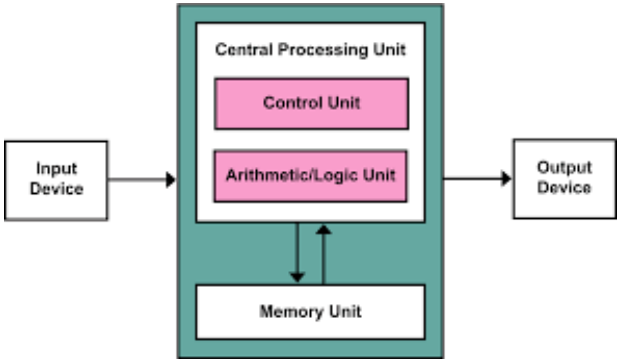
- CPU can only access registers
- Data should be copied from memory to register for computing
- Most of our program is stored in secondary storage, and loaded into main memory during execution
- Cache is designed to reduce number of accesses of main memory



CPU Architecture

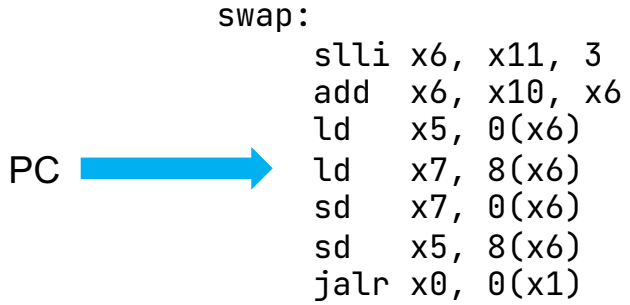
Von-Neumann Architecture

- A processing unit with both an arithmetic logic unit (ALU) and registers
- A control unit that includes an instruction register and a program counter
- Memory that stores data and instructions
- External mass storage
- Input and output mechanisms



Program Counter (PC)

- The counter that points to the instruction to be executed

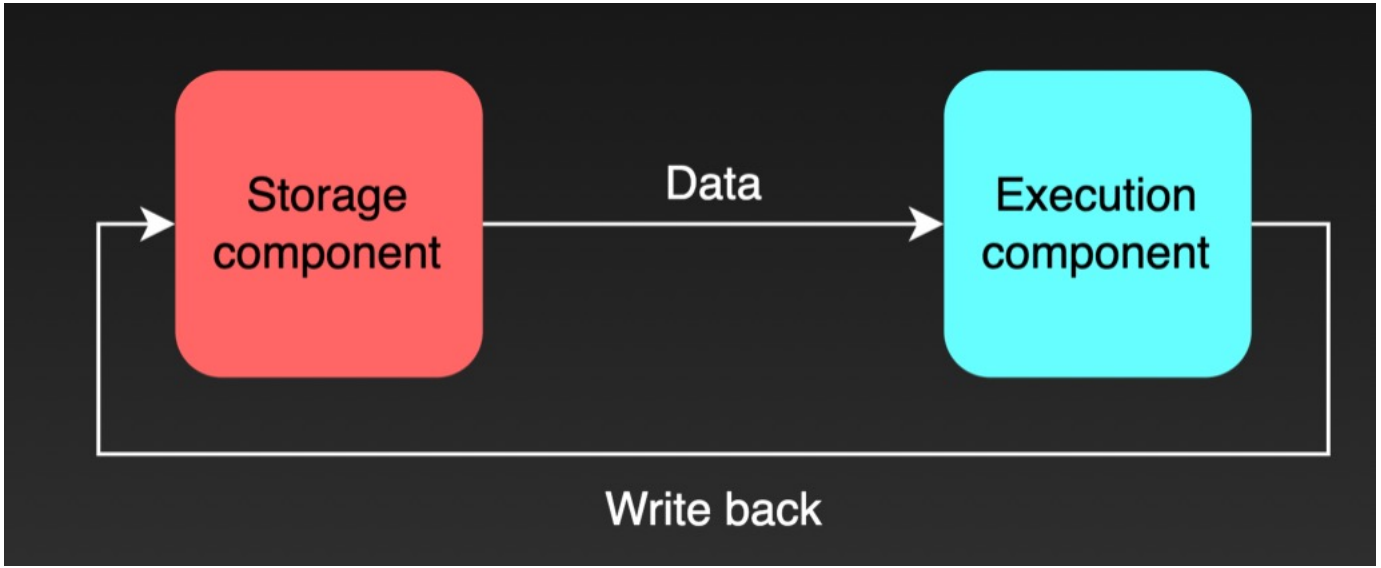


A Simple Compute Flow

1. PC + 1:
   Program counter points to the next instruction
2. Fetch instruction from memory
3. CPU decodes the instruction
4. The registers are read out by the instruction
5. ALU performs calculation
   (e.g., ADD, MUL, SUB)
6. Write back data to register / memory

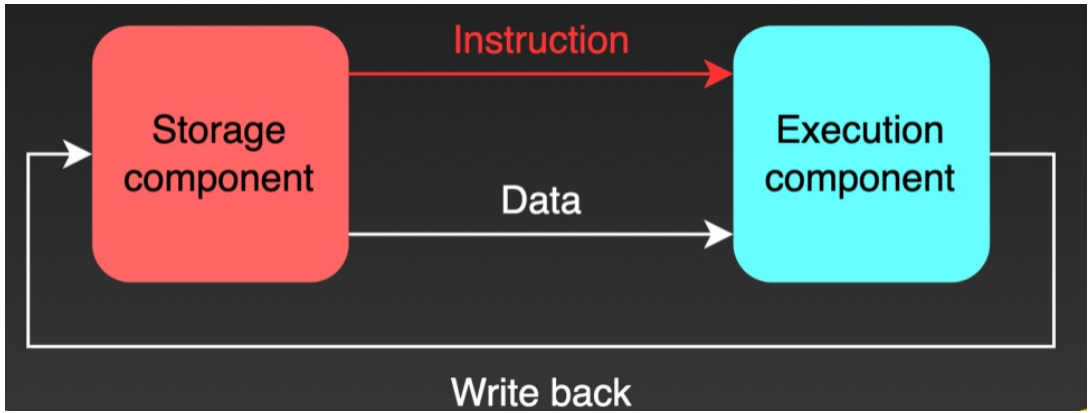

```
swap:
        slli x6, x11, 3
        add  x6, x10, x6
        ld   x5, 0(x6)
        ld   x7, 8(x6)
        sd   x7, 0(x6)
        sd   x5, 8(x6)
        jalr x0, 0(x1)
```

