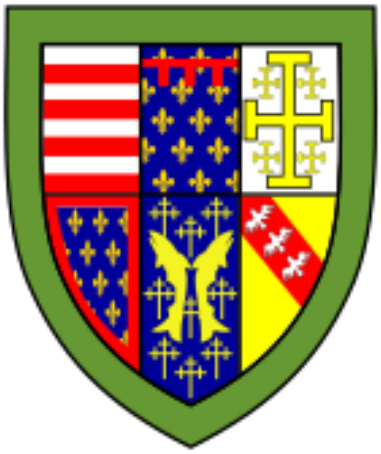
Queens' College Cambridge

Computer Design



Alistair O'Brien

Department of Computer Science

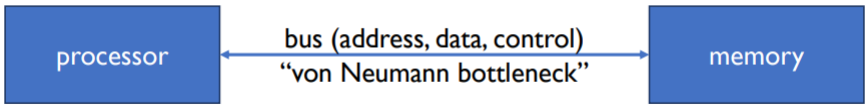
June 2, 2021

Contents

1 Background

1.1 Von-Neumann Architecture

Definition 1.1.1. (**Von-Neumann Architecture**) A computer architecture where data and instructions are stored in a linear addressable memory



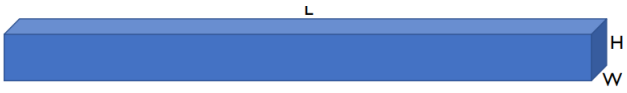
- **Stored program concept** is the idea that instructions and data are stored together in memory, which can be fetched, decoded and executed.
- **Problem**: The *Von-Neumann Bottleneck*. May only address either data or instructions (not both) $\implies$ data rate $<$ processor execution rate.
- **Solution**:
  - Harvard Architecture:
  - Memory Hierarchy
  - More registers.

1.2 Moore's Law

- **Moore's law** states that silicon transistor density doubles every 2 years.

- **Problem**: *However*, the rate of increase is slowing (due to issues e.g. quantum effects).
- Moore's law also applies to DRAM and Flash memories.
- **Kryder's law** for magnetic hard disks, hard drive density (bits per mm^2) doubles every 13 months.
- Latest transistor technology is more expensive. Industry is beginning to use previous generation processors.

1.2.1 Wire Speed



- Consider a wire of length ℓ, width w and height h. The resistance of the wire is
$$R = \rho\frac{\ell}{A} \propto \frac{\ell}{w \times h},$$
where $A = w \times h =$ area.
- The capacitance C of the wire is
$$C = \varepsilon\frac{A}{d} \propto w \times \ell.$$
(This is a parallel plate capacitor where the surroundings are the other plate and the plate area is the base of the width $w \times \ell$). Hence
$$\tau = R \times C \propto \frac{\ell^2}{h},$$
where τ is the time constant.
- **Conclusion**: Desire to minimize τ, hence ℓ^2 and maximize h (tall short skinny wires). Place buffers / repeaters between short wires.
- Requires repeaters / buffers between short wires to produce larger wires.

1.2.2 Dennard Scaling

- Transistor dimensions scaled by $\times 0.7$ every generation, thus halving their area ($0.7^2 \approx 0.5$) (Moore's law).
- Reduces delay $\times 0.7$ ($\mathrm{d}t = v\,\mathrm{d}s$), increases operating frequency by $\times 1.4$ ($f = 1/T$).
- To keep electric field E constant, V (voltage) reduced $\times 0.7$ ($V = E \times d$) $\implies$ reducing energy $\times 0.35$ ($E = P \times t$) and power (at $\times 1.4$ frequency) $\times 0.5$. ($P = CV_{DD}^2 f/2$)
- **Conclusion**: transistor density doubles $\implies$ circuit is $\times 1.4$ faster (frequency), power consumption (with double transistors) is equal.
- **Problem**: Around 2006, cannot decrease V_{DD} less since transistors leak too much charge (static power dramatically increases compared to dynamic power) .Wires don't scale for the same chip area.
- Desire for better architecture for performance increase.

1.2.3 Dark Silicon

- Minimize number of switching transistors $\implies$ minimize dynamic power.
- **Accelerators**: Domain optimised circuits (optimizations minimize dynamic power). Switched on only when required (minimize static power).
- In computers, use small efficient cores for light loads, large cores for heavy loads. (*Hetrogeneous cores*)
- Race-to-dark (RTD) silicon, turn off unused cores. Saves static and dynamic power.

1.2.4 Optimizations

- **Dynamic Voltage Frequency Scaling**:

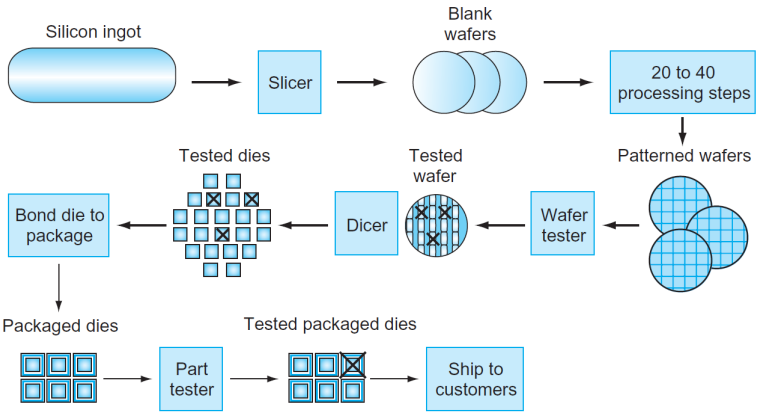
  - Adjust voltage and frequency dynamically wrt performance and dynamic power:
$$P_D = \frac{1}{2} C V_{DD}^2 f \mathcal{A},$$
where $\mathcal{A}$ is the activity factor (proportion of transistors that are switching).
  - Voltage V_{DD} is limited because increasing f increases V, which increases resistance ($R = V/I$) (hence increases temp.) CMOS circuits degrade as temperature increases. (Thermal throttling)
  - Aggressive cooling allows f to be increased (Overclocking).

- **Idle Power**:
  - Power $\not\propto$ Load due to state power + resistance.
  - RTD attempts to achieve energy-proportionate compute.

- **Solution**:
  - Energy proportional compute: Designing processors (compute units) w/ Power $\propto$ Load
  - RTD: A means to achieve energy proportional compute w/ hetrogeneous architectures

1.3 Manufacturing Processors

- Integrated circuit (IC) combined hundreds of CMOS transistors on a single chip.
- Manufacturing process starts with doped silicon wafers. Wafers are patterned using a **mask**.
- **defects** in the wafer (microscopic flaws / pattern failure) result in an area of failure on the wafer.
- Wafer is chopped/diced into components called dies / chips. dies with defects are removed.
- **Yield**: the proportion of fully functioning dies.



$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{Yield}}$$

$$\text{Dies per wafer} \approx \frac{\text{Wafer area}}{\text{Die area}}$$

$$\text{Yield} = \frac{1}{(1 + (\text{Defects per area} \times \text{Die area}/2)^2)}$$

Since dies are rectangular and wafers are circular.

- **Costs**:
  - Non-recurrent (one-off) costs e.g. mask costs for patterning $\approx$ \$10m. $\implies$ ICs manufactured iff high volume to offset costs.
  - Cost per transistor increasing, industry split into cheap processors and high-performance processors (expensive).

1.4 The Eight Great Ideas in Computer Architecture

1.4.1 Moore's Law

- Traditional assumption, no-longer holds.
- Now require new design models e.g. Intel's tick-tock model:

- **Tick**: *New* architecture tested on *existing* silicon (performance via architecture)
- **Tock**: *Existing* architecture tested on *new* silicon (performance via silicon)

Prevents debugging architecture on untested silicon (vice-versa)

1.4.2 Abstraction

- Characterize the design of hardware/software into different levels of representation (abstractions). Forming a stack, where level x_i is only aware of level x_{i-1} and x_{i+1}.
- Used w/ Hardware Definition Languages (HDLs).

1.4.2.1 Architecture Interfaces

- **Instruction set architecture** (ISA) is an abstract interface provided by the processor (implementation of ISA). e.g. RISCV ISA.
- **Application binary interface** is the user portion of the instruction set + interfaces provided by the OS.

1.4.3 Common Case Fast

- **Idea**: Make the common instruction / operators fast + efficient
- Governed by Amdahl's law for sequential programs:

$$t \text{ after improvement} = \frac{t \text{ affected by improvement}}{\text{amount of improvement}} + t \text{ unaffected} ,$$

where t is execution time.

- e.g. If integer division is used 1% of the time, then an improvement of $\times 10$ results in a total improvement of

$$\frac{0.01t}{10} + 0.99t = 0.991t,$$

hence a 0.9% improvement. (Although adding integer division decreases clk frequency, so improvement isn't worth it)

1.4.4 Parallelism

- **Data-level Parallelism**: Instructions act on more than 1 data (SIMD).
- **Instruction-level Parallelism**: Execute more than 1 instruction per clock cycle (superscalar pipelining)
- **Thread-level Parallelism**: Execute n concurrent threads on m processors ($m = 1$ is known as *concurrency*)

1.4.5 Prediction

- Predict what instructions are executed next to improve efficiency of pipelining.
- **Control-flow prediction**: predict the instructions to be executed next
- **Data-value prediction**: predict the outcome of an instruction before it executes e.g. predicting whether a value will change.
- **Data-access prediction**: predict what data will be accessed. Ensures that data needed soon is accessible with low-latency.

1.4.6 Dependability via Redundancy

- Common forms of redundancy: Error Correcting Codes (ECC), RAID (Redundant array of inexpensive disks).
- Extreme redundancy: Triple modular redundancy, three independent processors executing the same program. Majority voting is applied to the output. e.g. Space shuttle.

1.4.7 Hierarchy of Memories

- **Volatile** memory retains data stored if it is powered.
- **Non-volatile** memory retains data stored in the absent of power.
- **Cache memory** consists of blocks of low-latency memory (SRAM), acts a buffer to higher-latency memories (DRAM).

- Uses automatic transitioning between memories to provide an interface of addressable memory. (Abstraction) A cache provides an abstraction for a *large and fast* addressable memory.

1.5 Performance

- Performance is defined wrt a metric e.g. speed of execution (more commonly referred to as *performance*), energy usage (power), silicon area, etc.

Definition 1.5.1. (**Performance**) Performance (wrt execution time) of a processor X is defined

$$\text{Performance}_X = \frac{1}{\text{Execution time}_X},$$

where Execution time_X is the time taken for the processor to complete the task (including I/O operations, OS overhead, etc).

- The relative performance n of two processor X and Y is

$$n = \frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X}.$$

  Often say X is n-times faster than Y.

- **Problem**: Need to distinguish between Execution time and **CPU time**, the time spent by the CPU on our process, this doesn't include I/O operations, or time spent by the CPU on other processes.

- **Solution**: CPU time can be split between the user (*user CPU time*) and the system (*system CPU time*). However, it's difficult to quantify these $\implies$ generic CPU time metric.

$$\text{User CPU time} + \text{System CPU time} + \text{IO time} = \text{Execution time}.$$

$$\text{CPU time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time} = \frac{\text{CPU Clock Cycles}}{\text{Clock Frequency}}.$$

- *System performance* refers to execution time on an **unloaded system** (1 process). *CPU performance* refers to user CPU time.

- Cycles per Instruction (CPI) is the *expected* number of clock cycles per an instruction for a program. It provides a metric for comparing two different ISAs.

$$\text{CPU Clock Cycles} \approx \text{Instruction Count} \times \text{CPI}$$

  Hence

$$\text{CPU Time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Frequency}}.$$

- Different instructions i take CPI_i cycles to execution (e.g. `add`). Hence the true number of cycles is given by

$$\text{CPU Clock Cycles} = \sum_{i=1}^{n} \text{CPI}_i \times \text{Instruction Count}_i,$$

  where $\text{Instruction Count}_i$ is the number of instructions i in the program.

- The weighted average CPI is

$$\text{CPI} \approx \sum_{i=1}^{n} \text{CPI}_i \times \frac{\text{Instruction Count}_i}{\text{Instruction Count}}.$$

Factor	Affect	Reason
Algorithm	Instruction Count, CPI	Algorithm determines the number of instructions and hence the number of different instructions, thus affecting CPI as well.
Language	Instruction Count, CPI	The language and compiler influence the instruction count, since statements are translated into ISA specific instructions. The compiler also performs optimizations which also determines the number of instructions. Language/compiler also affects CPI
ISA	Instruction Count, CPI, Clock Rate	The ISA affects all three aspects, since it affects instructions, the number of cycles per instruction, and the overall clock rate.
Processor	CPI, Clock Rate	The processor determines the CPI and it's minimum clock rate.
Silicon	Clock Rate	CMOS implementation influences the clock rate of the processor.

2 RISC-V

- **Instruction set**: The set of instructions (in machine code) that can be recognized / executed on a given architecture.
- **ISA**: An architecture determined by an instruction set. (An interface)
- **RISC-V** is an ISA w/ the following principals:
  - Simplicity favors regularity. (Simple decoding, no variable length instructions, register operands always in the same position, etc)
  - Smaller is faster. (Single task interface pattern. Each instruction executes a single task)
  - Common cast fast.
- **ABI**: RISC-V uses little-endian; the least significant byte of the word is stored at the smallest address.

2.1 Registers and Operands

- Registers are fast temporary memory used within the processor.
- RISC-V has 32 (smaller is faster) general purpose registers `x0-x31` (although `x0` is hardwired to `0`).
- In RISC-V 64, registers store 64-bits (a **double word**) whereas in RISC-V 32, registers store 32-bits (a **word**).

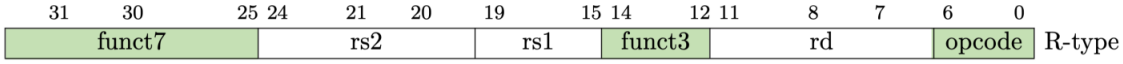
2.1.1 Operands

- Every RISC-V instruction has 3 operands. (Simplicity favors regularity).

- **Addressing mode** refers to the way in which the operands of an instruction are specified :
  - **Direct Addressing**: Operand is a register / memory address to be used.
  - **Immediate addressing**: Operand is located *immediately* after the opcode for the instruction. e.g. `addi t0, t0, 1` (`t0++`). RISC-V has many immediate instructions for ease of use (common case fast). Immediate instructions end in an `i`.

2.2 Instruction Formats

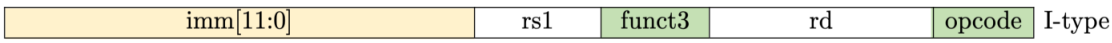
- **Instruction format** is a form of representation of an instruction composed of **fields** of binary numbers.
- RISC-V *R*-type format:



  - **opcode**: The field that denotes the operation / instruction
  - `rd`: The destination register operand.
  - `funct3`: An additional opcode field.
  - `rs1`: The first source register operand.
  - `rs2`: The second source register operand.
  - `funct7`: An additional opcode field.

  This format is used for register-based instructions e.g. `add`.

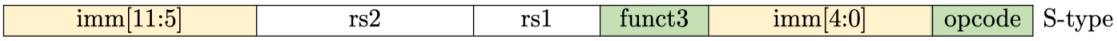
- RISC-V *I*-type format:



  - `immediate`: A 12-bit immediate operand. (sign-extended)

  This format is used for immediate addressing instructions e.g. `addi`, or `ld`.

- RISC-V *S*-type format:



  - `immediate`: A 12-bit immediate operand (split across two fields).

