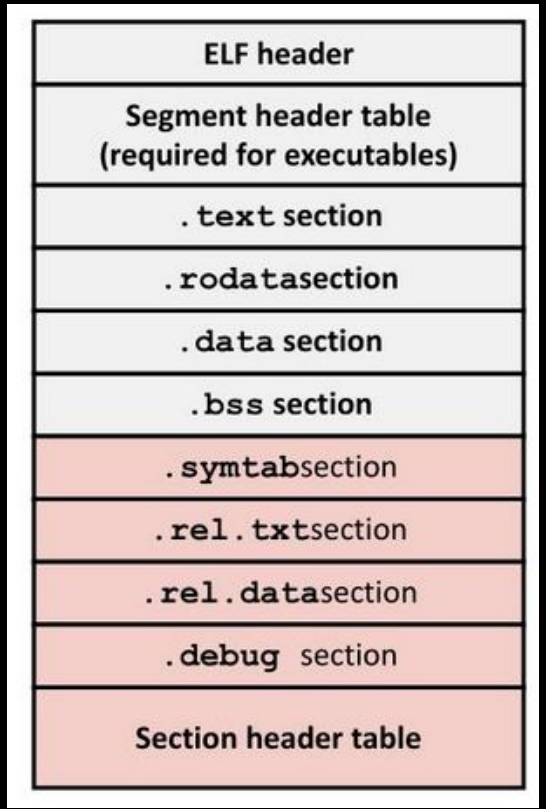
Object Loading

By: Thomas & Tanush

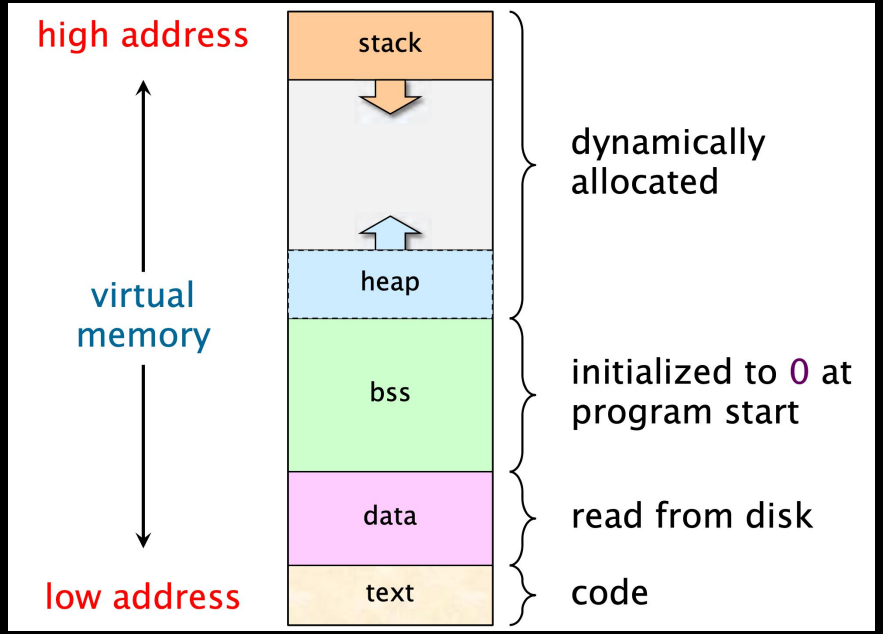
What is ELF again?

- ELF is an *object* format.
- An object means that it holds some machine language data, which we can execute.
- The question is, how does the OS actually load and execute an ELF?

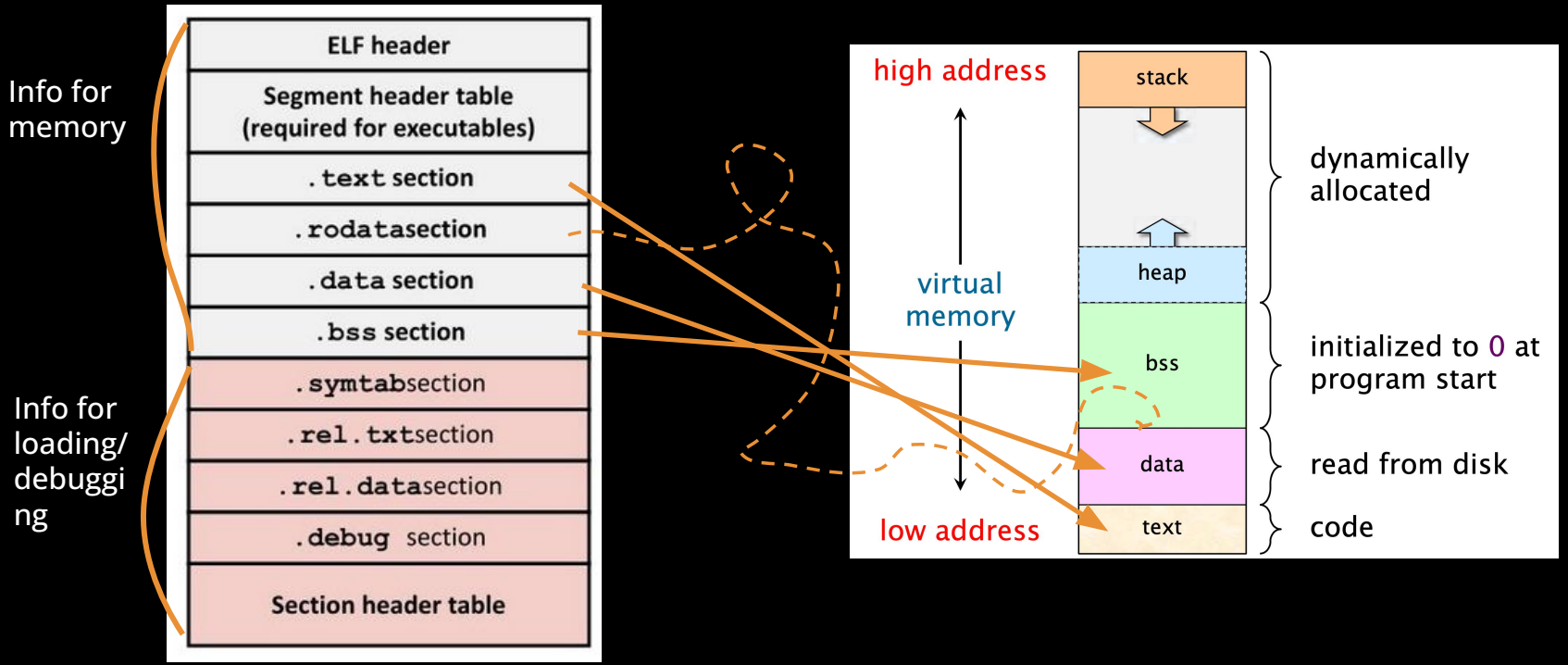


Virtual Memory

- Virtual Memory is the memory space that processes actually exist in.
- .text holds the executable code (read/execute).
- .data holds non-zero-initialized global variables (read/write)
- .bss holds zero-initialized global variables (read/write)
- And there's also .rodata, which holds string literals (readonly).