Building a CPU - Simple Datapath Structure



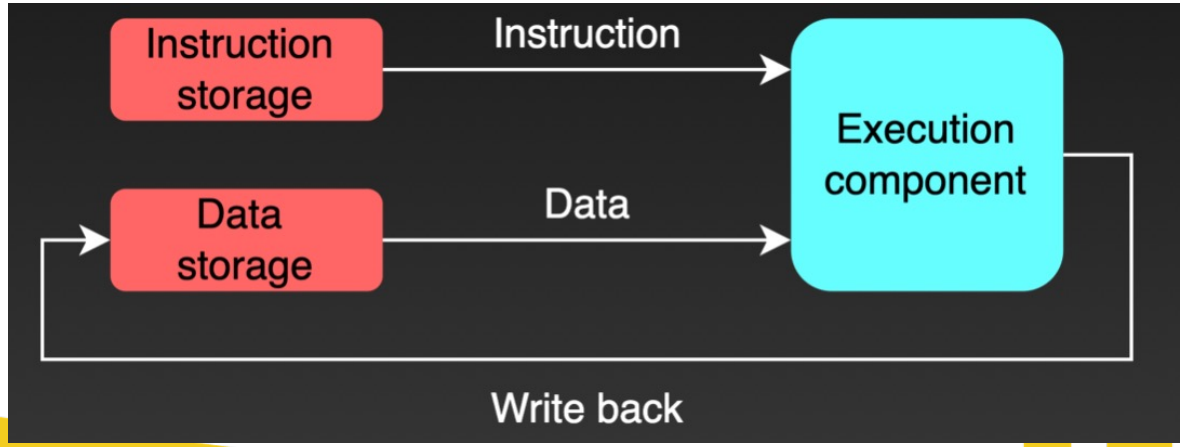
Building a CPU - Instruction Datapath

- The execution component requires the following information:
- Instruction (e.g., ADD)
- Data to be calculated (e.g., 8+9)



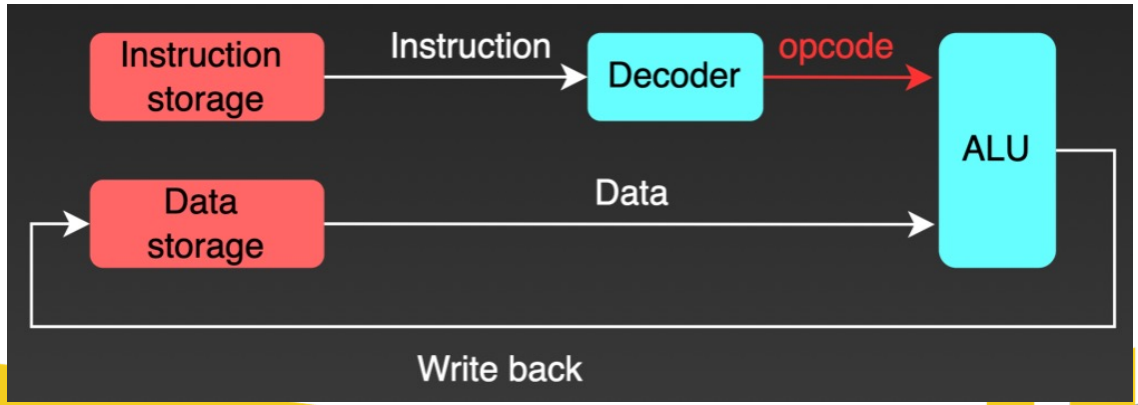
Building a CPU - Instruction & Data Storage

- It may be difficult to design a storage which can provide 2 different data at the same time
- For simplicity, we split it into instruction storage & data storage