  The immediate operand was split across two fields to keep the *R*-type format for `rs1` and `rs2`. This format is used for store instructions e.g. `sd` which have two source registers and an immediate offset.

- The 3 instruction formats are examples of "Good design demands good compromises" (Tradeoff):
  - All registers operands are in the same position
  - All opcodes and `funct` bits are in the same position

2.2.1 Wide Immediates

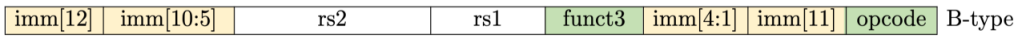
- **Problem**: Commonly require more than 12-bits for immediate operands.
- *load upper immediate* (`lui`) instruction: load a 20-bit constant into bits `12` through `31` of a register `reg[32:12]`. The leftmost 32-bits are filled with copies of bit 31, and the rightmost 12-bits are filled with zeros.
- Requires a new instruction format, the *U*-type format:

- Allows loading of 32-bit constants:

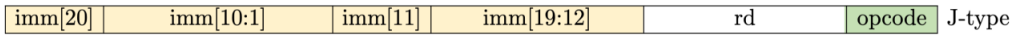
```
lui: to load bits 12-31
addi: add bits 0-12
```

2.2.2 Wide Addresses

- For wide addresses in branching instructions, RISC-V uses the *SB*-type instruction format:



- The address is encoded using a 12-bit immediate operand. Sign-extended and the 0th bit is zeroed (relative jumps are even (16-bit RISCV)).
- RISC-V also provides PC-relative addressing via SB and UJ jumps. An addressing mode where the address is the sum of the program counter and the immediate offset.
- **Problem**: Relative address limit: $\pm 12^{12}$ bytes.
- The unconditional jump-and-link uses a UJ-type instruction format:



  to provide a 20-bit immediate address.
- `jalr` is used for any jumps that is greater than $\pm 2^{18}$ words away.

2.3 Instructions

2.3.1 Data Transfer Instructions

- **Data transfer instructions** are instructions to move data between memory and registers. These include the `l`t`, s`t instructions (load and store) where t is the word type: byte, halfword, word, double word.

Operator	RISC-V Instruction	Effect	Type
Load	`l`t `rd, x(rs1)`	`rf[rd] = data[rf[rs1] + x]`	I
Store	`s`t `rs1, x(rs2)`	`data[rf[rs2] + x] = rf[rs1]`	S
Load Reserved	`lr.`t `rd, rs1`	`rd[rd] = data[rf[rs1]]` and places a "reservation" on address `rf[rs1]`.	R
Store Conditional	`sc.`t `rd, rs1, rs2`	`data[rf[rs2]] = rf[rs1]`. `rf[rd] = 1` $\iff$ valid reservation on `rf[rs2]` address.	R

2.3.2 Logical Operators

Operator	RISC-V Instruction	Effect	Type
Shift Left	`sll, slli`	`<<`	R/I
Shift Right	`srl, srli`	`>>`	R/I
Shift Right Arithmetic	`sra, srai`	`>>` but uses sign bit (instead of 0)	R/I
Bitwise-And	`and, andi`	`&`	R/I
Bitwise-Or	`or, ori`	`|`	R/I
Bitwise-Xor	`xor, xori`	`^`	R/I

2.3.3 Arithmetic Operators

Operator	RISC-V Instruction	Effect	Type
Add	`add, addi`	`+`	R/I
Subtract	`sub, subi`	`-`	R/I

- Note that `subi = addi` where immediate operands are sign extended. (Should be a pseudoinstruction)
- Multiplication + Division added by extension M.

2.3.4 Branching Instructions

- **Conditional branches** are instructions that test a condition and that moves the `pc` accordingly.

Operator	RISC-V Instruction	C Translation
Equal	`beq rs1, rs2, o`	`if (rs1 == rs2) PC += o;`
Not Equal	`bne rs1, rs2, o`	`if (rs1 != rs2) PC += o;`
Less Than	`blt rs1, rs2, o`	`if (rs1 < rs2) PC += o;`
Greater Than or Equal	`bge rs1, rs2, o`	`if (rs1 >= rs2) PC += o;`
Less Than (Unsigned)	`bltu rs1, rs2, o`	`if ((unsigned)rs1 < (unsigned)rs2) PC += o;`
Greater Than or Equal (Unsigned)	`bgeu rs1, rs2, o`	`if ((unsigned)rs1 >= (unsigned)rs2) PC += o;`

- **Bounds check** shortcut: $0 \leq x < y$ w/ $y > 0$. If $x < 0$, then (unsigned) $x >$ `max_signed_positive_int`. Hence $x < y$ unsigned comparison suffices for the bounds check.
- Type: SB
- Loops are branches to the beginning of the block (denoted by a label) e.g.

```
main:                                   int j = 20;
    li t0, 20                           while(j-- > 2);
    li t1, 2
.L1:
    addi t0, t0, -1
    bgt t0, t1, .L1
```

- **Basic block** is a sequence of instructions without branches (except at the end).
- `switch-case` statements are implement using **branch address tables** (branch tables), a table of addresses of alternative instruction sequences.

- RISC-V loads the branch table and uses an indirect jump, which performs an unconditional branch to the address specified in the register. `jalr`.

2.3.5 Pseudo Instructions

Pseudo Instruction	Definition
Jump to Register	`jr rs1` $\triangleq$ `jalr x0, 0(rs1)`
Jump to Label	`call .L1` $\triangleq$ `jal ra, .L1`
Return	`ret` $\triangleq$ `jalr x0, 0(ra)`
No-op	`nop` $\triangleq$ `add x0, x0, x0`
Move	`mv rd, rs1` $\triangleq$ `addi rd, rs1, 0`
Branch if zero	`beqz rs1, .L1` $\triangleq$ `beq rs1, x0, .L1`
Branch greater than	`bgt rs1, rs2, .L1` $\triangleq$ `blt rs2, rs1, .L1`
Load immediate	`li rd, x` $\triangleq$ Determined by assembler. Based on width of `x`

2.4 Procedures and Calling Convention

- **Procedure** is a labelled sequence of instructions that requires **arguments**.
- **Caller**: invoking (or calling) procedure
- **Callee**: Procedure being called

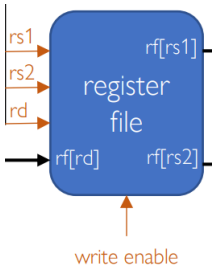
Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

- **Calling procedure**:
  1. Store arguments in `a0 - a7`
  2. Call procedure (`call`), storing return address in `ra`
  3. Allocate stack space (procedure *prologue*) to save registers
  4. Execute procedure
  5. Store result in `a0` (return register), deallocate stack space, restore callee saved registers.
  6. Return using `ret` w/ return address `ra`.
- **ABI**:
  - **Stack pointer**: A value storing the most recently allocated address on the stack. Used for storing registers when calling procedures.
  - **Leaf procedure** is a procedure that doesn't call any other procedures. Leaf procedures don't require stack space (unless they require more registers than are available).

- **Global pointer**: The register that is reserved to point to the `.data` section.

3 Processor Design

- Stages of FDE cycle:
  - **Fetch**: PC (program counter) is used to fetch the instruction.
  - **Decode**: Decode instruction depending on opcode and instruction format.
  - **Execute**: Depends on instruction class:
    - ALU used to calculate: Arithmetic/Logic result, Memory address for load/store, Branch comparison
    - Access memory for load/store.
    - Write results back to register file.
    - Branch / increment PC.
- **Register file**: A set of registers than can be read and written to by supplying a register number (5 bits in RISC-V).



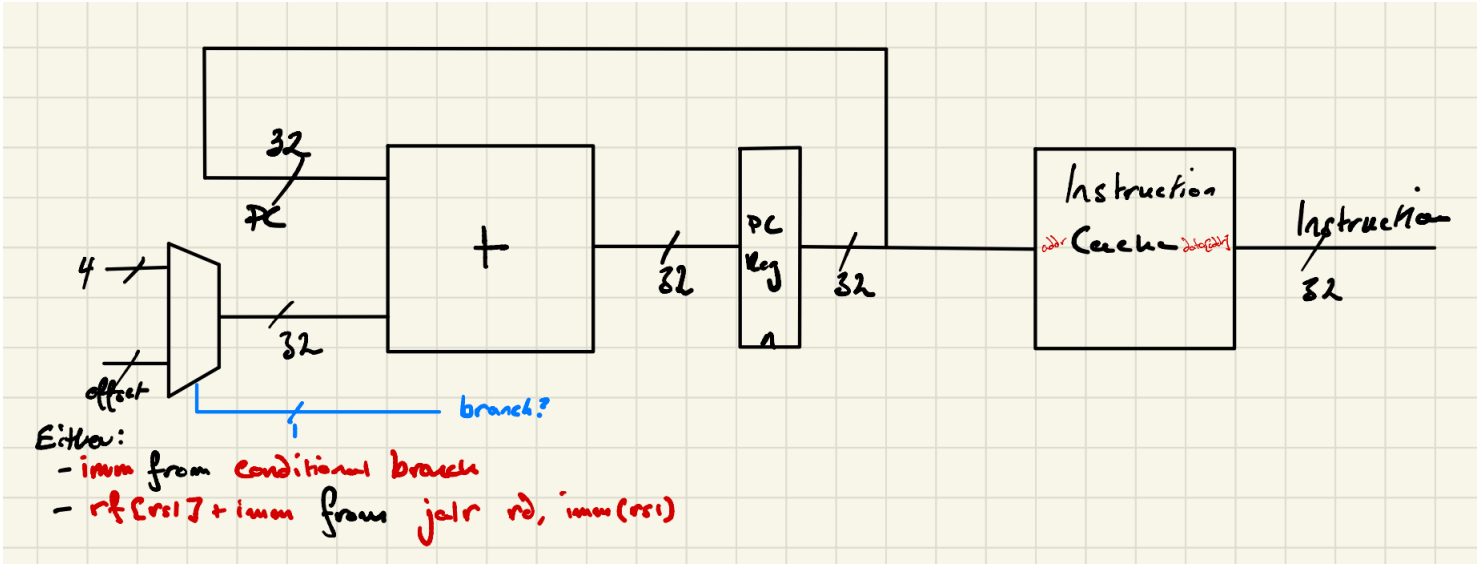
3.1 Datapaths

- **Datapath**: A datapath is collection of datapath elements connected with buses.

- **Datapath element**: A unit that is used to operate / hold data within a processor. e.g. ALU's, register files, etc.

3.1.1 Fetch Datapath

- Possibly ways of updating PC:
  - Increment PC to next instruction: `pc + 4`
  - Branch/Jump using an offset: `pc + offset`
  - Jump w/ offset: `rf[rs1] + offset`, for `jalr rd, imm(rs1)`.
- Following portion of the datapath for fetching instructions and incrementing the `pc`:

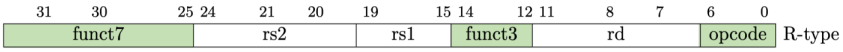


3.1.2 Arithmetic-Logic Datapath

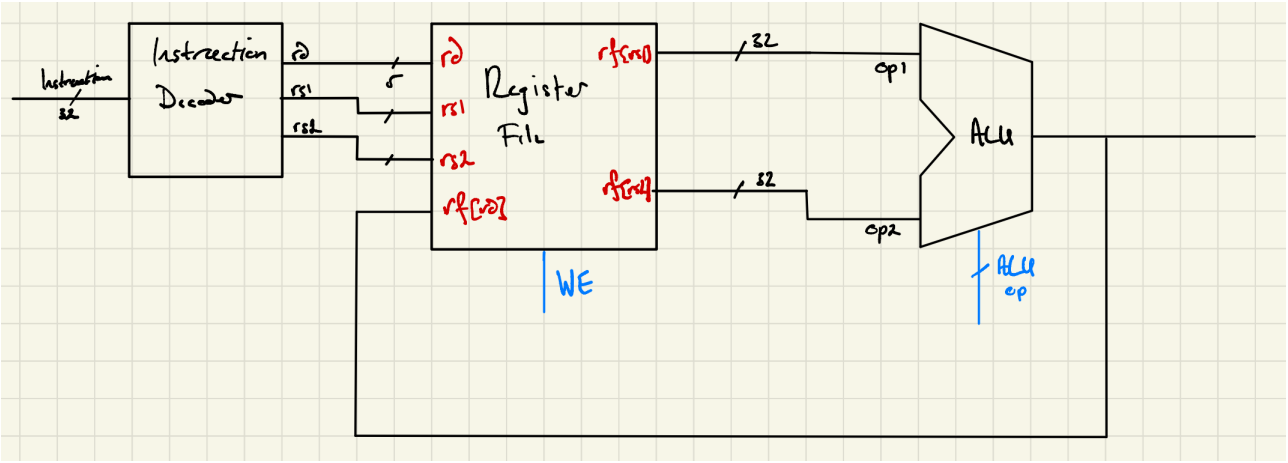
- Datapath section deals with arithmetic-logic instructions e.g. `add`, `ori`, etc.

3.1.2.1 Register-Register Instructions

- Several *R*-type arithmetic instructions. All operations read `rs1` and `rs2` and write back to `rd`. `funct3` and `funct7` is used to select the ALU operator.

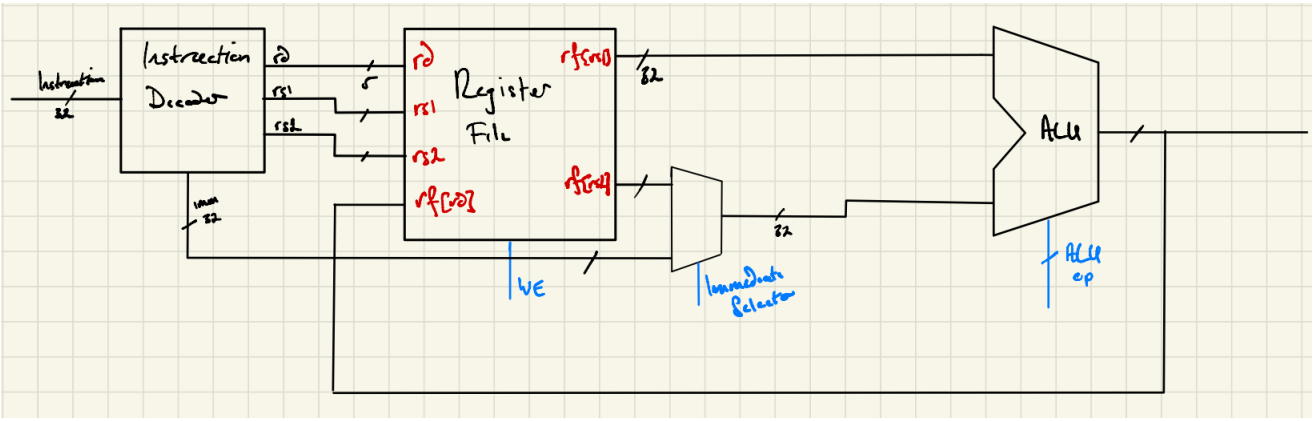


- Steps:
  - Decode register operands: `rd, rs1` and `rs2`.
  - Fetch register contents: `rf[rs1], rf[rs2]`.
  - Perform operation using ALU
  - Write result to `rd`. `rf[rd] = alu(rf[rs1], rf[rs2])`.