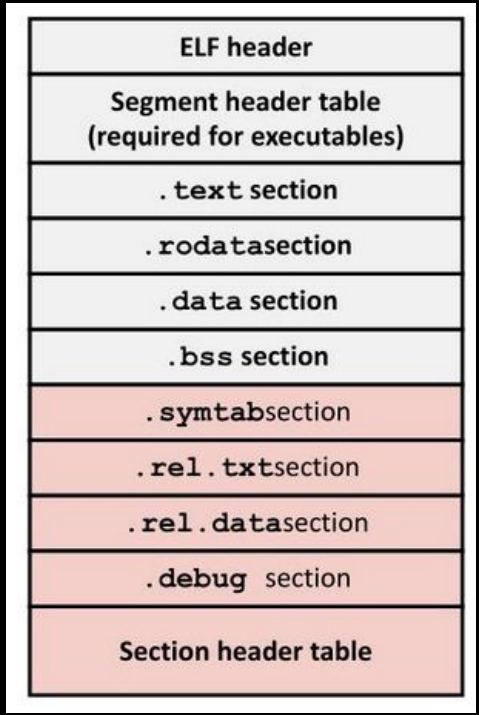
Comparing Models



As you see, there's a strong connection between ELF segments and the virtual memory layout.

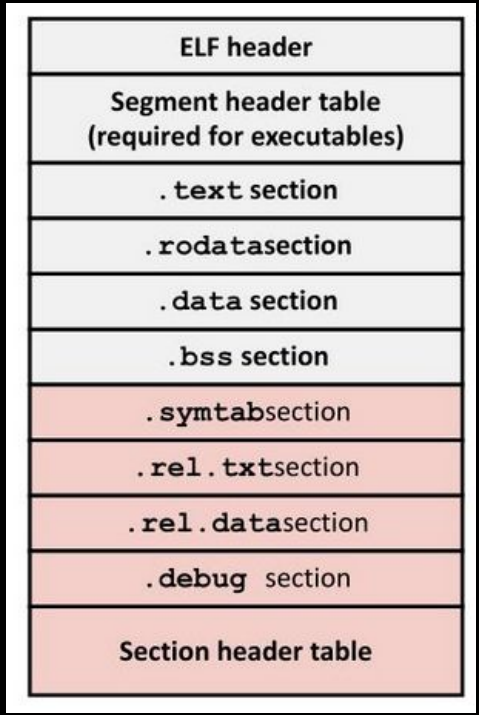
Loading .text

- Text is executable memory, which we must tell the OS to allocate.
- You just copy all the .text data into virtual memory, mprotect it to PROT_EXEC
- Now, you can just JMP to a symbol and execute a function.



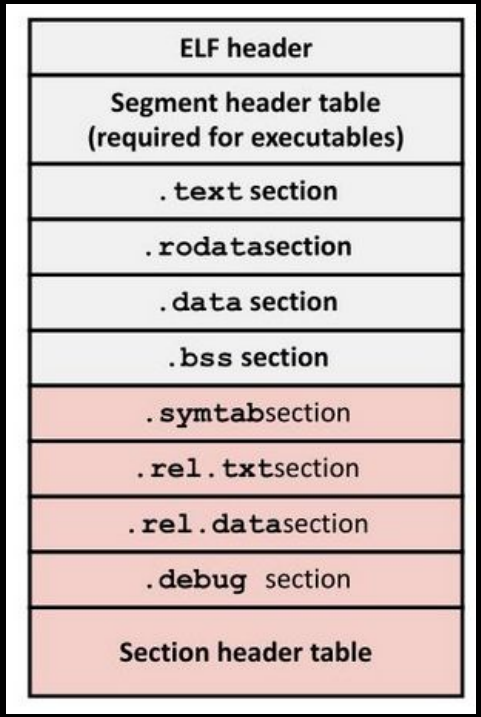
Those Pesky Relative Sections

- Unfortunately, we are not able to call other functions, because the compiler cannot assume relative location of functions and code them in (assuming non-static).
- Thus, the .rela sections exist to tell the loader where to patch in actual relative jumps.
- The address formula is L + A - P
  - L = Symbol address
  - A = Append constant (in .rel section)
  - P = The memory offset .data is stored in



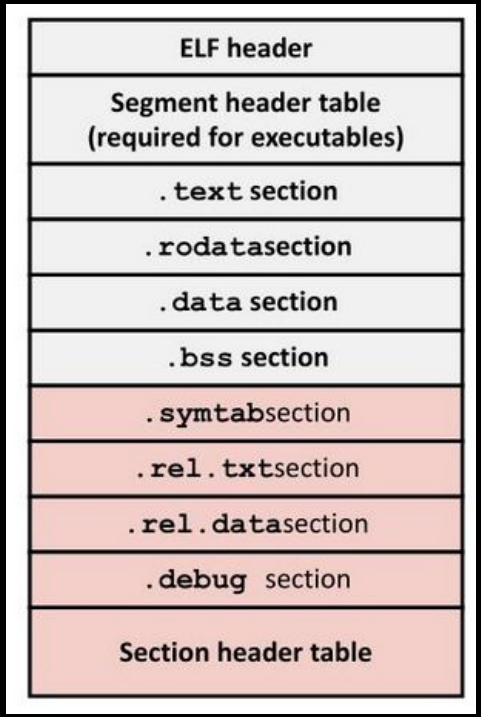
Those Pesky Relative Sections

- The address formula is L + A - P
  - L = Symbol address
  - A = Append constant (in .rel section)
  - P = The memory offset .data is stored in
- You have to patch the memory data that you've copied, and replace the offsets referenced with our calculated offsets.
- This allows for variance in memory layout.



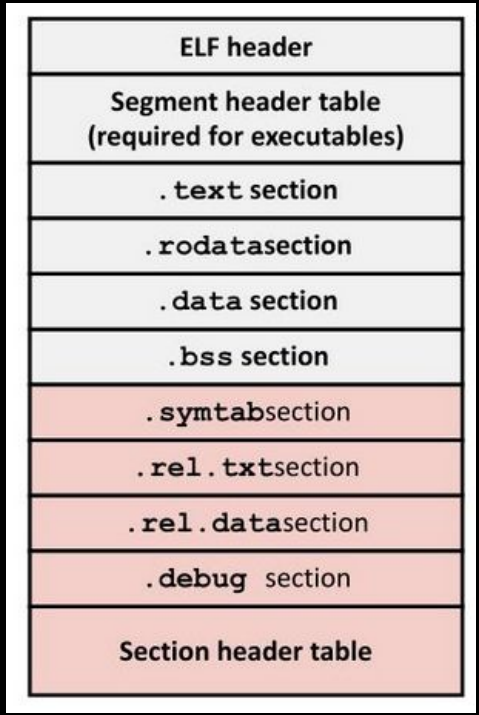
Loading .data

- The data section holds the global variables that are not allocated to zero.
- It should be made read/write via mprotect.