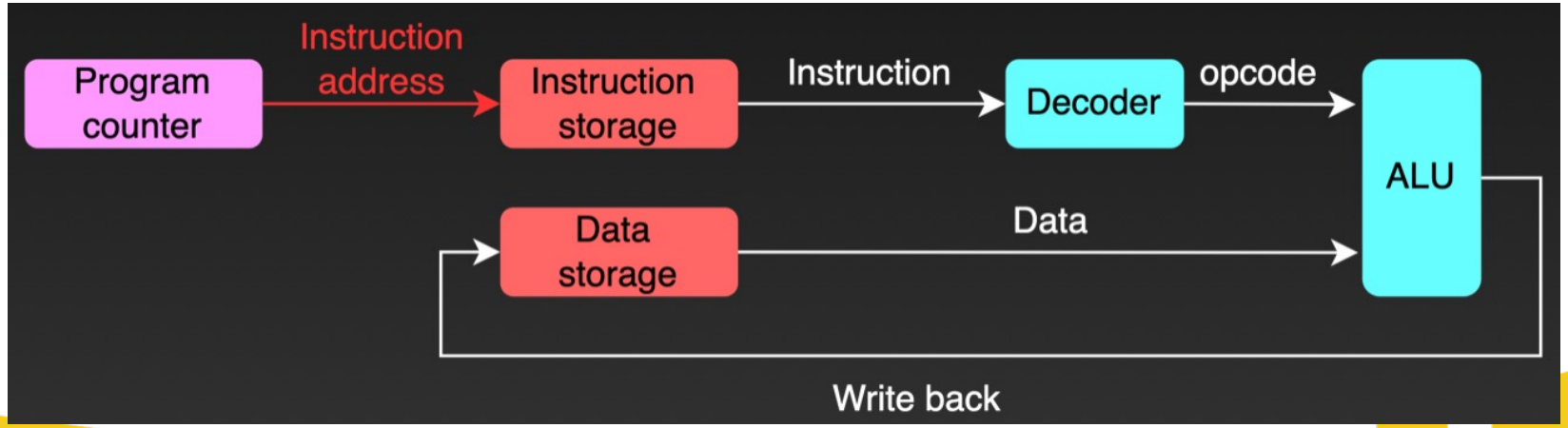
Building a CPU – Decoder & ALU

- To make designing the execution component easier, we split it into decoder and ALU (arithmetic logic unit)
- Decoder: figure out which operation should be done with the provided instruction
- ALU: do only the calculation (e.g., add, multiply, compare...)



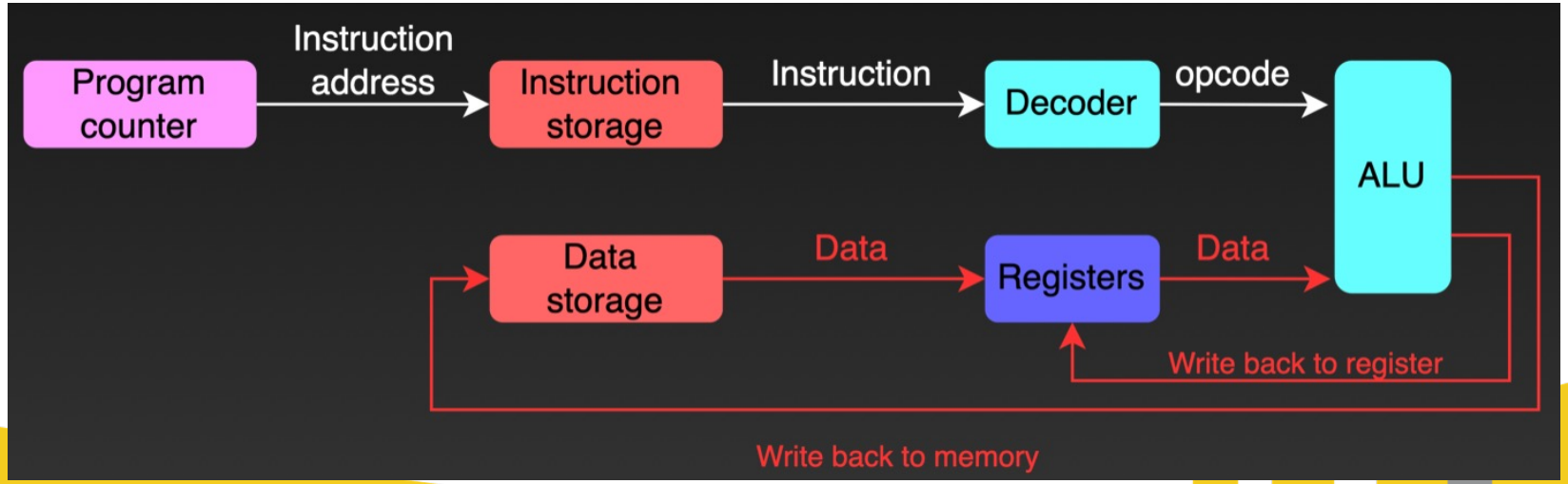
Building a CPU – Program Counter

- We need the address of the current instruction so that we can fetch it from the instruction storage