3.1.2.2 Register-Immediate Instructions

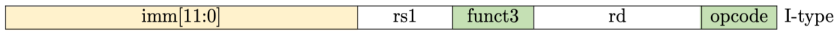
- All 12-bit immediates `imm[11:0]` are **sign-extended** to XLEN-bits.
- Steps:
  - Decode register operands: `rd, rs1` and immediate `imm[XLEN-1:0]`
  - Fetch register contents: `rf[rs1]`.
  - Perform operation using ALU.
  - Write result to `rd`.



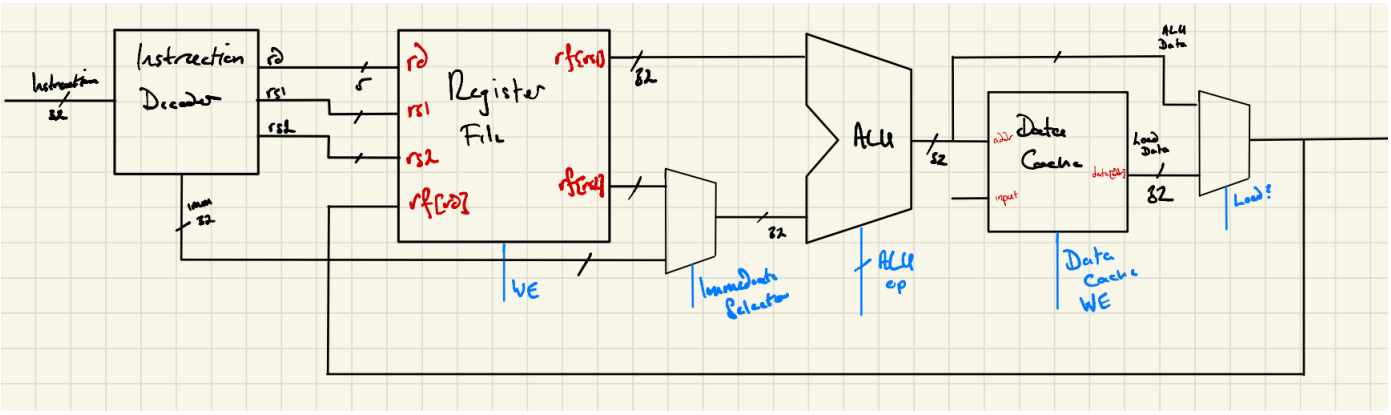
- The register-register and register-immediate datapaths are combined using **multiplexors** and **control-bits**.
- Control bits:
  - `WE`: Write enable bit for register file
  - `Immediate Selector`: Selects either `rf[rs2]` or `imm` using multiplexor.
  - `ALU Op`: operation bits for ALU, determined by instruction `opcode` (and `funct` bits).

3.1.3 Load/Store Datapath

- Loads:
  - Encoded using *I*-type format:



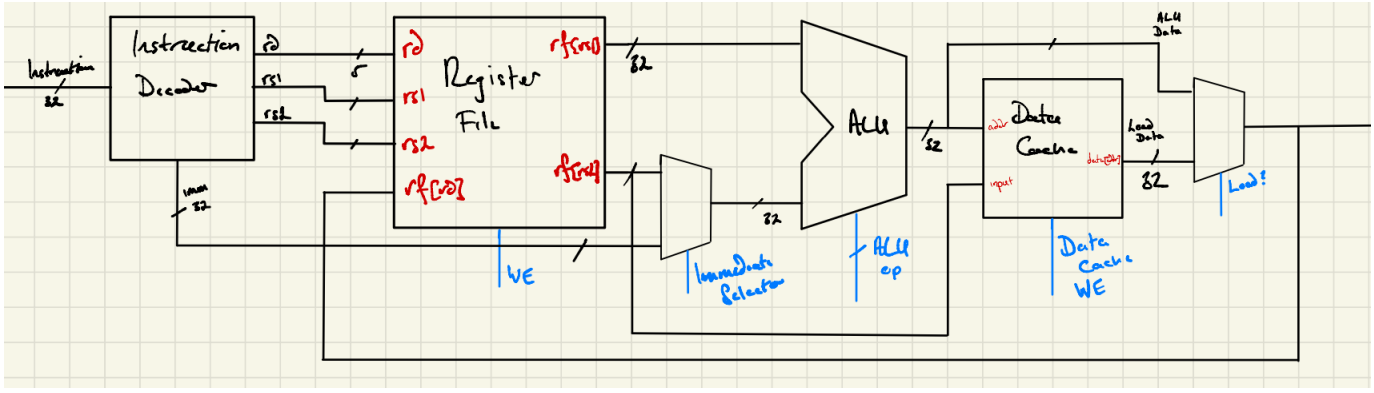
  - `funct3` specifies the width (`B, H, W` and `D` for byte, half-word, word and doubleword).
  - Steps:
    1. Decode register operands `rd, rs1` and immediate to `imm[XLEN-1:0]`
    2. Calculate address (using ALU): `addr = rf[rs1] + imm`
    3. Fetch contents of `addr` from data memory
    4. Store in `rd`.



- Stores:
  - Encoded using *S*-type format:

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type

  - `funct3` specifies the width (`B`, `H`, `W` and `D` for byte, half-word, word and doubleword).
  - Steps:
    1. Decode register operands `rs1`, `rs2` and immediate to `imm[XLEN-1:0]`
    2. Calculate address (using ALU): `addr = rf[rs1] + imm`
    3. Store `rf[rs2]` at `addr` in data memory



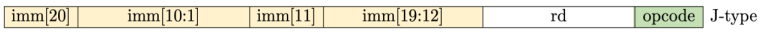
- Control bits:

- `Data Cache WE`: Write enable bit for data cache (used for store instruction)
- `Load`: True if current instruction is a load instruction.

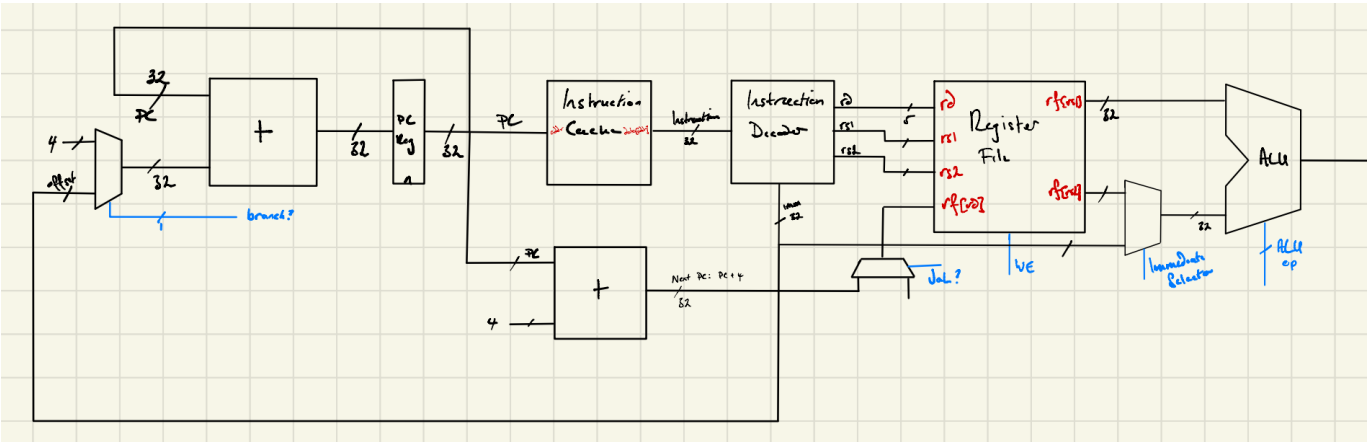
3.1.4 Branching Datapath

3.1.4.1 Unconditional Jumps

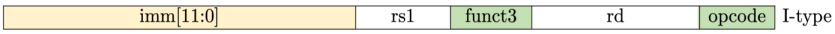
- Jump and link (`jal`):
  - Encoded using *J*-type format:



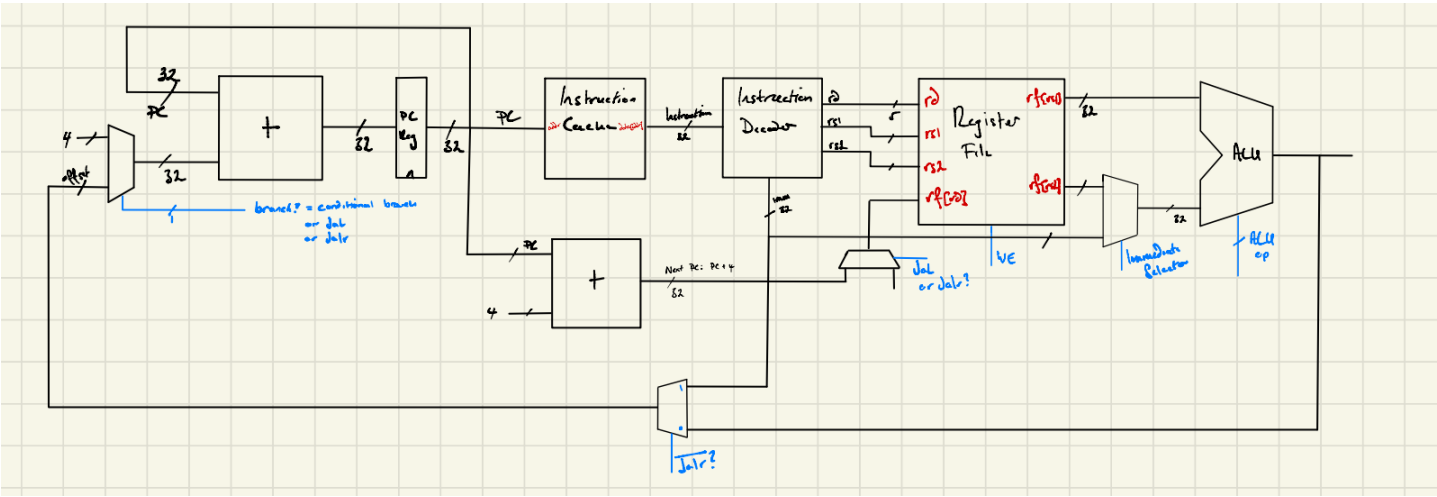
  - The immediate `imm[20:1]` is sign extended to `imm[XLEN-1:1]` and shifted left by `1` to form `offset[XLEN-1:0]`.
  - Steps:
    * Decode register operands `rd` and immediate `offset[XLEN-1:0]`
    * Calculate `pc + 4` and store to `rd`. `rf[rd] = pc + 4`
    * Calculate the jump target address `pc + offset` and jump.



- Jump and link register (`jalr`):
  - Encoded using *I*-type format:



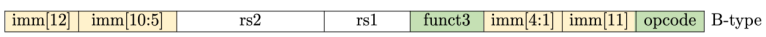
- 12-bit immediate is signed extended and shifted left to form offset `offset[XLEN-1:0]`.
- Steps:
  - Decode register operands `rd, rs1` and immediate `offset[XLEN-1:0]`.
  - Fetch register contents: `rf[rs1]`.
  - `rf[rd] = pc + 4`
  - Calculate the jump target address `rf[rs1] + offset` and jump.



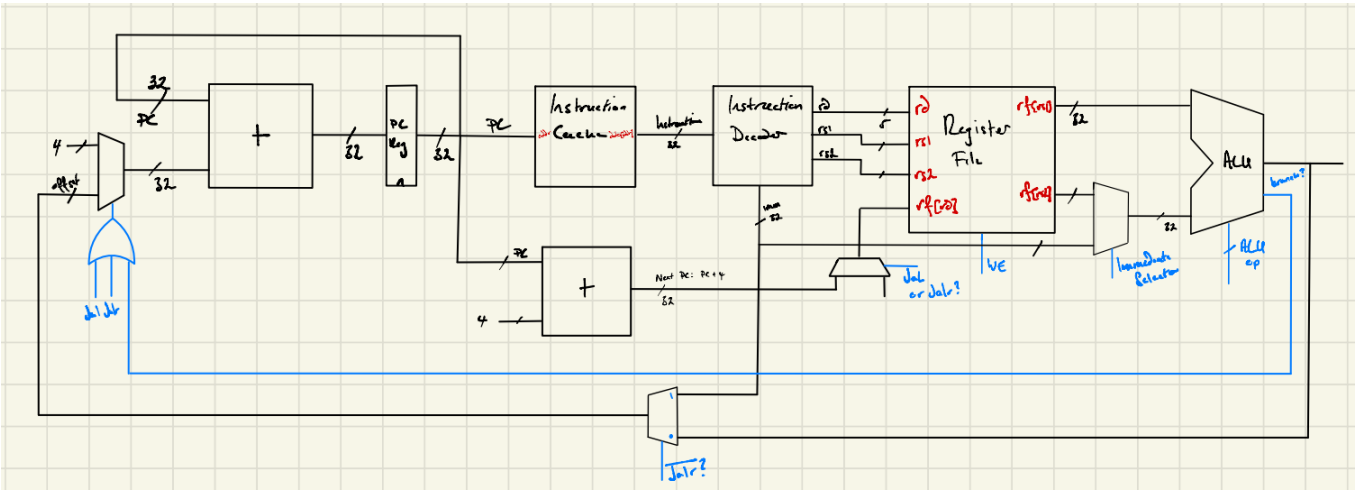
- Control bits:
  - `Jalr`: True if current instruction is jump and link register
  - `Jal`: True if current instruction is jump and link

3.1.4.2 Conditional Branches

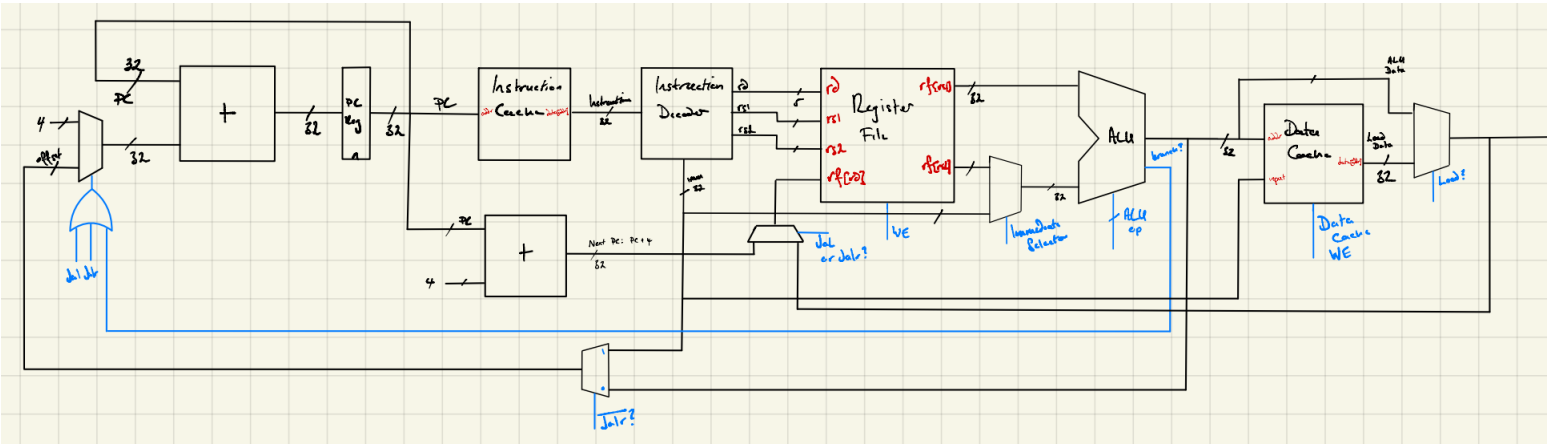
- All branch instructions encoded using *B*-type format:



- 12-bit immediate is sign extended and shifted left to form the branch offset `offset[XLEN-1:0]`
- Steps:
  - Decode register operands `rs1, rs2` and the immediate `offset[XLEN-1:0]`
  - Fetch register contents: `rf[rs1], rf[rs2]`
  - Determine branching condition `b` (using ALU)
  - If `b = 1`: Calculate jump target address `pc + offset` and jump.