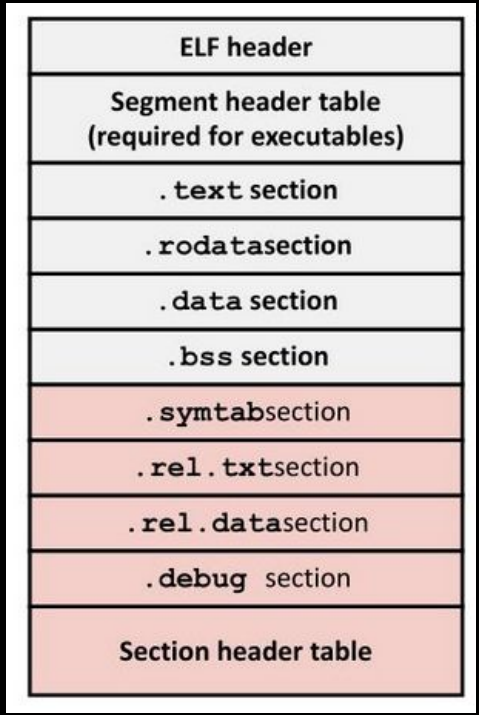
Loading .data (But relative...)

- .rel sections are similar to how they exist in text, but made more complicated because they're possibly different allocations.
- Many instructions may expect e.g. a 32-bit relative jump, but it's possible that they were allocated further in virtual memory.
- As such, it's recommended to follow with the conventional memory layout to ensure they are all allocated nearby.



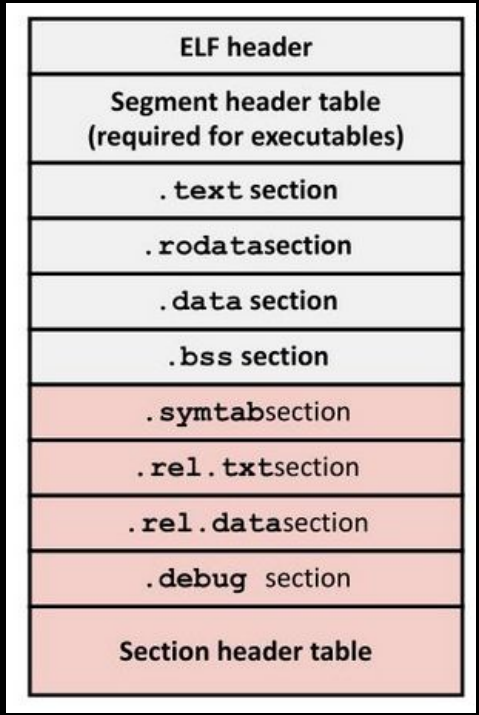
Loading .bss

- BSS is the zero-initialized global variables.
- The reason it's separate from .data is that it doesn't store all of the zeros it needs in the ELF, but merely stores the size required for when it is loaded.
- Just mmap the size you need, and you're done.
- The relative stuff is the same for .data.



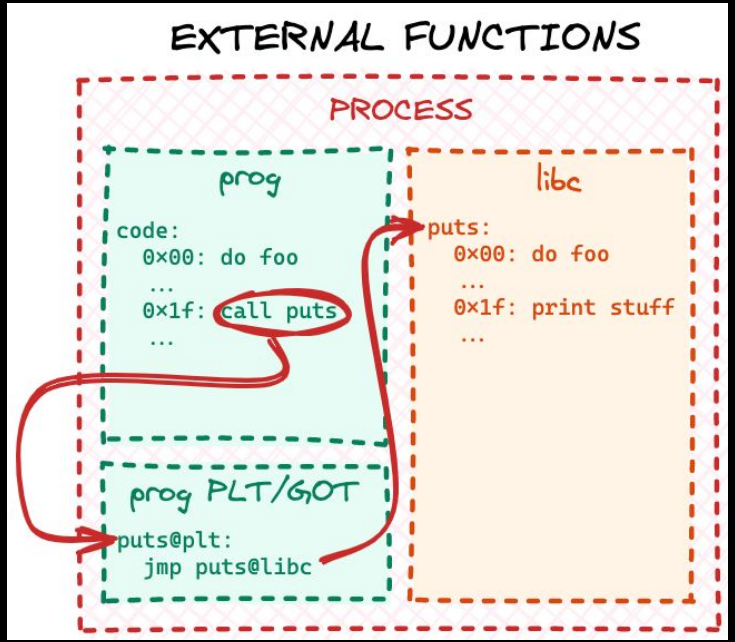
Loading .rodata

- .rodata is the section that stores… read-only data.
- It’s mainly for string literals, such as the “Hello World” we all know and love.
- Other than that, it’s the same as .data.
- It’s interesting that it can make sense to layout the other data sections *under* .text to ensure close relative jumps.



External Libraries

- All of this has relied on the data we wanting being placed within the same ELF file.
- During the linking step, the linker creates the Procedure Linkage Table (PLT) and the Global Offset Table (GOT).
- This is where **ld** comes in, the *dynamic loader*, which lazily processes the jump table references and turns them into the actual runtime addresses.



PLT/GOT

- If you find an entry in the .rel sections that points to 0, that means it’s pointing to the PLT/GOT.
- We then have to allocate our own jumptable (readonly/execute), which takes looks up the external function and then executes it.
- LD/ld-linux.so load the shared objects and does this lookup for us.

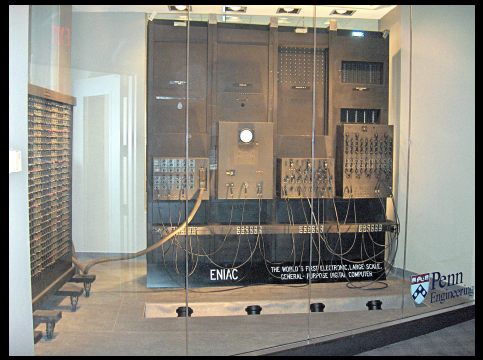
Von Neumann Architecture

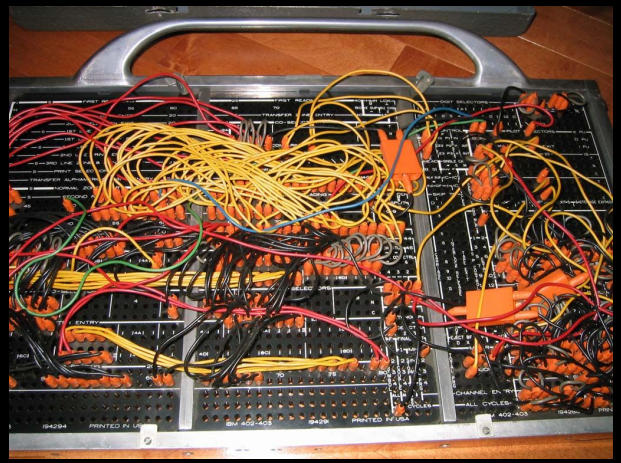
- Created by John von Neumann in 1945
- The original idea was an architecture for an electronic digital computer made up of “organs”
  - A central arithmetic unit to perform arithmetic operations;
  - A central control unit to sequence operations performed by the machine;
    - Memory that stores data and instructions;
  - An "outside recording medium" to store input to and output from the machine;
  - Input and output mechanisms to transfer data between the memory and the outside recording medium.