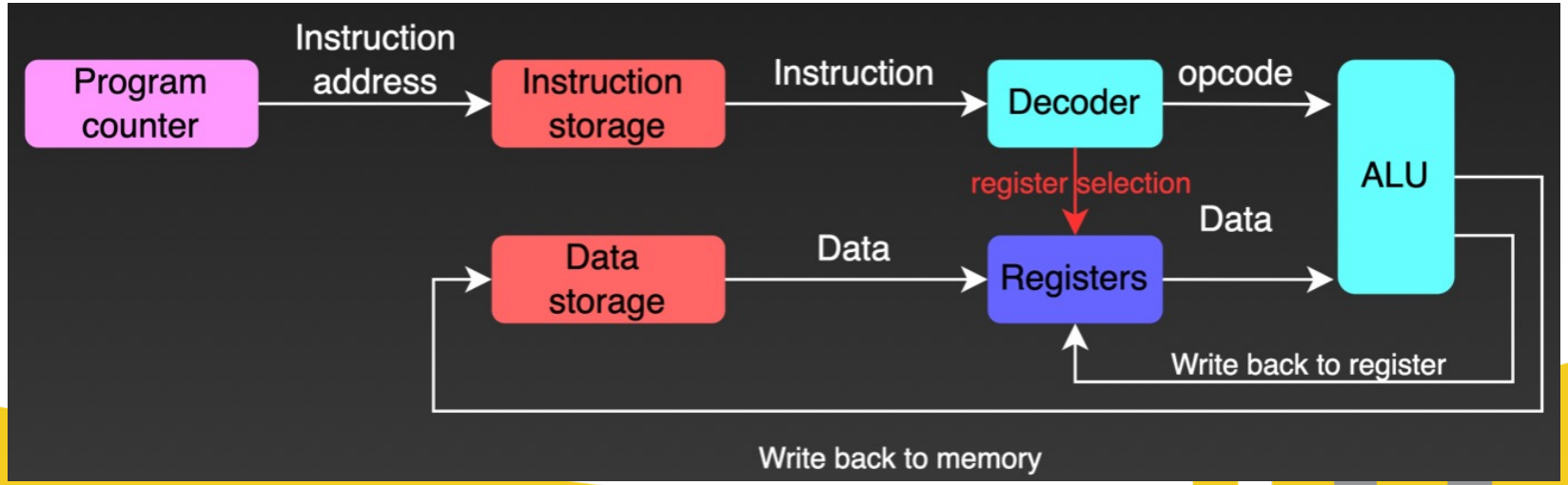
Building a CPU – Registers

- The data must be loaded into register first

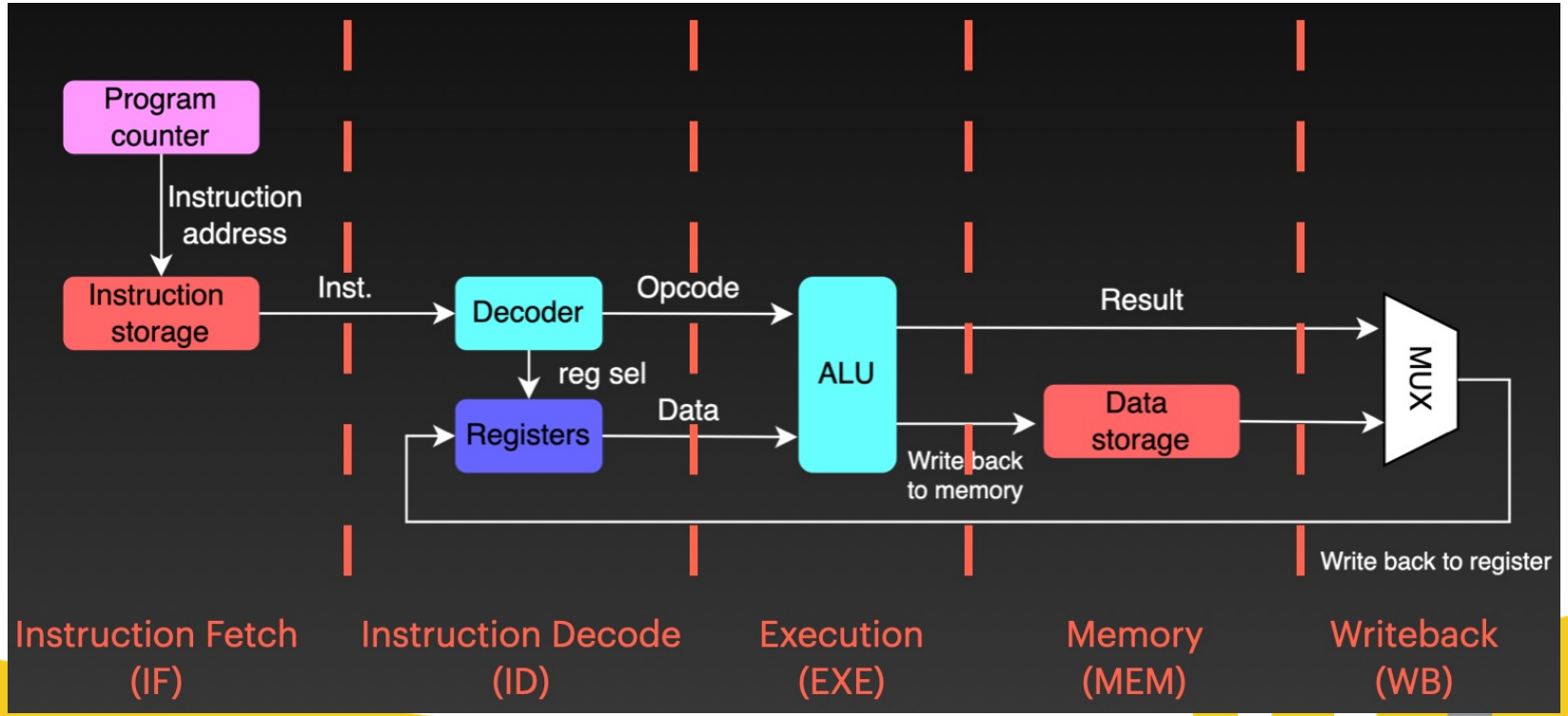


Building a CPU – Register Selection

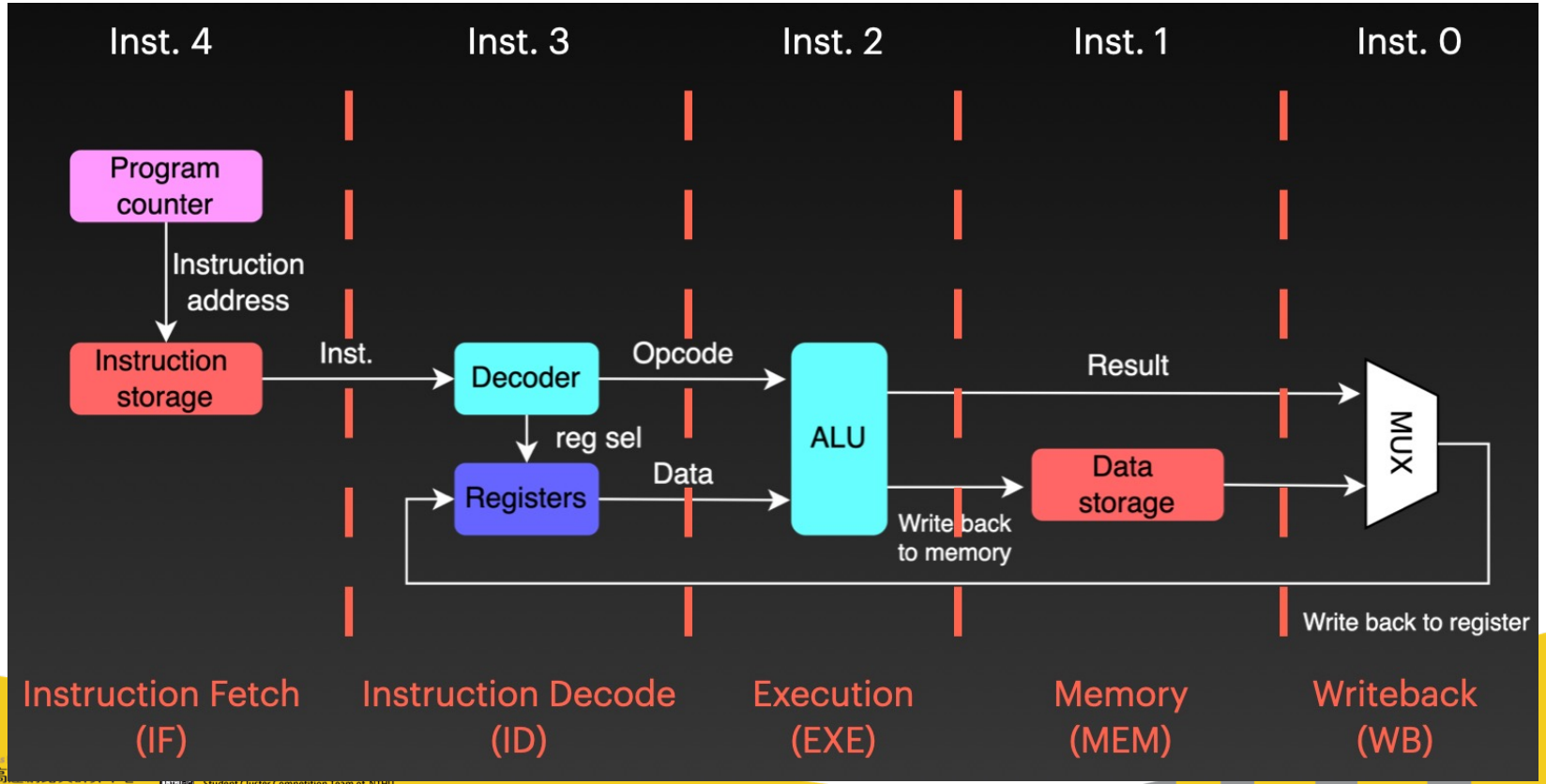
- We need to know which register is selected to be read
- This information is encoded in the instruction:
  ADD x1, x2, x3 (x1 = x2+x3)



Building a CPU – A Simple 5-Stage Dataflow



Pipelining


Inst. 4
Inst. 3
Inst. 2
Inst. 1
Inst. 0
Program counter
Instruction address
Instruction storage
Inst.
Decoder