- Control bits:
  - `branch`: True if current instruction is a conditional branch and ALU determines that branching condition `b` holds.
- Composing all the datapaths yields the following processor design:



3.2 Control Paths

- **Control Unit**: A component of a processor that directs the operation of the datapath.
- **Control Path**: The collection of control bits and multiplexors that modify the operation of the datapath.
- Control bits are usually determined by the instruction (using an instruction-decoder) and various outputs of datapath e.g. the ALU `zero` output.

3.3 Pipelining

- **Problem**: Critical path in datapath determines determines the clock period T.

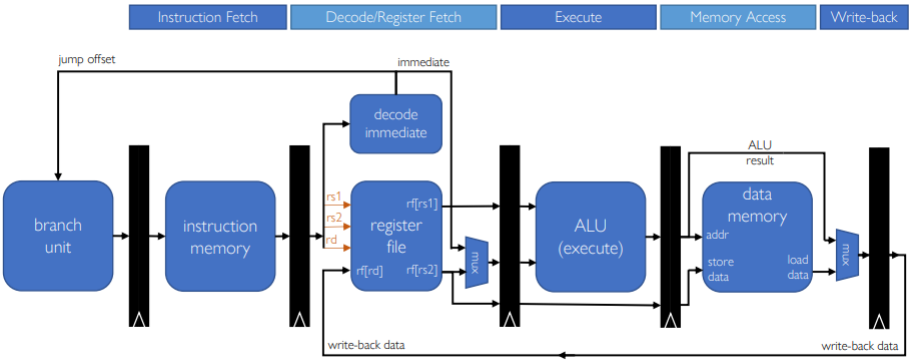
  Violates **common case fast** principle. (Really bad when floating point instructions introduced).

Definition 3.3.1. (**Pipelining**) Pipelining is an implementation technique where multiple instructions are overlapped in execution. The datapath is divided into k staged. Each stage executes in parallel.

- Doesn't decrease execution time of an instruction, but increases **throughput** (number of instructions per second).

$$\text{Throughput} = \frac{n}{(k+n-1)T} \qquad \text{Throughput} = \frac{1}{kT}.$$

- The time period T is determined by the **slowest stage**.
- **Theoretical Speedup**:
  - Consider k staged pipeline with a clock cycle of T. Consider n instructions to be executed. Assume no stalls occur and pipeline is balanced (all stages take the same time).
  - The first instruction takes kT time to execute.
  - The subsequent $n-1$ instructions only require an additional cycle to execute.
    $$T(k,n) = (k+n-1)T.$$
  - Speedup is
    $$\begin{aligned} S &= \frac{\text{Execution time (non-pipelined)}}{\text{Execution time (pipelined)}} \\ &= \frac{nkT}{(k+n-1)T} = \lim_{n\to\infty} \frac{nkT}{nT} = k \end{aligned}$$
- The CPI of ideal pipelined processor is 1.
- **Implementation**: Control and instruction data are stored in **interface registers** m-bit registers storing the immediate output between two stages.



3.3.1 Hazards

Definition 3.3.2. (**Hazard**) A hazard is any dependency that prevents the pipeline starting the next instruction in the next cycle.

- These hazards introduce **stalls**, or bubbles, cycles where the pipeline has no-operation (a `nop` instruction).
- Control logic determines **stalls**: ensures next instruction will not be fetched and suspends instruction that causes hazard.
- **Structural Hazard**:
  - A resource conflict: one-or-more instructions try to access same resource in the same cycle.

    e.g. If we had a single instruction and data memory:

	IF	ID	EX	MEM	WB
I_1	IF	ID	EX	MEM	WB
I_2		IF	ID	EX	MEM
I_3			IF	ID	EX
I_4				IF	ID

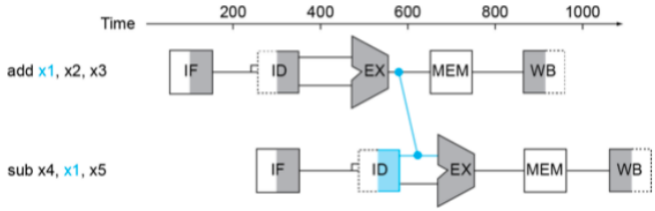
  - **Solution**: Introduce redudant resources via hardware renaming or stall.
- **Data Hazard**:
  - Occurs when an instruction depends on state that has yet to be updated by previous instructions (still executing) (since WB stage is the last stage, while register-file is read in ID stage).
  - e.g.

```
I_1 | add x1, x1, 1
I_2 | add x1, x1, (-1)
```

    By the sequential model: `x1` isn't changed by the above (no effect). However with pipelining I_2 reads `x1` before I_1 writes back. Resulting in `x1 = x1 - 1`.

- Define $R(I)$ to be addresses read by instruction I, and $W(I)$ to be addresses written to by instruction I. Three dependencies (in general):
  * Read-after-write (RAW): $W(I_1) \cap R(I_2) \neq \emptyset$. I_1 writes after I_2 reads.
  * Write-after-read (WAR): $R(I_1) \cap W(I_2) \neq \emptyset$. I_1 reads before I_2 writes.
  * Write-after-write (WAW): $W(I_1) \cap W(I_2) \neq \emptyset$. I_1 and I_2 write to the same address.

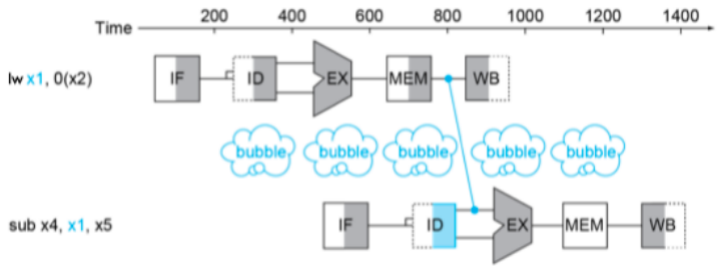
  WAR and WAW occur for *out of order* processors. RAW affects *pipelined* processors.
- **Solutions**:
  1. Stall
  2. **Operand Forwarding**: Interface registers forward output from instruction I_i at stage k to input interface register of instruction I_{i+1} at stage $k-1$ $\implies$ Forwarding back $n+1$ stages results in a stall for n cycles.
     Implemented using some control logic and multiplexors.



  3. **Internal forwarding**: if `rf[n]` is being written to an read in the same cycle, the value to be written is passed through.
- Some situations cannot be fixed by **operand forwarding**:

```
ld t1, 0(t0)
add t1, t1, 1
```

  This is a **load-use** data hazard. Requires a stall for 1 cycle.



Compilers attempt to prevent this by **load-hoisting**

- **Control Hazard**:
  - Occurs due to a conditional branch or jump instruction, a modification in the program's flow-control.
  - e.g.

```
        I_1 | ...
        I_2 | bgt t0, t1, .L1
        I_3 | ...
        I_4 | ...
        ...
.L1:
        I_b | ...
```

I_1	ID	IF	EX	MEM	WB	
I_2		ID	IF	EX	MEM	WB
I_3			ID	IF	EX	MEM
I_4				ID	IF	EX
I_b					ID	IF

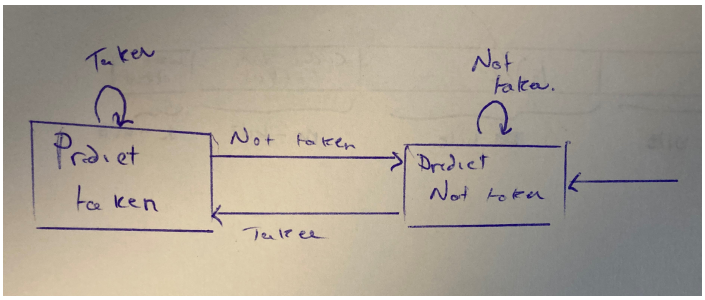
Problem: Breaking the *sequential model* since I_3 and I_4 are executing.

  - **Solutions**:
    1. **Stall** on branch for two cycles. Then restart with correct `pc`.

2. **Branch Prediction**: If we incorrectly predict the branch address of I_i, then I_{i+1} and I_{i+2} are converted into "bubbles" (`nop`s); and undo any state changes produced by I_{i+1}, I_{i+2} (**flushing** the pipeline), and restart the pipeline with the correct `pc`.
3. **Evaluate branching address earlier**: e.g. add additional hardware to ID stage $\implies$ reduces number of stalls required. *However*, adds complexities e.g. forwarding source operands from EX/MEM or MEM/WB interface registers. Thus potential data hazards.
4. **Branch Hoisting**: Place instructions that are independent to the branch after the branch instruction. (If no independent instructions, place `nop`s). Prevents flushing / bubbles.

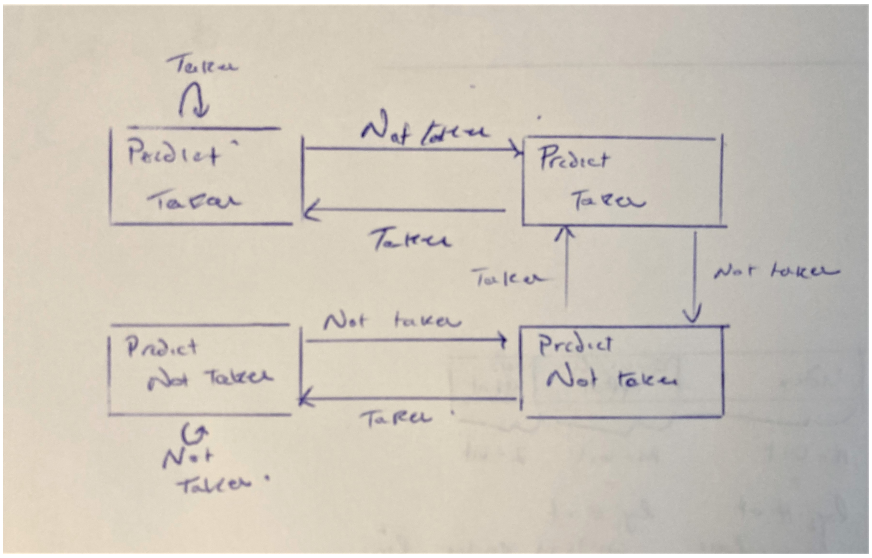
3.3.2 Branch Prediction

- **Static Branch Prediction**: Predict behavior of branch based on instruction type
  - Assume branch is always take
  - Forward branches aren't taken (if statements), backward branches are take (while loops)
- **Dynamic Branch Prediction**:
  - Additional hardware to predict branches *at runtime* based on observed behavior.
  - **Branch prediction table**: A table indexed by a lower portion of branch instruction addresses. Stores whether branch was previously taken or not.
  - **1-bit Predictor**:



Problem: If branch is almost always taken (e.g. a loop), predictor will predict incorrectly twice (at start and end).

– **2-bit Predictor**:



(Note, optimal for initial state to be a weak predictor)

3.3.3 Exceptions and Interrupts

Definition 3.3.3. (**Exceptions**) **Exceptions** are defined as unexpected events that occur during the sequential execution of instructions (e.g. a page fault, or sys-call (`ecall`))

- Exceptions are said to be *synchronous*.

- Exceptions produce **control hazards**: Jump to the *trap handler* immediately after exception occurs and can occur at any stage within the pipeline.

  e.g. page-fault occurs in MEM stage $\implies$ stall the pipeline (and flush), then execute the trap handler stored in the `stvec` register.

- **Registers**:
  - `stvec`: Supervisor trap vector. Either the address of the handler, or an address of an array of handlers indexed by `scause` (dependent on settings).
  - `sepc`: `PC` of the trapping instruction
  - `scause`: Cause of the trap. (See Tiny-OS).

Definition 3.3.4. (**Interrupts**) **Interrupts** are defined as unexpected events that are caused by external inputs e.g. timers, PLICs, etc. They occur *asynchronously*.

- Interrupts don't produce control hazards, since we can change between executing the instructions and the trap handler without any side effects after all currently executing instructions are executed (pipeline is *empty*).

  Provided the trap handler correctly restores the state of the registers.

3.3.4 Instruction-Level Parallelism

- **Instruction-Level Parallelism**: Executing multiple instructions in parallel.
- **Methods**:
  - Pipelining: Increase number of stages in pipelines. Increases # instructions executing at once. Decreases clock period $\implies$ increases speedup.

    More forwarding and hazards.
  - Multiple pipelines: Duplicate pipeline stages. Executing multiple instructions per clock cycle (true parallelism). Known as *superscalar or multiple issue* pipelines.

3.3.4.1 Multiple Issue Pipelines

- **Problem**: Issuing determined either statically or dynamically.
- **Static Multiple Issue**:
  - Compiler groups instructions into *issue slots*
  - Compiler ensures no hazards
  - Requires processor implementation to be exposed to compiler.
- **Dynamic Multiple Issue**:
  - Processors analyzes instructions (via a scoreboard). Dispatching issue groups w/out hazards.
  - **Very complex**. Requires effect buffers, schedulers and flushing logic.

4 Memory

4.1 Memory Technologies

4.1.1 Volatile

- **Static Random Access Memory** (SRAM):
  - A type of random access memory (RAM) that uses flip-flops to store each bit.
  - Fast, consistent read and write times. (1-10 clock cycles)
  - Energy Efficient. No refresh required (unlike DRAM).
  - Expensive. Requires more silicon (6-8 transistors) per bit.
- **Dynamic Random Access Memory**:
  - A type of random access memory (RAM) that uses capacitor transistor pairs.
  - A charged capacitor is `1` (vice-versa). The transistor is used to select whether the capacitor is charging or discharging.
  - **Problem**: Capacitors leak charge $\implies$ must be refreshed (refresh cycle). Data cannot be read/written while refreshing.
  - DRAM with clocks, synchronous DRAM (SDRAMs) allow successive burst of bits to be read. DDR (Double Data Rate) transfers data on both rising and falling edge of the clock.
  - Fast, but slower than SRAM. (100 clock cycles). Cheaper.

4.1.2 Non-Volatile

- **Flash Memory / SSD**:

  - Uses programmable ROM chips (EEPROM). e.g. Floating gate transistor:
    * Controlled by 2 gates: **control gate** and **floating gate** isolated between high resistive material (isolator).
    * The two gates are sandwiched between the isolator, trapping the charge in the floating gate.
  - Controller splits memory into blocks (prevents memory bit wear out, *wear leveling*).
  - Block-based device: data can only be read / written to in blocks. Requires error checking (e.g. hamming codes)
  - Faster than disk drive. (20 μs) High bandwidth (16 GB/s). Energy Efficient. Expensive.

- **Hard Disk Drive (HDD)**:
  - Consists of a collection of magnetic platters (disks in a *cylinder*) spinning at $\sim$ 7500 RPM.
  - Data is stored on the surfaces - change in direction of magnetic domains (polarity) represents a bit.
  - Disk split into tracks (concentric circles) and sectors (segments of a track).
  - Actuator arm moves *read-write head* across the surface of the disk, addressing *tracks*, waiting for the *sector* to spin around (*rotational latency*).