Von Neumann Architecture - What it fixed

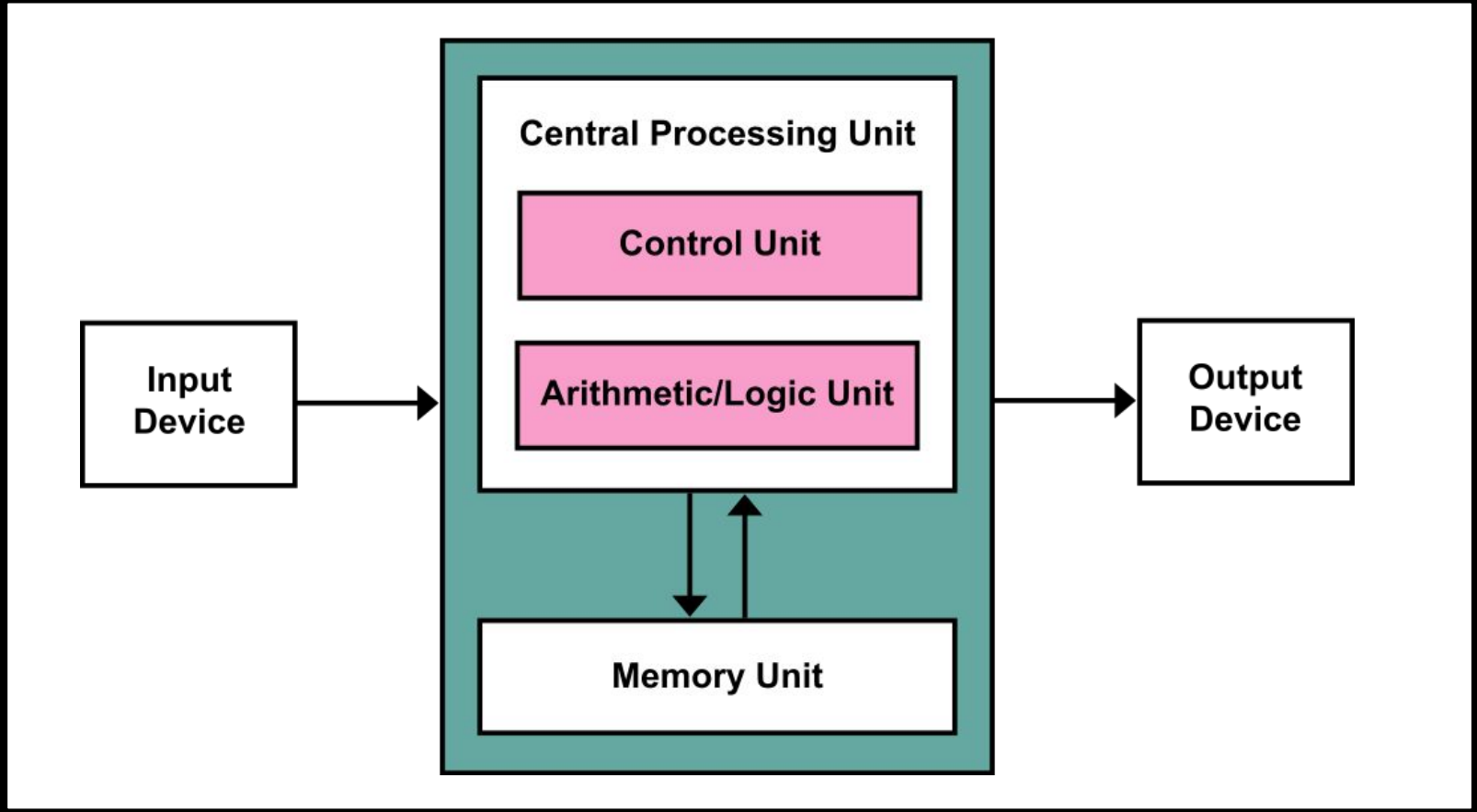
Fixed-program / rewired electronic computers

- Early electronic machines could compute fast, but changing the program was painfully manual.
- They were a pain in the ass since we can't easily
- Changing the program execution was horribly slow
    - Ie we could make a system that does very complex
    - Things very fast
        - Everything is just bits/electronics
    - But reorganization took days or weeks





Von Neumann Architecture

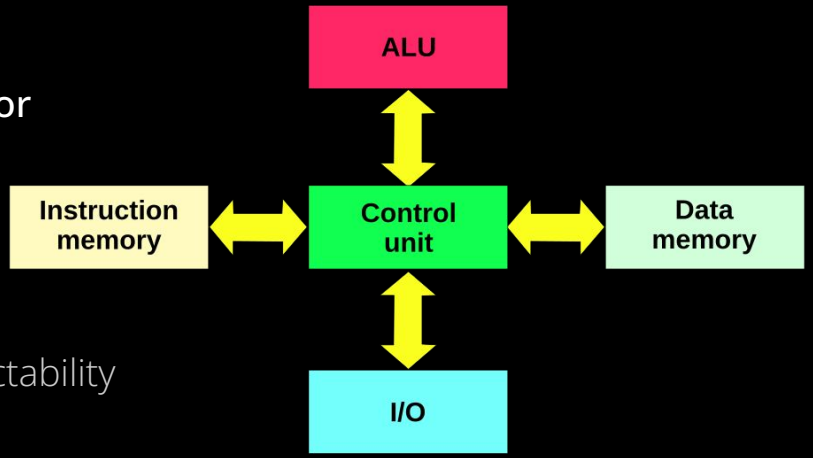


Von Neumann & Harvard Architecture

- It is now synonymous with to any Stored-program computer
  - Any system which instruction fetch and a data operiaton can't take place at the same time since they exist on a common "bus" / communication layer
    - This is known as the von Neumann bottleneck

The harvard architecture

- You have different a different communication path for
  - instructions and data
  - Used in RTOS/ Low-power systems
- Pros :
  - Faster (can fetch/decode) at the same time, Predictability
  - And can have security by default
- Cons: more complex hardware and if you have bottle neck
- One side but not the other



Why it Won in the End?