Opcode
reg sel
Registers
Data
ALU
Result
Data storage
Write back to memory
MUX
Write back to register
Instruction Fetch (IF)
Instruction Decode (ID)
Execution (EXE)
Memory (MEM)
Writeback (WB)

Branch Condition

- Multiple instructions are executed (at different stage) in a pipelined CPU
- In this example, inst. 3 and 4 will only be executed if the branch statement if FALSE.

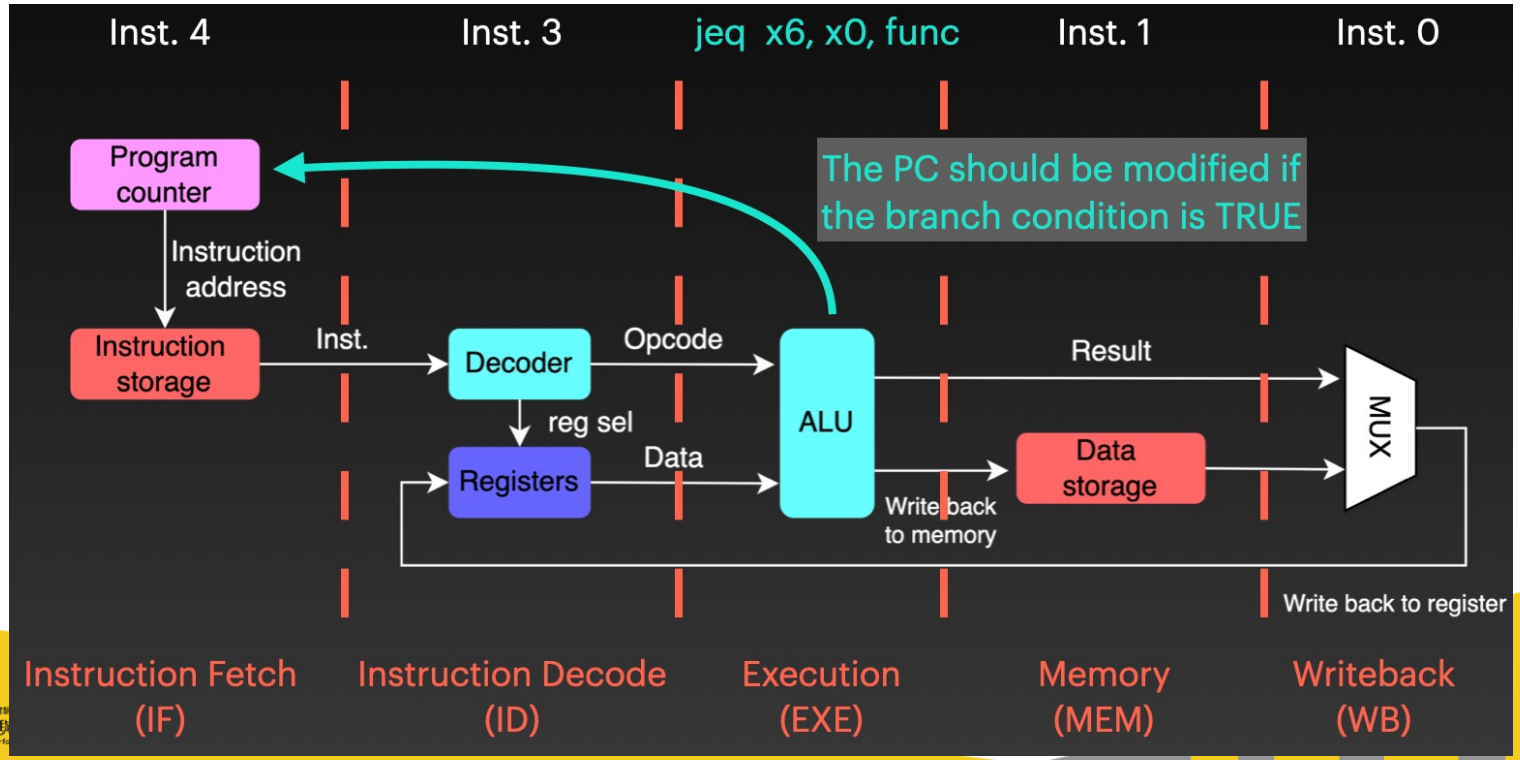
Branch instruction
(e.g., `if-else`)

```
func:
    inst. 0
    inst. 1
    jeq  x6, x0, func
    inst. 3
    inst. 4
```