4.2 Locality and Memory Hierarchies

Definition 4.2.1. (**Temporal Locality**) Temporal locality refers to the reuse of an item x over a small time duration.

- If x is referenced, then it is likely to be referenced again soon.

Definition 4.2.2. (**Spatial Locality**) Temporal locality refers to the use of items $x_1, \ldots, x_n$ s.t $x_1, \ldots, x_n$ are "close" over a small time duration.

- If x_i is referenced, then the items $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ whose addresses are relatively close to x_i are likely to be referenced soon.

- Instructions and data both exhibit temporal and spatial locality.

Definition 4.2.3. (**Memory Hierarchy**) A structure with multiple *levels* of memories, as the distance from CPU increases, the size and access time of memory increases.

- Hierarchy consists of ℓ levels, only need to consider level i and $i+1$ since the design is simply repeated.

4.3 Caches

Definition 4.3.1. (**Cache**) A cache is small amount of low-latency (SRAM) memory that stores recently used data.

- Used to alleviate the **Von-Neumann bottleneck**. Provide abstraction of low-latency large addressable memory.
- Organized in several layers: $L1$, $L2$, etc.

Definition 4.3.2. (**Cache Line**) A cache line the minimum unit of data stored in the cache. Often 4 contiguous words.

4.3.1 Cache Operations

4.3.1.1 Cache Misses, Hits and Reads

- When a processor access data, it checks the corresponding cache entry:
  - If present in the cache, a **cache hit** occurs (and data is read from cache).
  - If not present, a **cache miss** occurs. Cache allocates a new line and copies in data from level $i+1$, then forwards data to processor.
- **Hit Rate**:

$$\text{Hit Rate} = \frac{\text{Number of Hits}}{\text{Number of Hits} + \text{Number of Misses}} = \frac{n_h}{n}.$$

Miss rate:

$$\text{Miss Rate} = \frac{\text{Number of Misses}}{\text{Number of Hits} + \text{Number of Misses}} = \frac{n_m}{n} = 1 - \text{Hit Rate},$$

where n is the number of total accesses.

- **Miss penalty**: Time taken to fetch data from $i+1$, then allocate and store cache line in i, and pass the data back to the requestor.
- Expected access time:

$$\mathbb{E}[T] = \text{Time for hit} + \text{Miss Rate} \times \text{Miss Penalty}.$$

- **Miss Sources**:
  - **Compulsory miss**: First access requires a miss to fetch line from level $i+1$.
  - **Capacity miss**: A line that is replaced (evicted) is later accessed again
  - **Conflict/Collision miss**: A miss due to a address to line mapping collision.
- **Pollution**: Data in cache that may never be read.

4.3.1.2 Writes

- **Problem**: On data-write hit, we update the cache line $\implies$ level i and $i+1$ are now inconsistent.
- **Solutions**:
  - **Write through**: writes always update both level i and $i+1$, ensuring consistency.
    Problem: Not efficient. If base $CPI = 1$ and 10% `s` instructions and write to level $i+1$ takes $CPI = 100$, then $\mathbb{E}[CPI] = 1 + 0.1 \times 100 = 11$.
    Solution: A **write buffer**, a FIFO queue that stores data waiting to be written to level $i+1$. Processor doesn't wait for level $i+1$ (unless the write buffer is full, then stall). Stalls can still occur:
    - If rate of store instructions $>$ rate that level $i+1$ writes complete.
    - Burst of writes $\implies$ buffer fills quickly.
  - **Write back**: Write to level i and set the cache line's *dirty bit*. When the cache line is evicted and if the dirty bit is set, then write line back to level $i+1$. Improved performance over write through. Complex implementation.

4.3.1.3 Cache Misses

- **Solution**:

  1. On cache miss, stall and forward A to memory level $i+1$ (main-memory).
  2. Send `READ` request to memory level $i+1$ and wait for response.
  3. Write response to cache line (determined by A), writing the tag field and setting the valid bit.
  4. Restart processor and re-fetch (this will ensure a cache-hit).

  (For instruction-fetch, simply restart instruction execution).

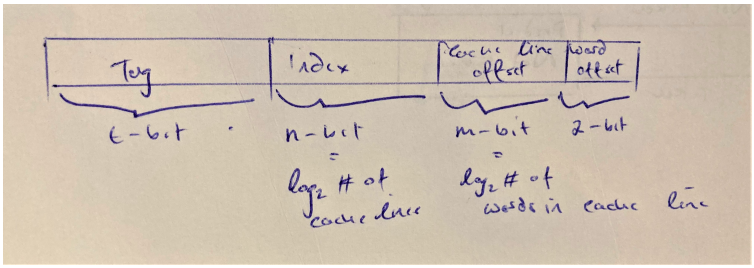
- **Write misses**:

  - **Write-through**:
    - Allocate a cache-line on miss then write
    - **Write-around**: Write data directly to level $i+1$. Improves performance.
  - **Write-back**:
    - Allocate a cache-line on miss.
    - Perform write and set dirty bit.

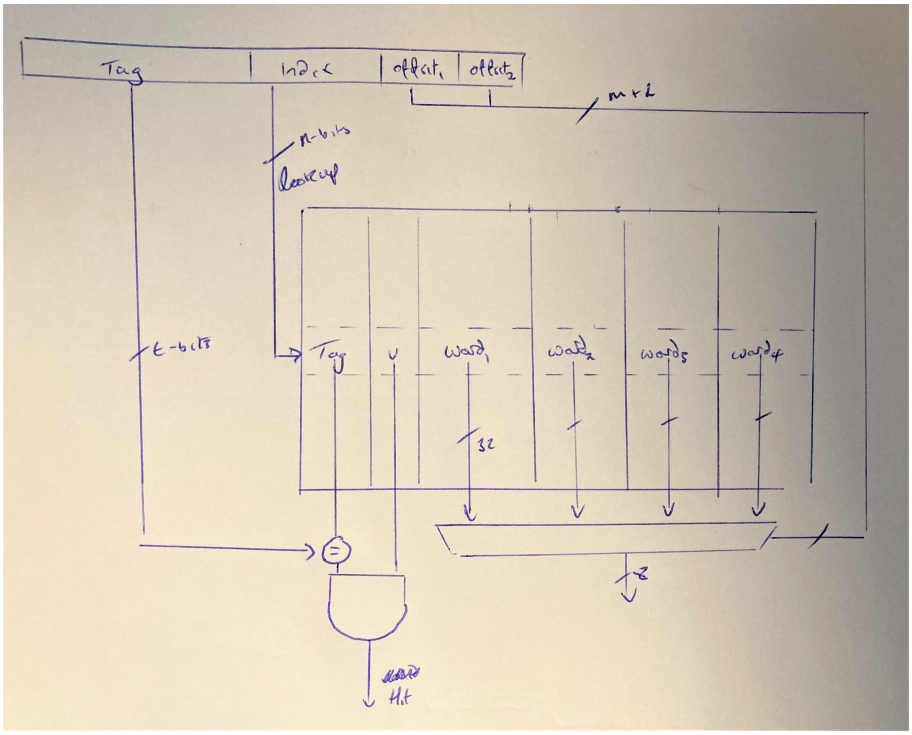
4.3.2 Direct Mapping

Definition 4.3.3. (**Direct Mapping Cache**) A cache structure in which any block of memory with address A from level $i+1$ can be placed into a *unique* cache line.

- Address A is split into a tag (t-bits), a cache line index (n-bits), a cache-line offset (m-bits) and a word offset (2-bits).



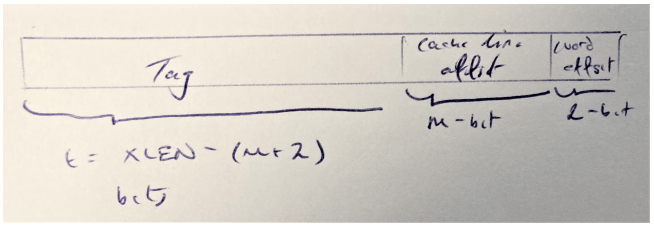
- Note that $t = XLEN - (n + m + 2)$.
- The tag and cache line index *uniquely identify* the addresses of the words stored in the cache line.
- A **valid bit** is used in the cache to indicate that the cache line contains valid data.



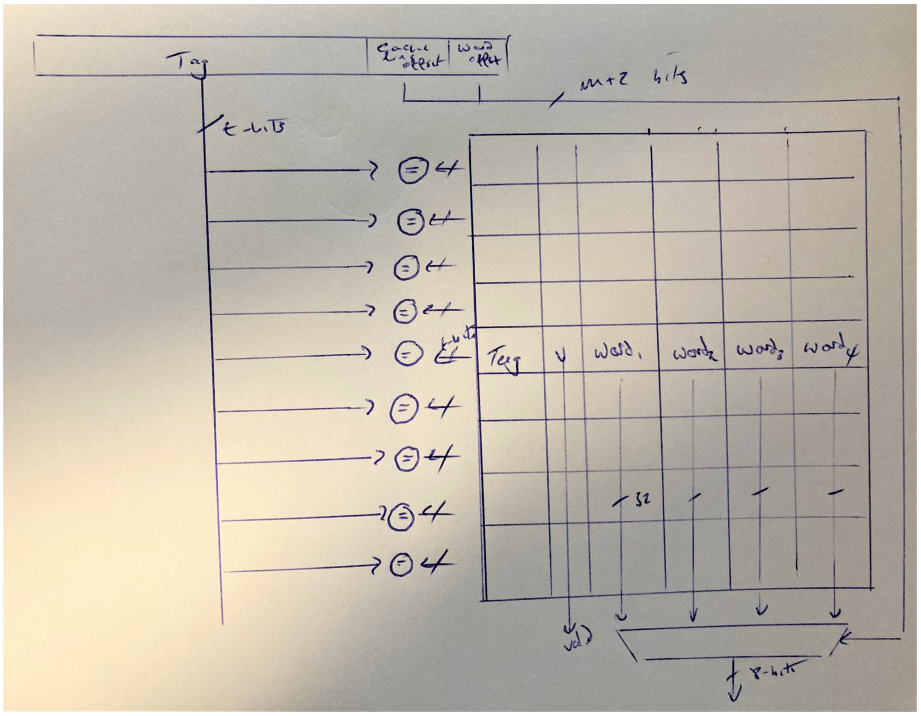
4.3.3 Fully Associative

Definition 4.3.4. (**Fully Associative Cache**) A cache structure in which any block of memory with address A from level $i+1$ can be placed into *any* cache line.

- Address A is split into a tag, a cache-line offset and a word offset.



- Requires all cache-lines to be searched using comparators associated with each line. Exploits hardware-level **parallelism**

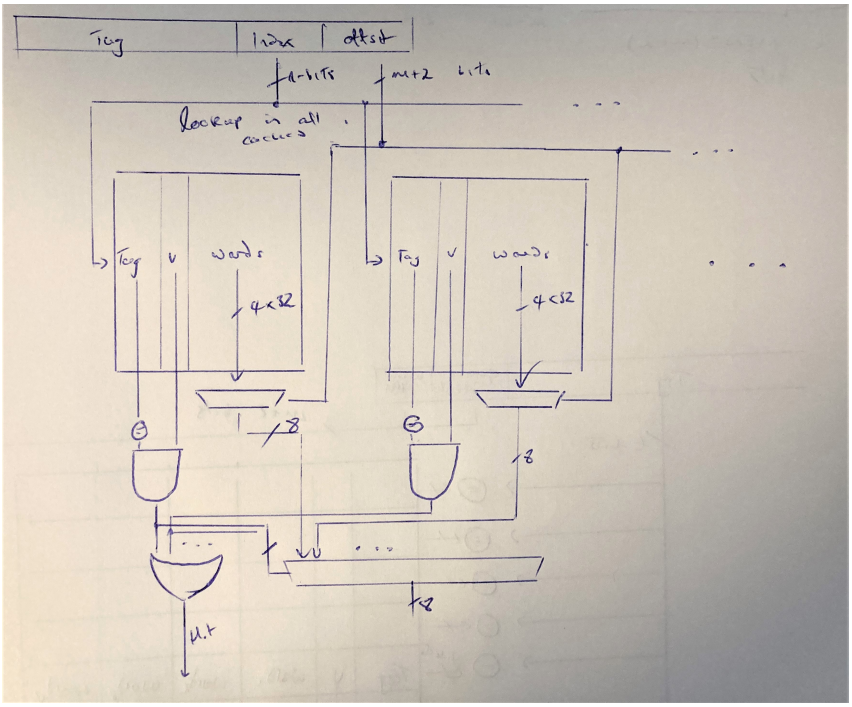


- Increased hit rate, but more expensive (due to comparators). Good for *small caches*.

4.3.4 Set Associative

Definition 4.3.5. (**Set Associative Cache**) A n-way set associative cache is a cache with a structure consisting of n direct-mapped caches, the *sets*, where an address A from level $i+1$ could be mapped to *any* of the sets.

- Address A is split into a tag (t-bits), a cache line index (n-bits) for each set, a cache-line offset (m-bits) and a word offset (2-bits).
- Tag field is searched in each set using comparators associated with the set. (**parallelism**)



- A fully associative cache w/ m lines is a m-way set associative cache with direct-mapped caches containing 1 line.

	Set	Lines / Set	Lookup	Comparators
Direct Mapping	1	# lines	Index lines via cache line index part of address	1
Set Associative	n	# lines	Index all sets in parallel. Compare results w/ multiplexor.	n
Fully Associative	# lines	1	Index all lines (sets) in parallel. Or use a lookup table	# lines

- **Implementation**:
  - **ASIC**: Direct mapped is more efficient (uses less silicon) than fully associative.
  - **FPGA**: Directed mapped is implemented efficiently using block RAM/SRAM.

    Fully associative requires additional logic blocks.
- **Hit Rate**: Fully associative has the highest hit rate
- **Trade Offs**:

	Advantage	Disadvantage
Cache size	Decreases capacity misses.	Increases access time
Associativity	Decreases collision misses	Increases access time
Cache line size	Decreases compulsory misses	Increases miss penalty (more data requested from level $i + 1$). Increased collision (fewer lines). Increased pollution.

4.3.5 Cache-line Eviction

- **Direct Mapped**: Evict block occupying cache-line since the requested block can only go in a unique line.

- **Set Associative**:
  - Choose an entry among the n sets (if no free entries).
  - **Least recently used** (LRU): The evicted block is the one that has been unused (unreferenced) for the longest time. Requires counters c for each line.
  - **Not last used**: Approximates LRU using a simpler implementation.

    Requires not-last-used bit for each line. Bit is set on line reference. Unset, when a different line is referenced.
- **Fully Associative**: If cache is full, randomly choose a block to evict from a cache-line with uniform probability.
- **Problem**: Potential "pathological" evictions with direct-mapped cache

```
int a, b; // assume both have same cache index but different tags
while (true) {
    a++;
    b++; // evicts a
}
```

- **Solution**: Augment direct-mapped cache with a tiny fully associative cache (buffer), the **victim cache**. Evicted cache-lines stored in victim cache.