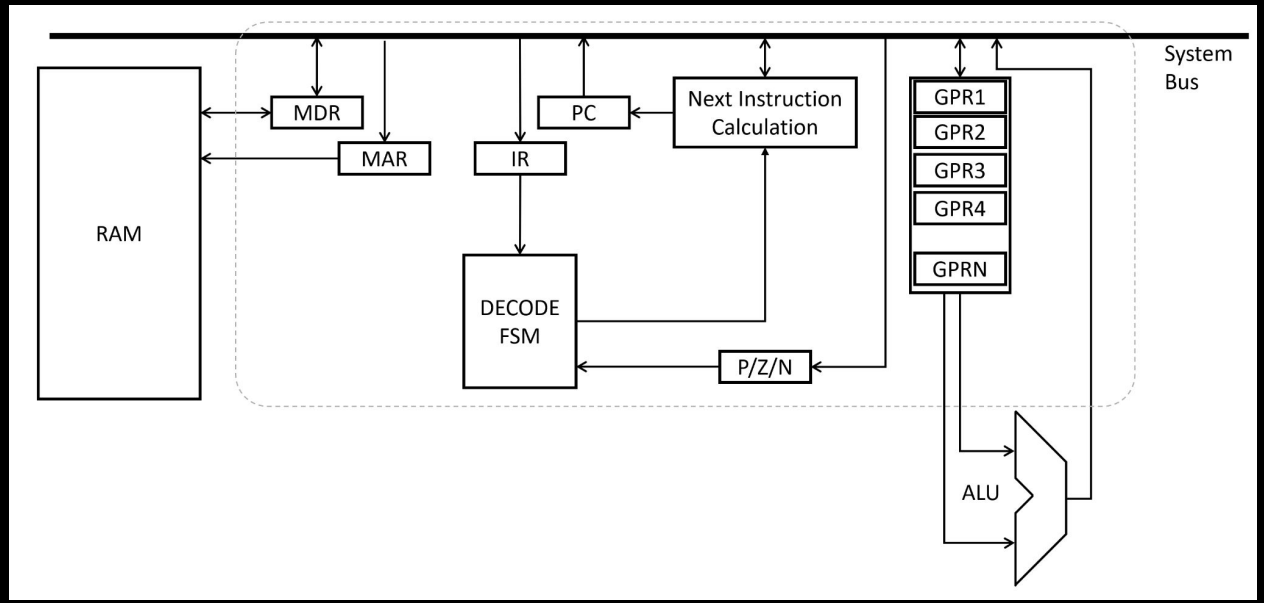
- Reprogrammability
  - (fast iteration, broad usefulness - think how much a computer does
- Unifying memory
  - made early designs practical when memory was expensive.
- The model scaled well even as we added caches, pipelines, and parallelism

Modern systems:

- Separate I-cache and D-cache (so instruction and data can be fetched in parallel)
- Deeper cache hierarchies + prefetching to hide memory latency
- Wide buses, burst transfers, out-of-order execution to keep the CPU busy

Control Unit

- “The heart” c
  - coordinating the action of registers and the ALU together to do actual calculation.
  - Made for stored-program

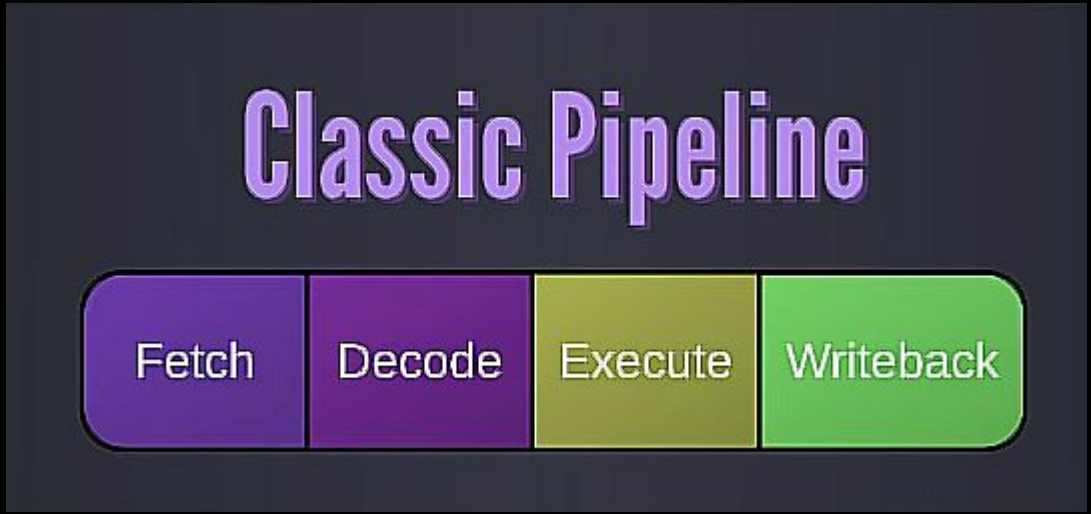


Control Unit - Registers

- Every CU has the same set of registers to store info
  - This is the fastest form of data and memory storage

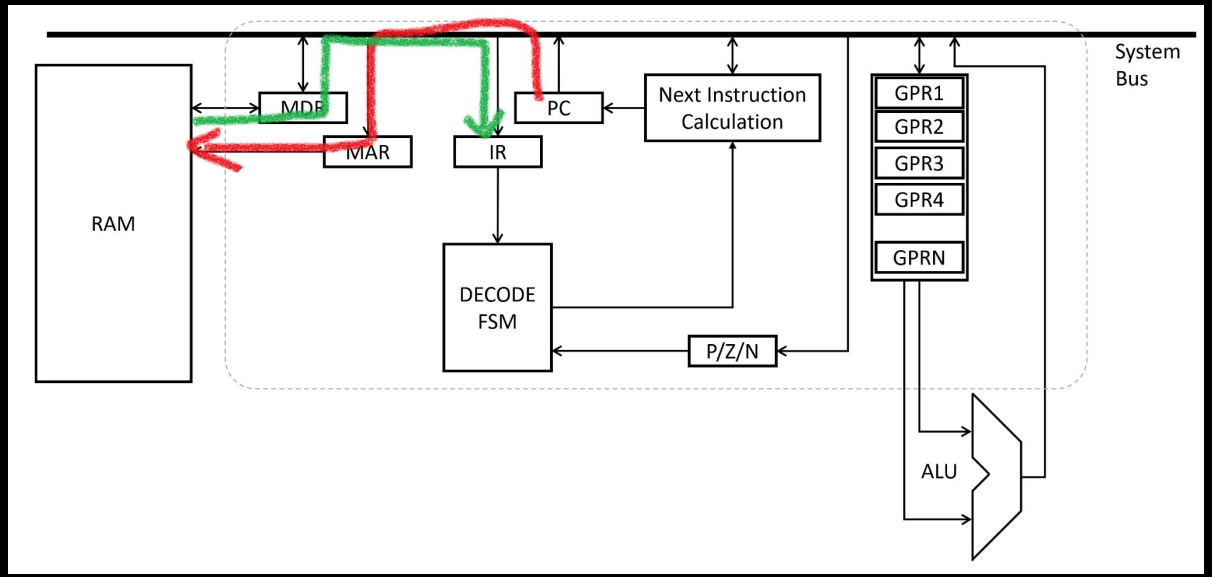
- MAR/MDR - memory address register and memory data register
  - Two key registers
  - They are connected directly to the RAM allow the CU to read and write from specific memory address
  - They laid the desired memory address
    - Then control signals are are sent to decide to read or write that address
      - For read we pull it off the RAM and place it into the MDR
      - Write we move it from the MDR to the RAM location defined in the MAR