Branch Condition

Branch instruction
(e.g., `if-else`)

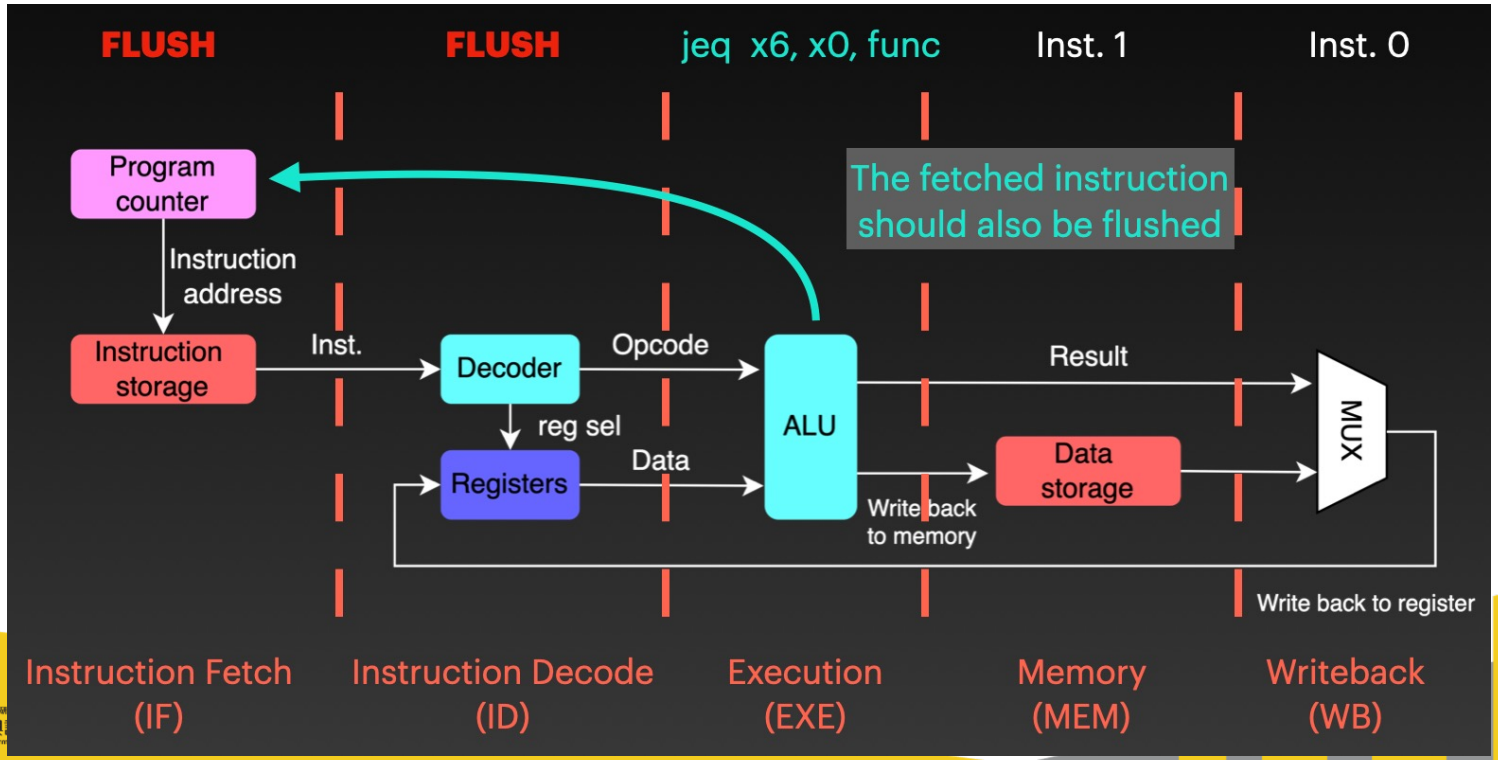
```
func:
    inst. 0
    inst. 1
    jeq  x6, x0, func
    inst. 3
    inst. 4
```