4.3.6 Performance

We have:

$$\text{CPU Time} = (\text{CPU Cycles} + \text{Cache-stall cycles}) \times T$$
$$\text{Cache-stall cycles} = \text{Read-stall cycles} + \text{Write-stall cycles}$$
$$\text{Read-stall cycles} = n_R \times \text{Read Miss Rate} \times \text{Read Miss Penalty}$$
$$\text{Write-stall cycles} = n_W \times \text{Write Miss Rate} \times \text{Write Miss Penalty} + \text{Write-buffer stalls}$$

Typically Write-buffer stalls ≈ 0. So we have

$$\text{Read-stall cycles} = n_R \times \text{Read Miss Rate} \times \text{Read Miss Penalty}$$
$$\text{Write-stall cycles} = n_W \times \text{Write Miss Rate} \times \text{Write Miss Penalty}$$

If we combine the read-write miss rates and penalty:

$$\text{Cache-stall cycles} = n_{Access} \times \text{Miss Rate} \times \text{Miss Penalty}.$$

5 OS Support

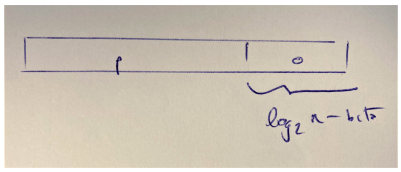
5.1 Virtual Memory

- **Motivation**:
  - Process isolation w/ virtual address spaces.
  - Allows main memory to act as a *cache* for secondary storage.

Definition 5.1.1. (**MMU**) The memory management unit (MMU) is additional hardware used to translate virtual addresses to physical addresses.

5.1.1 Paging

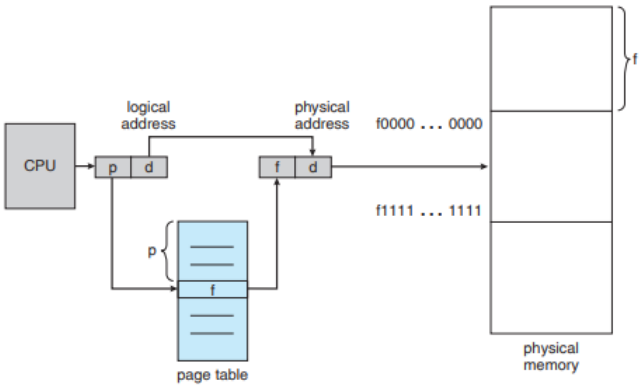
- **Idea**: The physical address space is split into *frames* or physical pages of n bytes. The virtual address space is split into *pages* or virtual pages of n bytes.
- The virtual address is split into a virtual page number (VPN) p and a page offset o, denoted (p, o)



Definition 5.1.2. (**Page Table**) A page table is an array of page table entries (PTEs) indexed by the virtual page number p of a virtual address (p, o).

Definition 5.1.3. (**Page Table Entry**) A page table entry consists of a physical page number (base frame address), protection bits `rwx`, valid, dirty, referenced bits, etc.

- The processor (or MMU) stores the address of the current page table in a register (e.g. `satp`).
- **Paging** translation:



1. Process loads virtual pages into physical pages on initialization.
2. Index current page table using VPN p, which maps to `pte`.
3. If `pte` is valid, then:
   (a) Obtain PPN f from `pte`.
   (b) Calculate the physical address $f + o$ using MMU.
4. If `pte` is invalid $\implies$ page not in main memory. Raise a *page fault exception*.

Definition 5.1.4. (**Page Fault**) A page fault is an *exception* that occurs if the `pte` for the virtual address (p, o) is *invalid* $\implies$ physical page is not in main memory.

- Page faults indicate:
  - A trapped address e.g. `null` pointers / guard pages.
  - The page is on disk.
- **Fault Handler**:

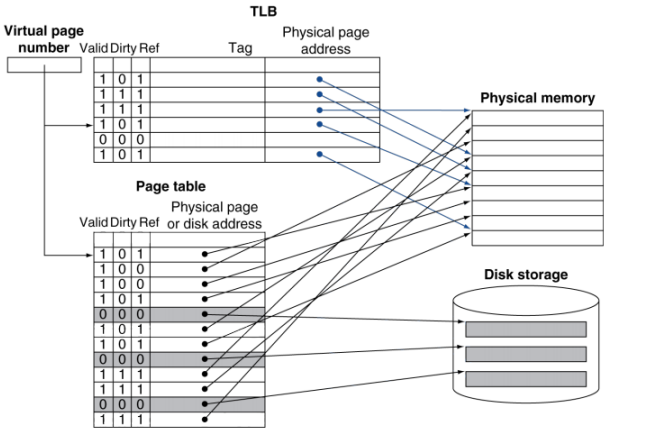
  - Locate page on disk (using virtual address). Stored in the `stval` register.
  - Free a virtual page yielding a free physical page. Allocate the physical page to the trapped virtual page. Read contexts into physical page. Update `pte`.
  - Freeing a virtual page:
    * If dirt bit set $\implies$ write back
    * Schemes: LRU, Not-Last-Used, etc.

- **Fault penalty**: Time taken for fault handler to execute.
- **Problem**: Page faults require millions of cycles (!!) to handle.
- **Solution**. Minimize fault rate:
  - Fully associative placement (implemented by page table).
  - LRU replacement.

5.1.1.1 TLB

- **Problem**: Each memory access requires page table lookup and address calculation.
- Page tables exhibit spatial and temporal locality $\implies$ caching is effective.

Definition 5.1.5. (**TLB**) A TLB (translation lookaside buffer) is a fully associative cache of page table entries associated with virtual page numbers.

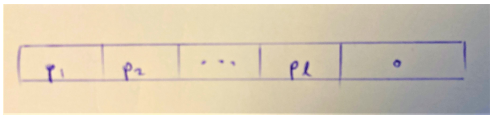


- **TLB Miss**:
  1. Lookup `pte` in page table
  2. If `pte` is valid: allocate TLB line, read to TLB, restart instruction.
  3. If `pte` is not valid $\implies$ raise page fault.
- TLB lookups are performed in hardware by the MMU.

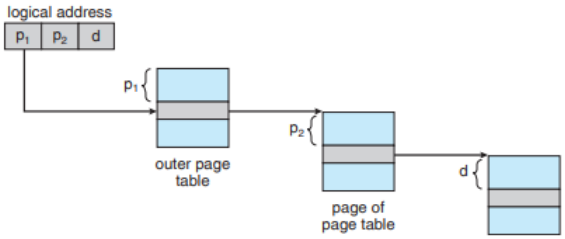
5.1.1.2 Multi-level Paging

- **Problem**: Large virtual address spaces $\implies$ excessively large page tables
- **Solution**: Multi-level (ℓ-level) paging.

  Split the virtual address into ℓ virtual page numbers and an offset: $(p_1, \ldots, p_\ell, o)$.



- **Paging** translation:




```
pte_t* walk(pagetable_t table, vaddr_t vaddr) {

    for (int level = ℓ; level > 0; level--) {
        pte_t pte = table[VPN(vaddr, level)];
        if ((pte & PTE_VALID) == 0) return NULL;
        table = (pagetable_t)PTE_TO_PADDR(pte);
    }

    return (pte_t)&table[VPN(vaddr, 0)];
}
```

- The page table walker (traverse) is implemented in hardware by the MMU.
- **Problem**: n-bit word length w/ page size of p bytes and pte size of m bits:

$$\text{overhead} = \frac{2^n}{p} \cdot \frac{m}{8} \text{ bytes} .$$

  (Upto 30 million GB).
- **Solutions**:
  - Superpages: A superpage at level i is a page of size:

$$\text{superpage}_i \text{ size} = \left(\frac{8p}{m}\right)^{\ell - i} \times p.$$

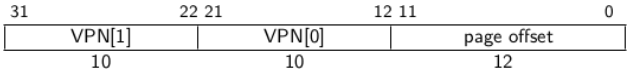
    Indicated by `R = 1` or `X = 1` (`RWX` $\neq$ `0`) protection bits for a non-leaf page table entry $\implies$ super page.
  - Inverted page tables.

5.2 RISC-V

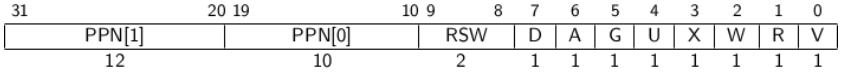
5.2.1 RISC-V Sv32 Addressing

Definition 5.2.1. (**Sv32**) RISC-V Sv32 is a 32-bit paged virtual memory addressing scheme.

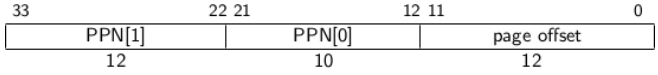
- Virtual Address:



- PTE:



- Physical Address:



- Page size $p = 4$ kb $= 4096$ bytes.

5.2.1.1 Protection

- Protection bits:

RWX bits	Description
000	Pointer to next level
100	Read-only page
010	Reserved for future
110	Read-Write page
001	Execute-only page
101	Read-Execute page
011	Reserved for future
111	Read-Write-Execute page

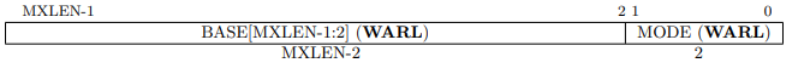
- **Problem**: TLB contains cached `pte`s from other processes $\implies$ invalid TLB hits.

  Solution:

  - Flush TLB: Inefficient, since next scheduled process suffers from compulsory misses.
  - ASID: `satp` contains a 9-bit ASID (Address Space Identifier). TLB entires store ASID. Hit if valid and ASIDs are equal.

5.2.2 Interrupts and Exceptions

- RISC-V operating modes:
  - User (U): No access to control status registers (CSRs).
  - Supervisor (S): Access to S-mode CSRs e.g. `stvec`, `stap`, etc. Restricted views of M-mode CSRs.
  - Machine (M): Access to M-mode CSRs e.g. `mhartid`, `mtvec`, etc.
- All interrupts and exceptions (traps) are handled by M-mode trap handlers (initially). traps may be delegated to low-level trap handlers via `mideleg` `medeleg` CSRs (interrupt and exception delegation registers).
- Trap handling CSRs:
  - `mtvec` register: Consists of a base address for the trap vector and a trap vector mode:

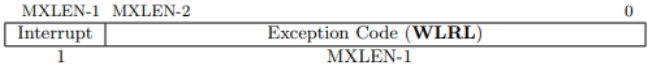


Modes:

  * Direct: Trap handler address is given by `base`
  * Vectored: Trap handler address is given by `base + cause * 4`

– `mepc` register: Machine exception program counter. Contains virtual (or physical) address of instruction that raised exception.

– `mcause` register: Stores the identifier for the event that caused the trap.



Interrupt bit is set if trap is an interrupt (asynchronous) or an exception (synchronous).

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	*Reserved for future standard use*
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	*Reserved for future standard use*
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	*Reserved for future standard use*
1	11	Machine external interrupt
1	12–15	*Reserved for future standard use*
1	≥16	*Reserved for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved for future standard use*
0	15	Store/AMO page fault
0	16–23	*Reserved for future standard use*
0	24–31	*Reserved for custom use*
0	32–47	*Reserved for future standard use*
0	48–63	*Reserved for custom use*
0	≥64	*Reserved for future standard use*

- `ecall`: The `ecall` instruction raises an environment call exception. This is handled by a trap handler in a more privileged mode (e.g. S/M)
- `mret/sret`: The `mret/sret` instruction returns from a trap in M/S-mode. The program counter `pc = mepc / sepc` is restored on return.

6 SoCs and DRAM

6.1 System On Chips

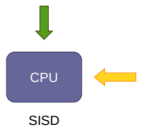
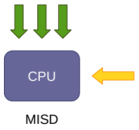
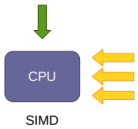
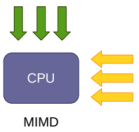
Definition 6.1.1. (**System on Chip**) A system on chip (SoC) is a collection of processors, memories and *interconnects* for a specific application.

- SoCs are:
  - **Tightly coupled**: Latency and power benefits.
  - **Customized**: Optimized for a specific application. (e.g. TPUs).
  - **Modular**: Built from modular components (assuming standardization)
- A **hetrogeneous** processor (or cores) is a processor consisting of different types of processors (CPUs, GPUs, etc).

6.1.1 Flynn's Taxonomy

- **Idea**: Classify processors into different classes $\implies$ Flynn's taxonomy (applies to theoretical models)

Definition 6.1.2. (**Flynn's Taxonomy**) Flynn's taxonomy defines the following classes:

Class	Diagram	Processor	Examples
SISD		A processor w/ a single stream of instructions and a single stream of data.	Uniprocessors, Turing Machine.
MISD		A processor w/ multiple streams of instructions and a single stream of data.	Redundancy / Error checking. Majority voting.
SIMD		A processor w/ a single stream of instructions and multiple streams of data.	Vector processing. GPUs (SIMT). k-tape Turing Machines.
MIMD		A processor w/ multiple streams of instructions and multiple streams of data	Multi-core processors

6.1.2 Parallelism, Amdahl's Law and Gustafsons' Law

- **Idea**: Exploit parallelism wrt some performance metric (usually execution time).

Definition 6.1.3. (**Speedup**) The speedup of architecture 2 with respect to architecture 1 is

$$S = \frac{T_1}{T_2},$$

where T_1, T_2 are performance metrics of architecture $1, 2$ respectively.

Theorem 6.1.1. (**Amdahl's Law**) Amdahl's law states that

$$T \text{ after improvement} = \frac{T \text{ affected by improvement}}{\text{Amount of improvement}} + T \text{ unaffected},$$

where T is some performance metric.

- Amdahl's law may be applied wrt parallelism (n-core processors)

Lemma 6.1.1. (**Speedup of Amdahl's Law**) The speedup under Amdahl's law for execution time is

$$S = \frac{1}{1 - p + p/n},$$

where p is the proportion of the program improved by parallelism, and n is the number of cores.

Proof. Let p, n be as described. Let t be the execution time and W be the *work load* (number of instructions per second, etc). The performance metric is $T = Wt$.

For fixed W, we have

$$\begin{aligned} S &= \frac{t_1 W}{t_2 W} \\ &= \frac{t}{\left[(1-p) + \frac{p}{n}\right] t} \\ &= \frac{1}{1 - p + p/n} \end{aligned}$$

□

Theorem 6.1.2. (**Gustafsons' Law**) Gustafsons law states that

$$S = (1 - p) + pn,$$

where p is the proportion of the program improved by parallelism, and n is the number of cores.

Proof. Let p, n be as described. Let t be the execution time and W be the *work load* (number of instructions per second, etc). The performance metric is $T = Wt$.

For fixed t, we have

$$\begin{aligned} W_1 &= (1-p)W + pnW \\ W_2 &= (1-p)W + pW \end{aligned}$$

Hence

$$\begin{aligned} S &= \frac{tW_1}{tW_2} \\ &= (1-p) + pn \end{aligned}$$

□

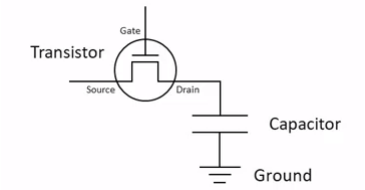
- **Conclusion**: Speedup via parallelism is only *applicable* for parallel problems (w/ high p).

6.2 DRAM

- **Problem**: Memory hierarchies give the abstraction of low-latency large addressable memories.
  Require a cheap addressable memory w/ optimizations for memory hierarchies e.g. prefetching, etc $\implies$ DRAM.