The Execution Cycle



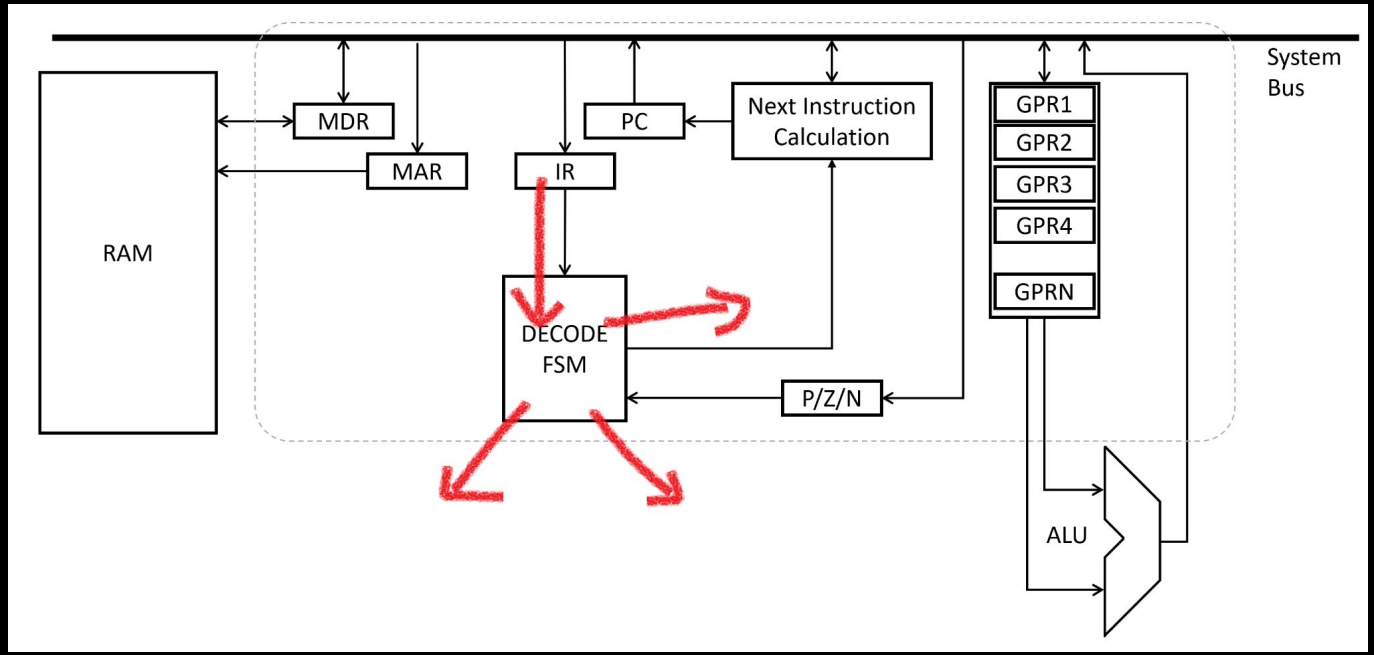
The Execution Cycle - Fetch

1. Instruction Fetch cycle
    a. We use the PC program counter or RIP instruction pointer in x86
        i. Add 4 to the PC and store the PC
            1. In MIPS its actually stored in new PC into an internal register called NPC
    b. The value is used to populate the MAR - memory address register
    c. We then do a read, to put data in the MDR and that is set to the instruction register (IR)



The Execution Cycle - Decode

2. Instruction Decode/register fetch cycle (ID)
    a. Determine which instruction we have, and then what register values do we need
        i. Set condition flags as needed

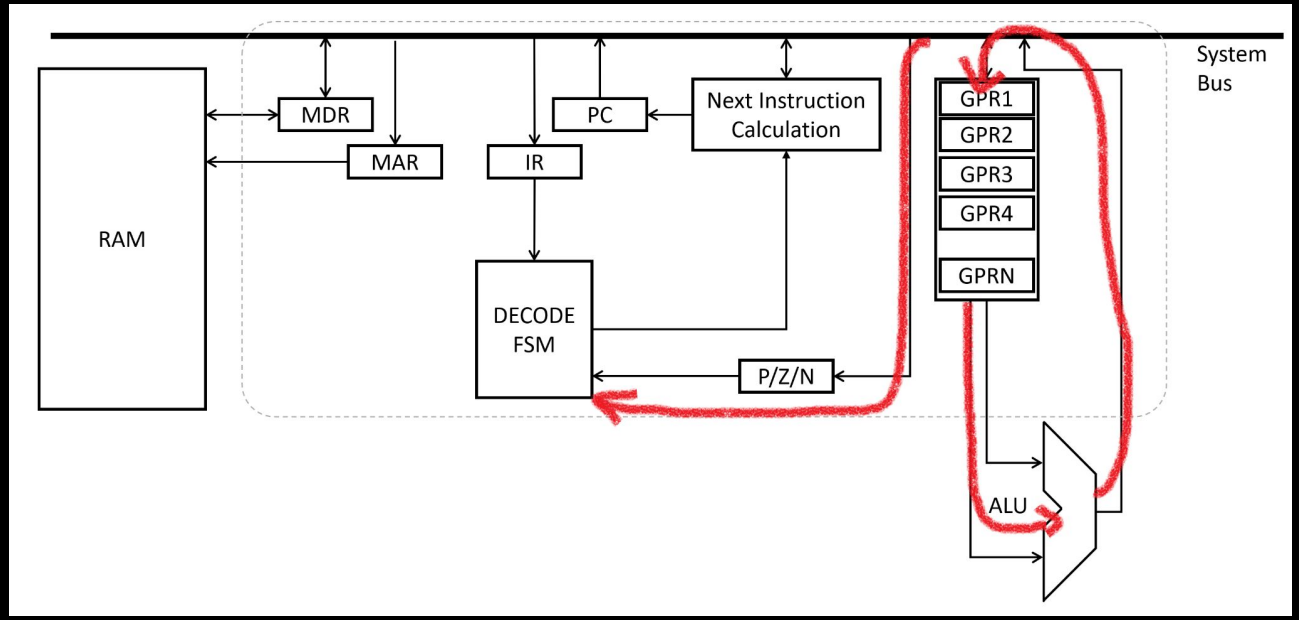


The Execution Cycle - Decode

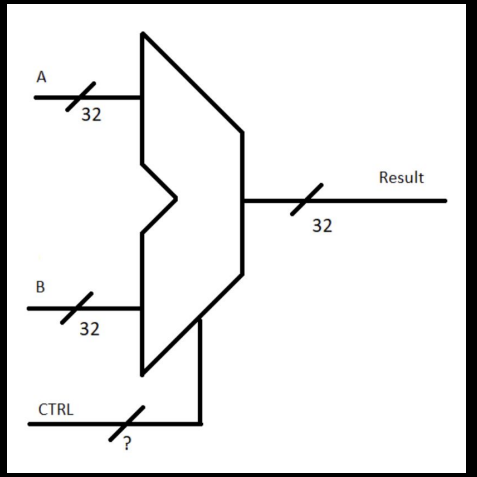
1. Instruction Decode/register fetch cycle (ID)
    a. Memory references
        i. The ALU adds the base register and the offset to form the effective address
    b. Register-register
        i. ALU instruction performs the operation of (addition /multiplication)
    c. Register-Immediate
        i. ALU instruction: perform the instruction on the first register read and the immediate value in the instruction.
2. Memory access
    a. If the instruction is a LOAD or a STORE, do the appropriate thing,
        i. update the PC using either NPC or the output of the ALU operation
3. Write-Back cycle (WB)
    a. If the instruction was LOAD, write the value fetched from memory