Branch Condition

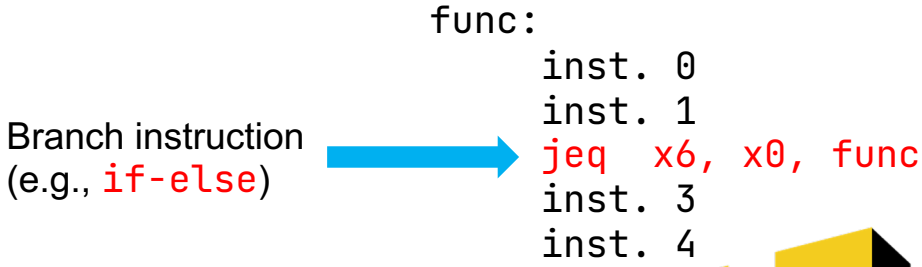
Branch instruction
(e.g., `if-else`)

```
func:
    inst. 0
    inst. 1
    jeq  x6, x0, func
    inst. 3
    inst. 4
```



Branch Prediction

- FLUSHing instruction results in wasted computational time
- In the previous example, we assume (predict) the branch is "NOT taken"
- The accuracy of branch predictor affects the CPU performance
- Commercial CPUs have a better and complicated branch predictor

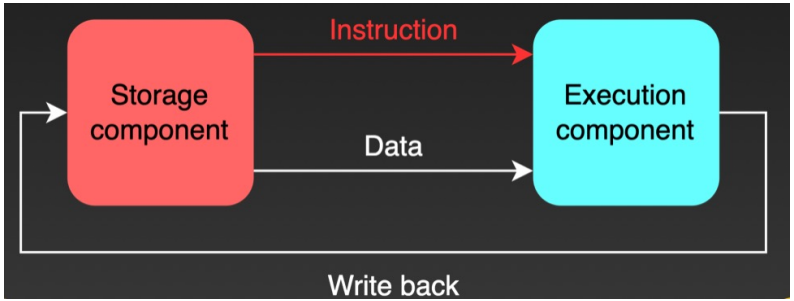


Vector Instruction

Review: A Simple CPU

- We define an instruction for a single calculation
  - E.g., C = A + B

```
swap:
      slli x6, x11, 3
      add  x6, x10, x6
      ld   x5, 0(x6)
      ld   x7, 8(x6)
      sd   x7, 0(x6)
      sd   x5, 8(x6)
      jalr x0, 0(x1)
```



Vector Operations

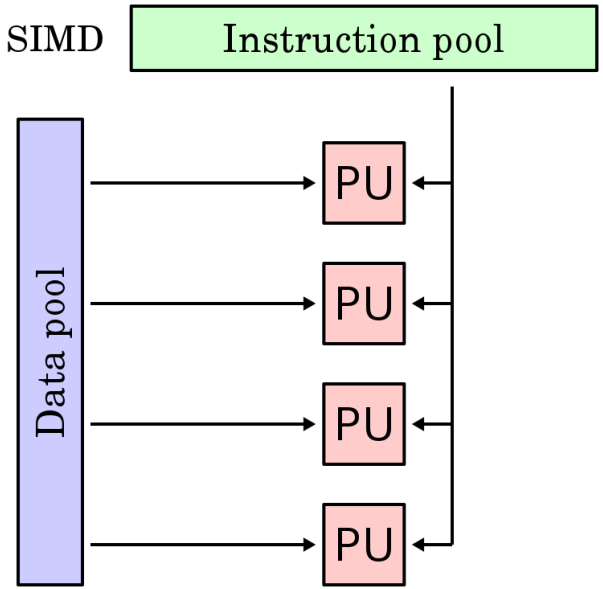
- Perform calculations on a series of number:

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

- It requires at least N multiplication instructions
- However, the operations are the same. Is it possible to perform a single calculation on multiple number?

Single Instruction Multiple Data (SIMD)

- Processes multiple data in a single instruction
- Streaming SIMD Extensions (SSE) provides 128-bit registers
  - Two-operand form: a ← a + b
  - 4 32-bit integers
  - 2 64-bit double-precision FP
  - 4 32-bit single-precision FP
- AVX512: 512 bits



Hardware Support

- Linux command: `lscpu | grep -i $instruction_set`
  - where $instruction_set could be:
  - mmx, sse, sse2, sse3, ssse3, sse4_1, sse4_2, avx, avx2, avx512
- Most CPUs released after 2011 support AVX instructions

```
fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
 syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmpe
rf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movb
e popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l3 cdp_l3 invpcid_s
ingle intel_ppin ssbd mba ibrs ibpb stibp ibrs_enhanced tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjus
t bmi1 avx2 smep bmi2 erms invpcid cqm mpx rdt_a avx512f avx512dq rdseed adx smap clflushopt clwb intel_pt avx512cd avx
512bw avx512vl xsaveopt xsavec xgetbv1 xsaves cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts
 pku ospke avx512_vnni md_clear flush_l1d arch_capabilities
```

Automatic Vectorization

- GCC
  - Vectorization is enabled by the flag `-ftree-vectorize`
  - Enabled by default with flag `-O3`
- Add flag `-march=native` to use instructions supported by the local CPU
- Add compiler flag `-fopt-info-vec-all` to see vectorization log
- Add `#pragma GCC ivdep` to code to declare that there is no data dependency in the following loop