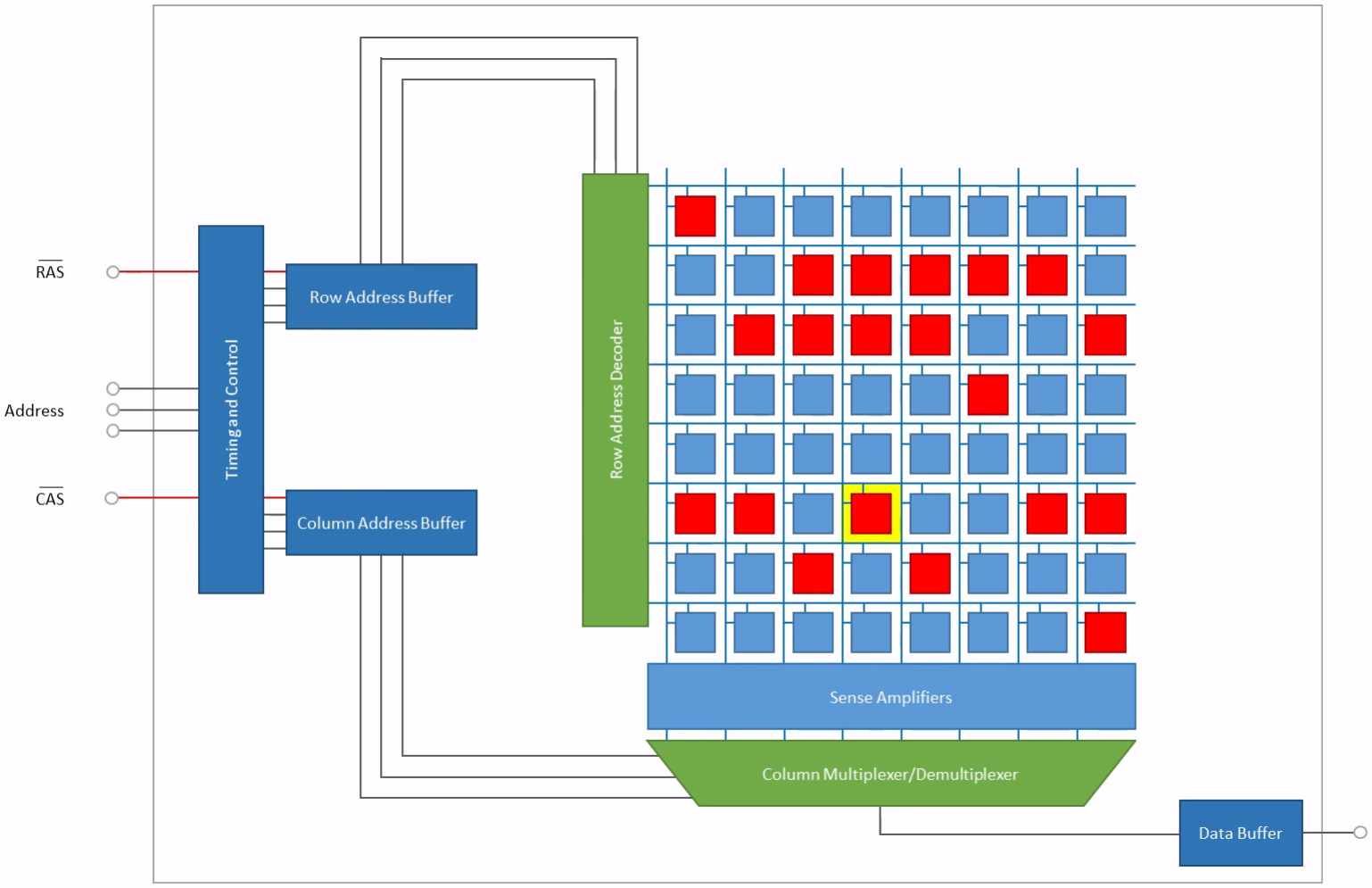
6.2.1 DRAM Cells and Arrays

- A DRAM cell is:



  where a charged capacitor represents the bit `1`.

- **Problem**: Capacitors leak charge $\implies$ data loss (if bit is 1).
- **Solution**: Periodically refresh capacitors by reading and rewriting.
- A DRAM array is a *rectangular array* of DRAM cells:



- **Reading**:
  1. The bit lines are *precharged* to $V_{DD}/2$
  2. Row address (`RAS = 1`) and row decoder set a wordline to V_{DD}.
  3. Following cases:
     - If the cell is charged (representing `1`) $\implies$ charge flows from capacitor to the bit line (increasing voltage by $\delta > 0$)
     - If cell is uncharged (`0`) $\implies$ charge flows from bit line to capacitor (decreasing voltage by $\delta < 0$).

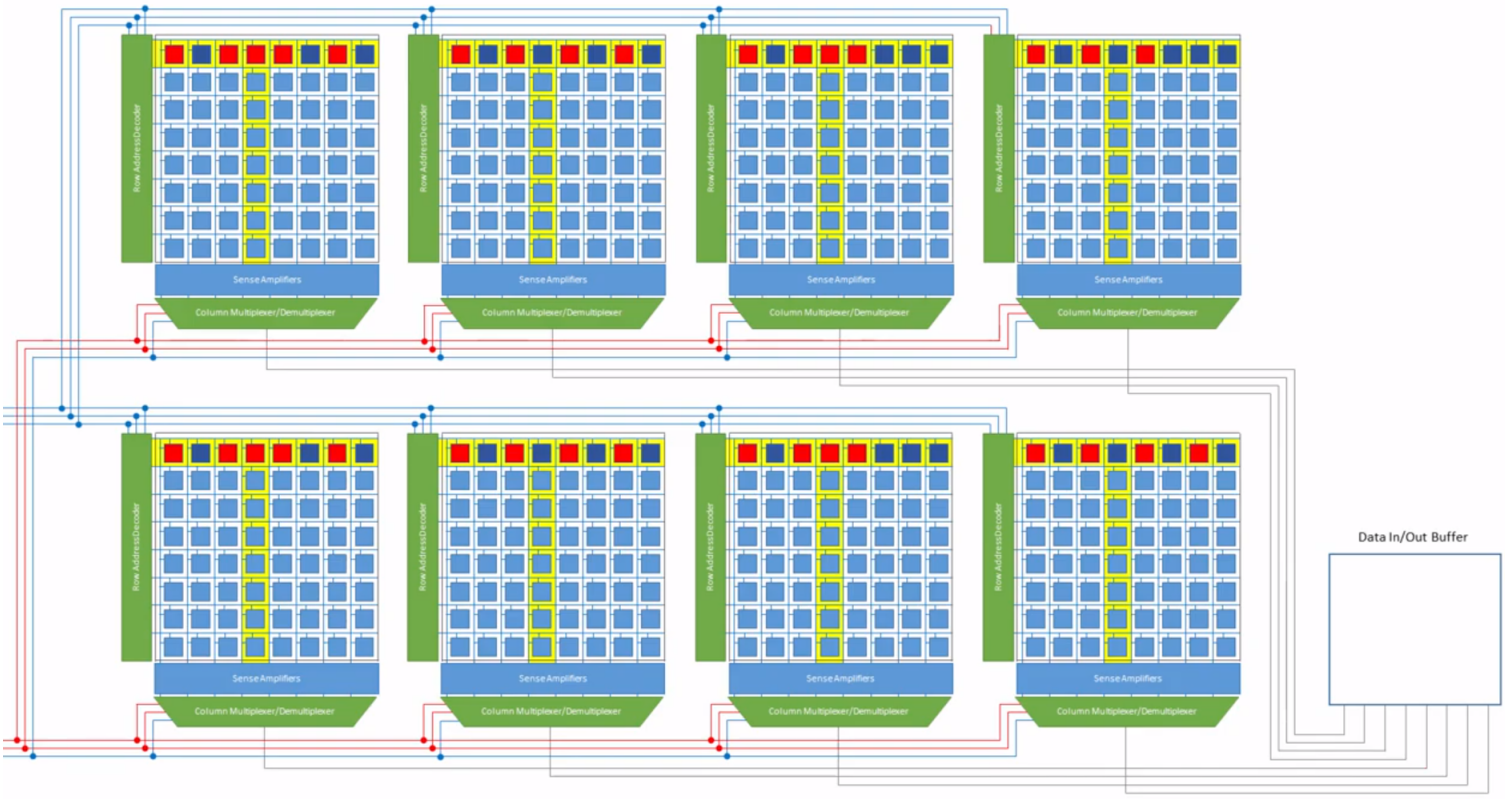
4. *Sense amplifiers* detect the change in voltage δ in the bit lines and digitize the change:
   - If $\delta > 0 \implies$ stores a `1` in internal D-flip flop
   - If $\delta < 0 \implies$ stores a `0` in internal D-flip flop
5. **Note**: Reading is a *destructive* operation (since it modifies the capacitor's charge).
   Solution: Write-back operation:
   - If `0` $\implies$ set bit line to 0 V
   - If `1` $\implies$ set bit line to V_{DD} V.
6. Set wordline to ground.
7. Column address (`CAS = 1`) and column demultiplexor *select a bit* from sense amplifiers onto data-out line.

- **Writing**:

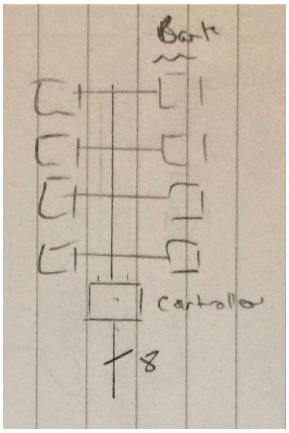
1. Read select row into sense amplifiers (See 1 - 4 above).
2. Set write enable to V_{DD}. Set data line w/ input bit.
3. Column address and column multiplexor selects sense amplifier D-flip flop. Data line bit written to DFF.
4. Performance write-back operation (See 5).
5. Discharge wordline, address lines, etc.

6.2.2 DRAM Organization

- A DRAM bank is a collection of DRAM arrays, typically 8:



- A DRAM device is a collection of DRAM banks, typically 8:



- The device controller converts between `read/write` requests into DRAM

bank read and writes:

- Selecting a bank using a bank address
- Row access with a row address and `RAS = 1`
- Column access with a column address and `CAS = 1`
- Precharge the bitlines for the next read/write.

Memory addresses are split into a bank address, row address and column address.

- A DIMM (Dual Inline Memory Module) consists of a rank (typically 8/16) of DRAM devices.
- **Note**: Devices operate independently $\implies$ each device reads 8 bits using the same address $\implies$ 64 bits.

6.2.3 DRAM Optimizations

- **Idea**: Saturate the data bus to alleviate the Von Neumann bottleneck.
- **Prefetching**:
  - **Problem**: Reading a single bit from a row is inefficient, since the write-back and precharge operations takes many cycles.
  - Optimization:
    1. Read data into sense amps and write to the data buffer.
    2. Increment the column address, reading subsequent bits into the buffer (Bank level buffer).
  - At the *bank level*, this "prefetches" a collection of bytes.
  - Allows entire cache line to be read via a single read
  - The *burst length* is the number of columns read by prefetching.
- **Double Data Rate** (DDR):
  - **Note**: 2 edges per a clock cycle, the rising and falling edge.
  - DDR sends data bursts on the rising and falling edge of a clock cycle

  - Doubles the bit rate, without increasing CPU frequency.

- **Bank Interleaving**:
  - DRAM banks:
    * Each bank operates independently within a DRAM device
    * Each bank may be at a different stage of read / write cycle
  - **Idea**: Device memory controller interleaves banks $\implies$ pipelining read/write operations.
  - By the time the last bank finishes reading / write, the first bank will have finished the write back + refresh commands.
  - Address bits split into bank, column and row addresses to ensure spatial locality exploits interleaving (wrt burst length).

- **Paging Policies**:
  - **Open page**:
    * Policy doesn't write-back and pre-charge implicitly after read-/write operation.
    * Reduces latency (since write-back and pre-charge take 10 cycles) for spatial / temporal locality.
  - **Close page**:
    * Implicitly performs write-back and pre-charge after operation.
    * Preferred when spatial / temporal locality is not present.

TRADEOFFS TODO

7 Multicore Processors

- **Idea**: Multicore processors exploit parallelism $\implies$ simultaneous reads / writes to memory *hierarchy*.
- **Problem**: Maintaining coherency between caches

7.1 Shared Memory Hierarchies

Definition 7.1.1. (**Shared Memory**) Shared memory is memory that may be simultaneous accessed by multiple processors.

- **Advantages**:
  - Communication between processors (and threads).
  - Avoids redundant copies of data if memory was disjoint.
  - DRAM is slow $\implies$ caches required for low-latency large addressable memories.

  Definition 7.1.2. (**Coherency**) Coherency is the uniformity of shared data between multiple private caches.

  Formally:

  - If processor P writes x to address A, then all subsequent reads to A must return x
  - If processor P_1 reads x from A and P_2 reads y from A (and no intermediate writes) $\implies x = y$.

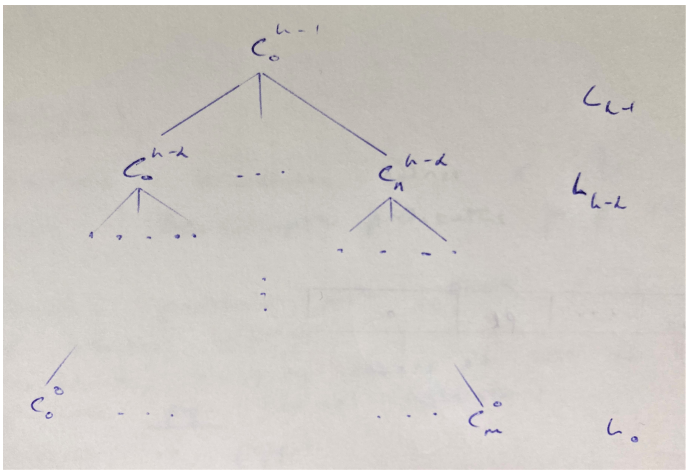
- **Disadvantages / Complications**:
  - **Lack of data coherence**: Changes in 1 private cache must be reflected in other private caches, otherwise inconsistent copies of the data (incoherence). **Solution**: Cache coherency protocols

– **Contention Issues**: Cache coherence protocols have degrading performance under contention.

7.1.1 Cache Sharing and Inclusion

- **Idea**: Split cache hierarchy into *shared caches* and *private caches* (only accessible to a single processor)

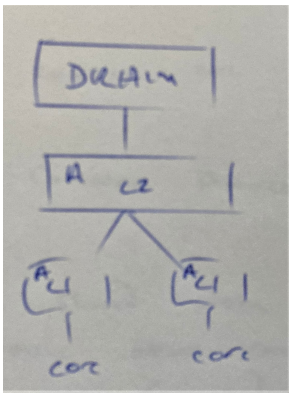
Definition 7.1.3. (**Multilevel Cache Hierarchy**) A multilevel cache hierarchy $\mathscr{C}$ of h levels of caches $L_{h-1}, \ldots, L_0$ is a tree:



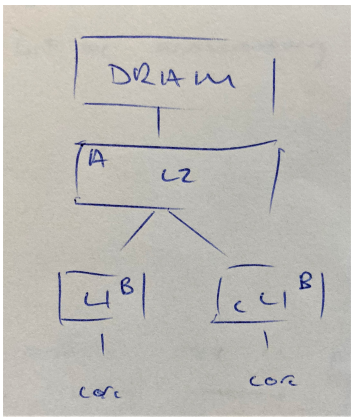
Caches in level L_{i+1} are shared between 1 or more caches in level L_i. Each cache in L_i has a *single* parent in L_{i+1}.

Definition 7.1.4. (**Cache Inclusions**) We define the following inclusions on $\mathscr{C}$:

(i) A cache hierarchy is **inclusive** $\iff$ all contents of caches in L_i is a subset of contents of the caches in L_{i+1}:



(ii) A cache hierarchy is **exclusive** $\iff$ all contents of caches in L_{i+1} is disjoint from the contents of all children caches in L_i:



(iii) A cache hierarchy is *non-inclusive non-exclusive* (NINE) $\iff$ it's not inclusive and not exclusive.