The Execution Cycle - Execute

We will execute the requested instruction here this is done by the ALU and after we update the program counter



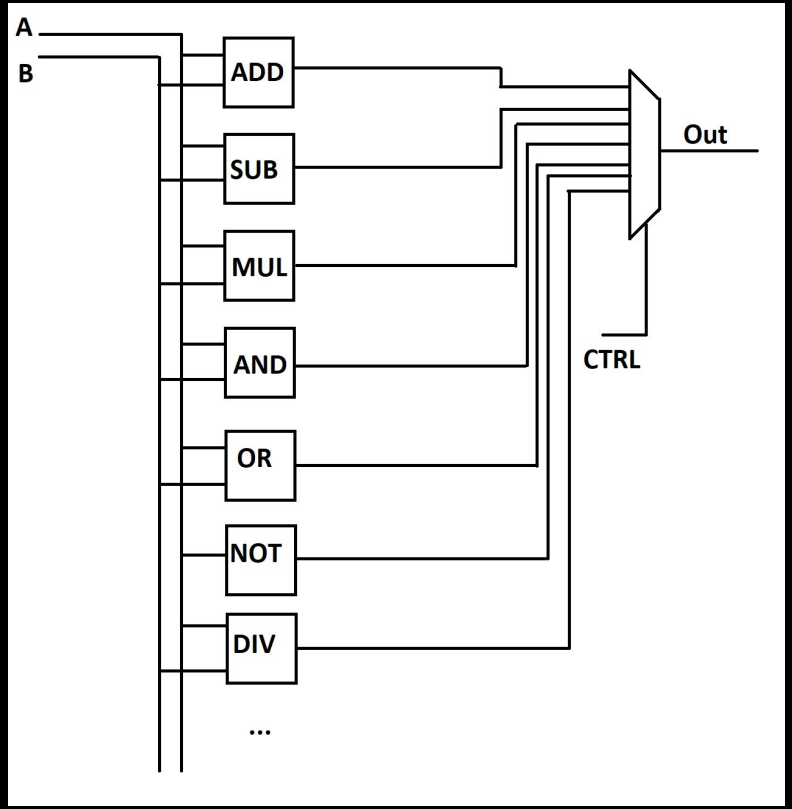
Processing Unit



It depends per process but the basic idea:

- 3 inputs
  - A & B
  - CTRL - what action to perform on these values `output = f(A, B, CTRL)`
    - This is a bit mapping ot certain basic operations
- There can by any number of CLTR bits just depends on the ways to combine them
  - Its key to remember there turning complete nature of some instruction allowing them to build others. Some very small ALU's might just have bitwise AND. bitwise NOT and addition
-

Combine everything into an ALU



Choosing the Next Instruction – Dealing with branches

Based on the flags set depend on the execution - carry/zero/etc

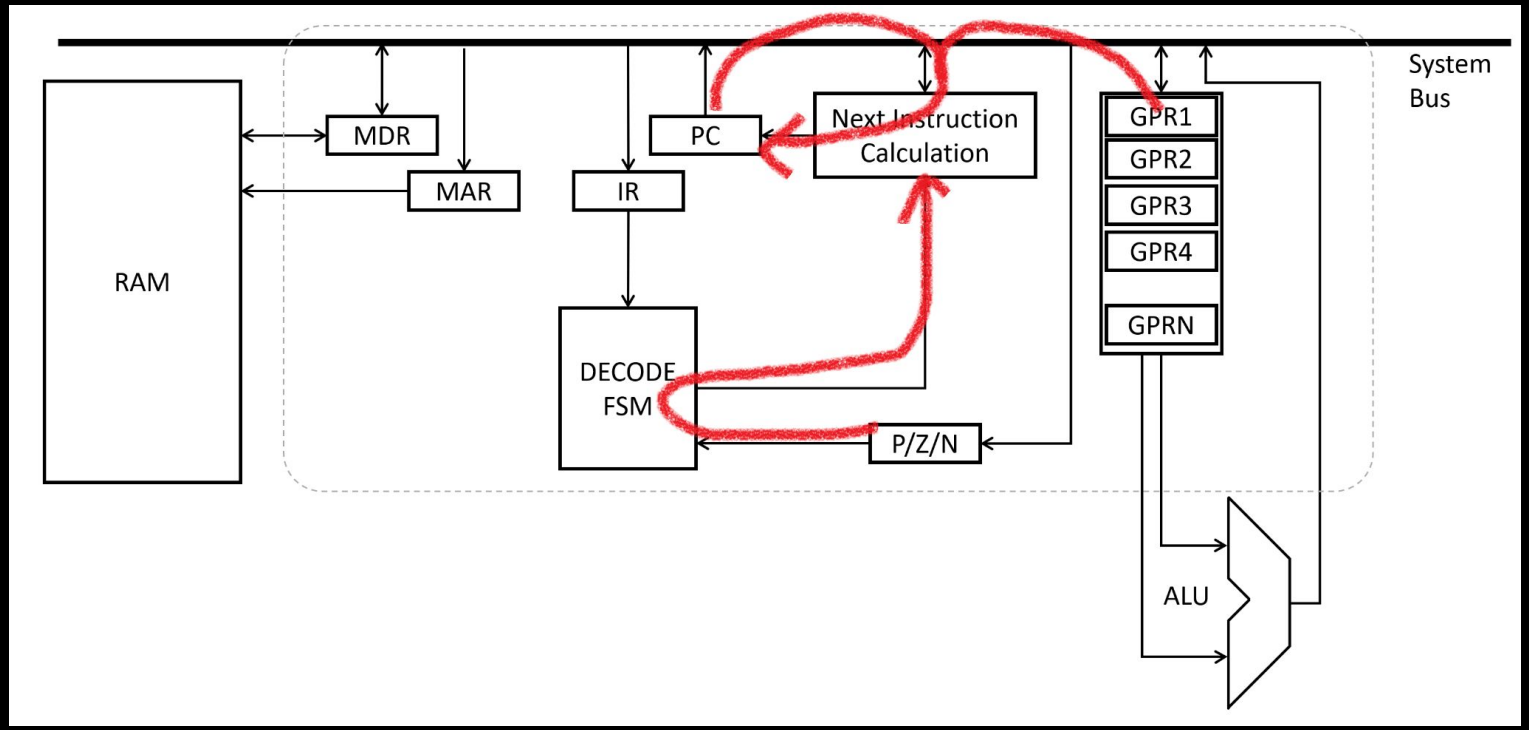
- Mind every flag is always set or unset ie 0 or 1, the previous action might set them based on its results