Automatic Vectorization

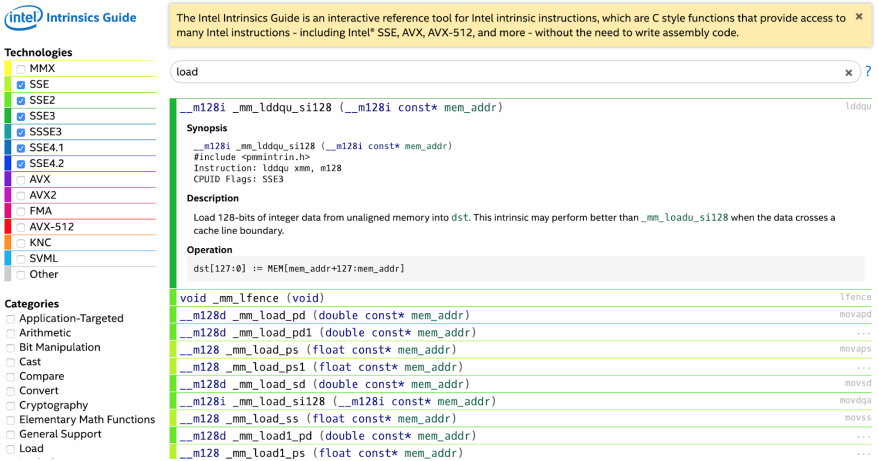
16 bytes = 128 bits

```
mpicc -O3 -march=native -fopt-info-vec-all array.c -o array
array.c:8:3: optimized: loop vectorized using 16 byte vectors
array.c:8:3: optimized:  loop versioned for vectorization because of possible aliasing
array.c:7:6: note: vectorized 1 loops in function
array.c:32:3: optimized: loop vectorized using 16 byte vectors
array.c:32:3: optimized:  loop versioned for vectorization because of possible aliasing
array.c:16:3: missed: couldn't vectorize loop
/usr/lib/gcc/x86_64-pc-linux-gnu/9.1.0/include/emmintrin.h:703:10: missed: not vectorize
d: no vectype for stmt: _40 = MEM[(const __m128i_u * {ref-all})_3];
 scalar_type: const __m128i_u
array.c:13:6: note: vectorized 1 loops in function.
```

- See more: http://hpac.rwth-aachen.de/teaching/sem-accg-16/slides/08.Schmitz-GGC_Autovec.pdf
- LLVM Compiler: https://llvm.org/docs/Vectorizers.html

Vectorize Loop with Intel Intrinsics

- Intel Intrinsics Guide: link
  - Check the instruction set you want to use
  - Use keyword to search
  - Check the variable type & operation
- Procedure
  - Load data from memory to the special registers
  - Perform vector instructions
  - Save data from the special registers to memory



GPU & Accelerators

Benefits and Drawbacks of a CPU

- ???

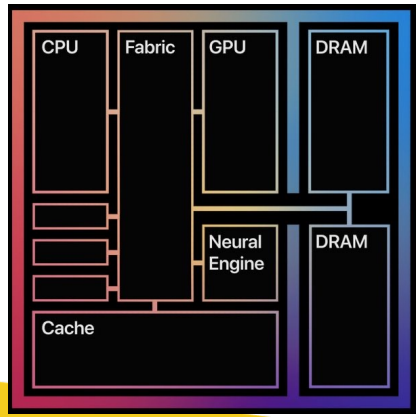
Benefits and Drawbacks of a CPU

- Can perform any types of calculations
- Flexible

- Requires reading a lot of instructions
- Slow

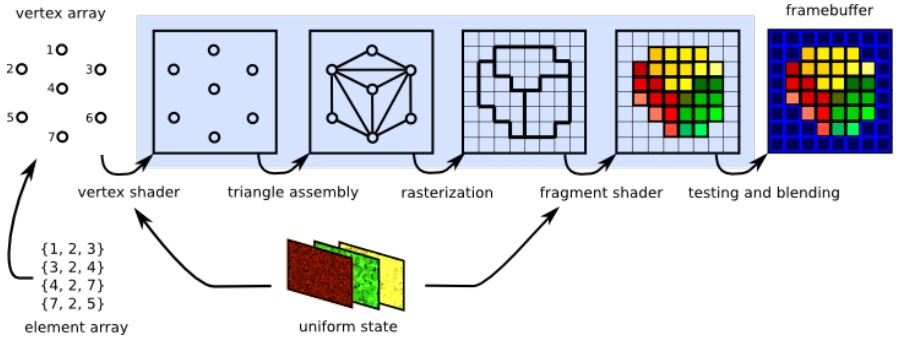
Heterogeneous Computing

- Application-specific chips
- CPU: general purpose computing
- GPU: graphics rendering
- Neural Engine: matrix multiplication accelator
- Media Engine: video decoding & encoding
- ...



Graphics Processing Units

- Originally design for rendering 3D scenes
- Highlly paralleled
- Have a lot of ALUs performing similar computation



https://graphicscompendium.com/intro/01-graphics-pipeline

General Purpose Graphics Processing Units

- Proposed to conduct general purpose compuation just like CPU
- GPU features a lot of ALUs, providing massive throughput
- NVIDIA 4090: 16,384 CUDA cores
  ➡️ 16,384 ALUs
- Programmed in a SIMD fashion:
  1 instruction ➡️ 32 data

Benefits and Drawbacks of a CPU

- Suitable for parallel computing (e.g., matrix multiplication)
  - Application: Deep learning, solving linear equations...
- Higher throughput
- High Performance Computing ✅

- Single core is slower than CPU
- Not OK for sequential algorithms

Hands-on Lab

Automatic Vectorization

- We provide a program named `lab1.cpp`
- Please compile and optimize it using:
  - SSE
  - AVX2
  - AVX512
- Complete the compararison and submit a report based on the template

- `g++ lab1.cpp -O3 -fopt-info-vec-all -ftree-vectorize -m<sse, avx2, avx512f>`

Automatic Vectorization

- For instance:

```
g++ lab1.cpp -O3 -fopt-info-vec-all -ftree-vectorize -msse
```