The jump can be made in several ways

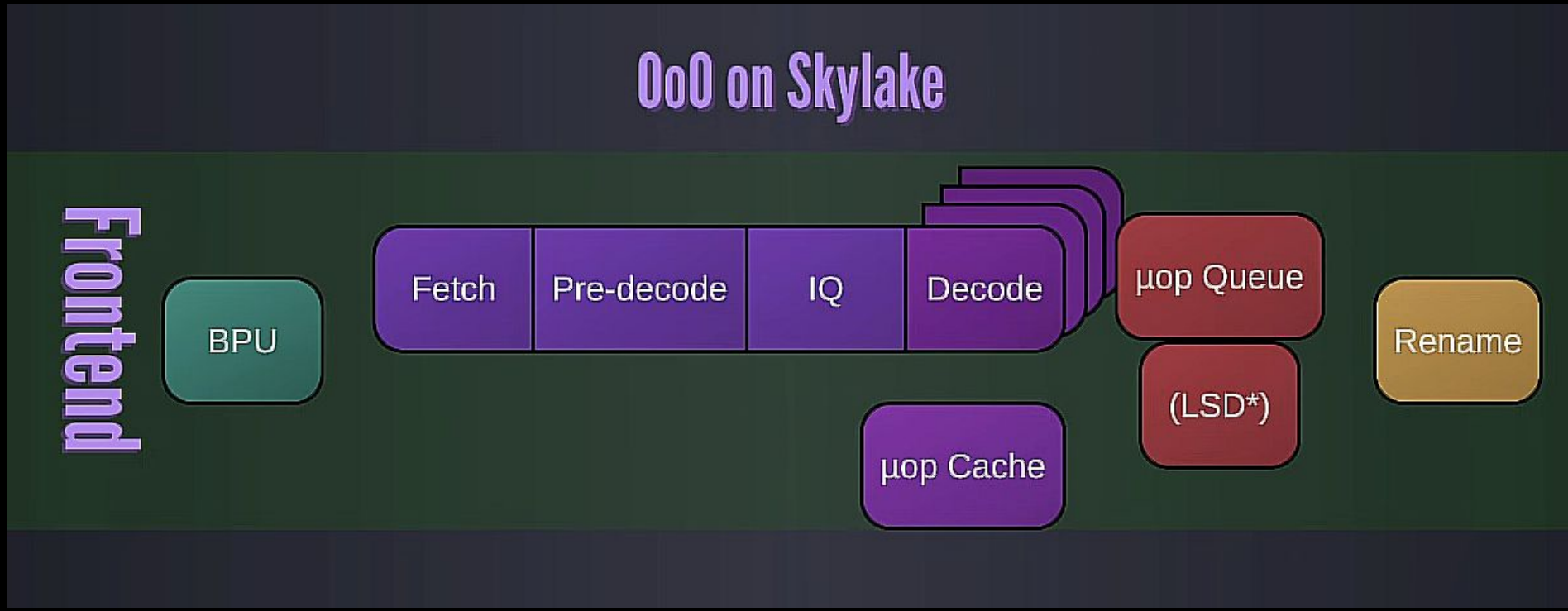
- we might encoded it into the instruction word itself
- pulled from a general purpose register or so on

Regardless it's just a means to an end to select what our PC is set to

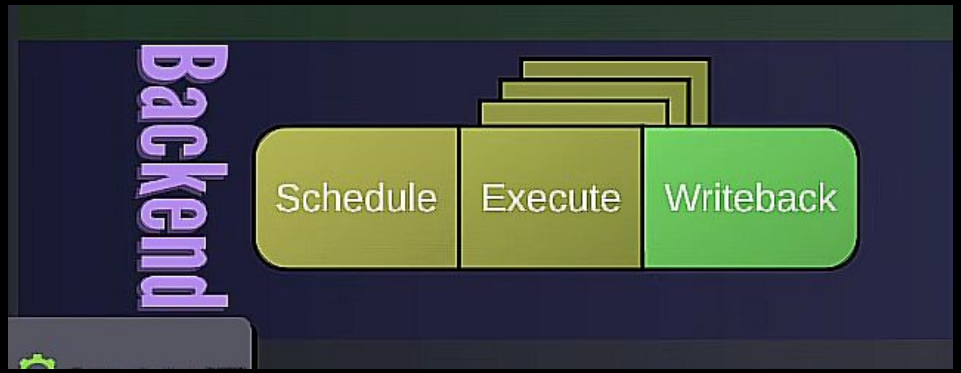
The cycle starts all over again !

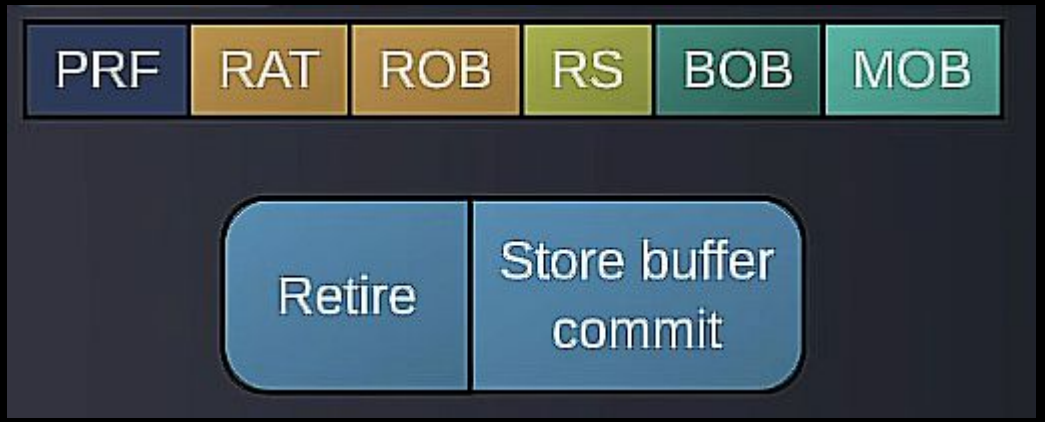


Modern Systems and an overview



Modern Systems and an overview





Modern Systems – Front End

The goal here is to fetch and decode instruction and translate them into micro-operations fast enough to keep execution speed up

- Fetch
  - Reads 16 byte of code at a time, the branch prediction unit helps here
- Pre-decode
  - Identify complex instruction that need to be broken down into uOps
- Macro-fusion
  - Combine pairs of instruction
- Decode 0
  - Complex decode can handle 4 uOps
    - Deals with switches for DIV and CPUID
- Decoders 1-3
  - Simple - handle 1 uOp

Modern Systems – Renaming

The renames will break down false dependencies by mapping the small set of architectural registers to a massive pool of internal physical registers

- Ie maps your EAX/RBX to PRF
  - There are hundreds of actual physical registers

The idea of false dependencies are the idea that we can parallely execute instruction if we know the previou value don't really matter for example

- mov eax, 1
- add eax, 2
- mov eax, 7

Register Alias Table - RAT

- Maps the logical registers to physical registers

Zero-latency idioms

- Say we do like xor eax,eax to clear a register we map it to a special zero reg that doesn't require execution

Modern Systems – The backend

Scheduler - (Reservation Station)

- Holds uOps till
  - their input operands are ready
  - An execution port is free
- Execution ports
  - These are our ALUs essentially
- Writeback
  - After execution is done we we broadcast the results if anything is waiting on the instruction in the schedular or the physical registers
- Retirement
  - Reorder buffer: We keep track of all in flight instruction in theri original program order
  - We are only fully done when an instruction reaches the head of the ROB
  - Commit, we write results to real memory
  - Freeing: physical register are freed only when the instruction is overwritten

Whatever Tanush puts here...