7.1.2 Cache Coherency Protocols

- **Problem**: Write-back private caches break cache coherency.
- **Solutions**:
  - **Write-through**: (See section ??). **Problem**: Still requires the cache to propagate the updated value to other private caches. Write-back caches are more efficient.

- **Cache Coherency Protocols**.

Definition 7.1.5. (**Snoopy Bus**) A shared data bus that allows other caches to view other cache transactions / requests. e.g. `read(`A`)`, `write(`A`,` x`)`.

- Snoopy busses are used to make coherency decisions.
- Coherency protocol is ran on each cache line w/ *status bits*:
  - Data is now dependent on cache coherency protocol
  - Valid and dirty bit is redundant (via new status bits)

7.1.2.1 MSI

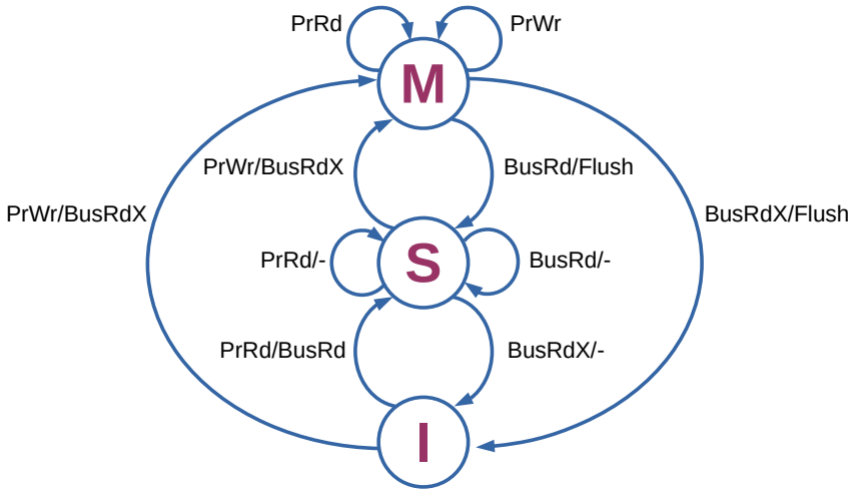
Definition 7.1.6. (**MSI**) MSI is a cache coherency protocol w/ 3 states (2 status bits):

Modified Single up-to-date copy in private cache. Copies in shared caches is stale

Shared Consistent copy in cache. Shared between caches.

Invalid No copy in current cache.

- MSI is described by a FSM (Mealy Machine) with the following events:
  - **Processor level**:
    - Processor read: `PrRd`
    - Processor write: `PrWr`
  - **Snoopy events** (Bus level):
    - Bus Read: `BusRd`
    - Bus Write: `BusWr`. Used for write-backs.
    - Bus Read Exclusive: `BusRdX`. Used to acquire unique modifiable copy of data.



- Possible state combinations:

	M	S	I
M	x	x	✓
S	x	✓	✓
I	✓	✓	✓

- **Problem**: No cache-to-cache transfers. Results in unnecessary write-backs w/ single writer, multi-reader scenarios.

7.2 Consistency

- **Problem**: Simultaneous writes and re-ordering writes / reads.

Definition 7.2.1. (**Memory Consistency**) Memory consistency is the set of constraints in which memory operations are re-ordered.

- Memory operations are often re-ordered for efficiency e.g. load hoisting.
- Sequential consistency = no reordering.
- **Problem**: Sequential consistency is inefficient. In general, we only require sequential consistency at certain points.

- **Solution**: Add re-ordering (relaxed consistency) for processor-local data. Add sequential consistency for shared data.

Definition 7.2.2. (**Memory Barrier**) A memory barrier, or *fence*, is a special instruction that ensures memory operations before the barrier are performed before memory operations after the barrier.

- Memory barriers + locks ensure sequential consistency for shared data.

7.2.1 Atomic Operations

Definition 7.2.3. (**Atomic Instruction**) An atomic instruction is an instruction that cannot be interrupted. The instruction is either executed or not, no partial effects are possible.

- Atomic instructions are used to build synchronization primitives (locks).
- *Read-modify-write* atomic instructions:
  - Used for CISC atomic operations
  - Read value, modify the value and atomically write back the new value
  - **Examples**: Fetch and add, Atomic exchange, Compare and swap
  - **Problem**: Read-modify-write instructions break RISC principles.
- **Solution**: Load linked, store conditional.

Operator	RISC-V Instruction	Effect	Type
Load Reserved	`lr.`t `rd, rs1`	`rd[rd] = data[rf[rs1]]` and places a "reservation" on address `rf[rs1]`.	R
Store Conditional	`sc.`t `rd, rs1, rs2`	`data[rf[rs2]] = rf[rs1]`. `rf[rd] = 1` $\Longleftrightarrow$ valid reservation on `rf[rs2]` address. Otherwise, `rf[rd] = 0`.	R

7.2.2 Memory Barriers and Locks

- **Problem**: Re-ordering of memory operations $\implies$ memory operations in critical section may be performed before lock is acquired!
- **Solution**:
  1. Place barrier after acquiring the lock. Ensures no memory operation in critical section occurs before the lock is acquired.
  2. Place barrier before releasing the lock.

     Ensures no memory operation in critical section occurs after releasing the lock.
- RISC-V implementation:

```
lock:
    ll.d t1, a0 # a0 = address of lock struct
    bnez t0, lock # if lock = 1 then spin
    li t1, 1
    sc.d t2, t1, a0
```

8 GPUs

- **Idea**: Specialized processors for graphics $\implies$ GPUs

8.1 GPU Design

- Old GPUs provided fixed implementations of the graphics pipeline (See IA graphics).
- Modern GPUs are highly parallelized multiprocessors exploiting SIMT + MIMD, with *programmable interfaces* that implement the graphics pipeline.
- **Goal**:
  - Maximizing *throughput* via hardware parallelism (many cores).
  - Minimizing the complexity of each core (no focus on instruction-level parallelism (pipelining))

8.1.1 SIMT

Definition 8.1.1. (**SIMD**) A SIMD processor is a processor with multiple cores performing the same instruction (single instruction stream) of multiple different streams of data.

- SIMD exploits *data-level parallelism*. Data streams are grouped into *short vectors* (typically 16 data streams per vector).
- SIMD "short vector spelling":

```
void add(uint32_t* a, uint32_t* b, uint32_t* c, size_t n) {
    for (size_t i = 0; i < n; i += 16) {
        uint32x16_t a16 = v_lw_16(a + i);
```

```
        uint32x16_t b16 = v_lw_16(b + i);
        uint32x16_t c16 = v_addw_16(a16, b16);
        v_sw_16(c + i, c16);
    }
}
```

Definition 8.1.2. (**SMT**) A SMT processor is a hardware-level multithreading processors, where the processor schedules hardware threads (`harts`, unit of scheduling).

- SMT requires duplicated hardware level thread state (register files, `pcs`, etc). Exploits *concurrency*. No context switching.

Definition 8.1.3. (**SIMT**) A SIMT processor is a hybrid model of SMT and SIMD, exploit both data-level parallelism and hardware-level concurrency.

- Programming model:
  - A program is defined as a collection of *kernels* ("compute units").
  - Each kernel is mapped to many (scheduled) *threads*, executing the same kernel instructions.
  - Threads are grouped into *thread blocks* or *co-operative thread arrays (CTAs)*.
  - Each CTA is mapped to a streaming multiprocessor (SM).
  - Each CTA is split into *warps*, the unit of scheduling in SIMT.
  - CTAs are grouped into grids. Each kernel is uniquely mapped to a grid.
- Each thread w/in a warp (on a SM) executes the same instructions in **lockstep**, but operating on different data (determined by the thread state (sub-register file)).
- SIMT "scalar spelling":

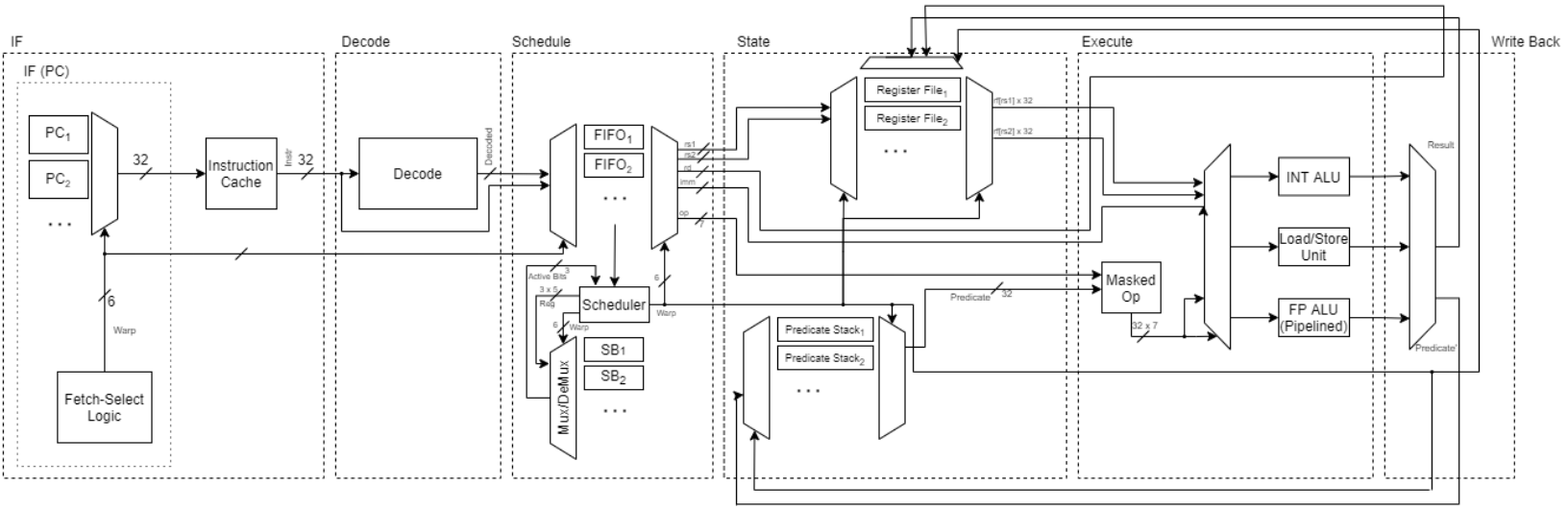
```
__device__ void add(uint32_t* a, uint32_t* b, uint32_t* c, size_t n) {
    size_t i = block.x * block.y + thread_id;
    if (i < n) c[i] = a[i] + b[i];
}
```

- Warp schedule schedules a warp each cycle (issuing a single instruction) in a round robin fashion (skipping warps w/ hazards).

  Exploits warp-level concurrency.

- **SIMT vs SIMD**:
  - Improved programming model (scalar spelling)
  - Control flow divergence (not possible effectively in SIMD)
- **SIMT vs SMT**:
  - Divergence of control flow limits SIMT efficiency.
  - Synchronization primitives leads to deadlock.

8.1.2 Streaming Multiprocessors



(*Instruction Fetch*) To exploit warp-level parallelism, we have a single program counter PC_k for each warp $1 \leq k \leq n$, where n is the number of warps per SM.

On each cycle, we select a warp k and it's program counter PC_k to fetch, skipping warps where the warp-scheduler disables instruction fetching (e.g. if the warp's FIFO instruction queue is full).

Fetch instruction using PC_k from the instruction-cache. On a cache miss, the selected PC is reset.

(*Decode*) The instruction from the *IF* stage is decoded, yielding an opcode, the encoding type e.g. R, I, etc, the destination and source registers and an immediate (if applicable).

The selected warp k is also passed through this stage (to the scheduler).

(*Schedule*) Each warp has a FIFO instruction queue, with an associated scoreboard (see below).

The output of the decode stage is enqueued onto the respective instruction queue for the selected warp k.

A round robin scheduler is used to dispatch an instruction every cycle, provided there are no:

- operand dependencies
- structural hazards

These are determined using a scoreboard (one for each warp):

- A scoreboard consists of a register containing an "active" bit for each destination register in the register file.
- On warp dispatch, the bit for the destination register for the instruction is set.
- When the writeback stage updates the register file, the "active" bit is unset.
- Before dispatch, the scheduler checks whether the active bits for the destination and source registers are set. If they are, then the warp is skipped.

These prevent RAW and WAW (WAW conflicts occur due to pipelined execute stage. Instructions can finish at different cycles) conflicts, that occur due to pipelining.

(*State*) Select register file k for given warp. Register file contains 1 sub-register file for each thread within the warp.

Predicate for warp stored in "predicate stack", entries consisting of:

- Predicate
- Diverging address.

The top of the stack stores the current predicate.

On each predicate `pop` instruction: pops the current predicate from the stack. Sets PC_k to branching address of new predicate.

(*Execute*) The predicate is used to compute the predicated opcode and operands (using a `nop` where the predicate bit for the thread is unset).

Either the integer ALU, floating-point pipeline (pipelined for throughput) or load/store units are used during the execute stage, dependent on the predicated opcode, one of each unit for the m threads.

The operands are determined using the predicated operands, the encoding type and immediate from the decode stage.

Branching addresses and predicates are computed here.

(*Writeback*) Updates the register files, predicate stack, etc.

Flushing is also performed by this stage (when branches occur). The stage also ensures that only *a single instruction* has it's writeback effects perform each clock cycle (since different instructions writeback at different times e.g. floating operations take longer than integer operations).

8.1.3 Control Flow

- **Idea**: Using predicates for diverging control flow

Definition 8.1.4. (**Predicate**) A predicate p is a m-bit string, where $p_i = 1 \implies$ thread i is active (wrt to predicate p).

- On **branch**:

  1. Calculate the branching address (diverging address) x_i for each thread. Compute the predicates $p, \overline{p}$.
  2. Push predicates onto the stack with respective diverging addresses $x, \overline{x}$.
  3. Write back, the branching address of TOS is written to PC_k

- Predicates result in inefficiency for branching programming $\implies$ GPU programs should avoid branching (a branchless programming model / minimize branch size).

Lemma 8.1.1. (**Efficiency**) The expected efficiency is

$$\mathbb{E}\left[\text{Efficiency}\right] = 1 - \frac{k}{n}(1-p).$$

Proof. Let m be number of threads, executing n instructions. Let k be the number of instructions in the branch.

We define efficiency to be

$$\text{Efficiency} = \frac{\text{number of useful instructions executed}}{\text{number of instructions executed}}.$$

Let I_i be the random indicator variable (denoted I_i) modelling whether thread i executes the branch. Define $X = \sum_{i=1}^{m} I_i$.

The number of instructions executed is nm and the number of useful instructions executed is given by

$$\#\text{ of useful instructions executed} = kX + (n-k)m = nm - (m-X)k.$$

Hence

$$\begin{aligned}\mathbb{E}\left[\text{Efficiency}\right] &= \mathbb{E}\left[\frac{nm-(m-X)k}{nm}\right] \\ &= 1 - \frac{k}{nm}\left(m - \mathbb{E}\left[X\right]\right)\end{aligned}$$

Assuming that $I_i \sim \text{Bernoulli}(p)$, where p is an unknown parameter $0 \leq p \leq 1$. So $\mathbb{E}\left[X\right] = \sum_{i=1}^{m} \mathbb{E}\left[I_i\right] = mp$. So we have

$$\mathbb{E}\left[\text{Efficiency}\right] = 1 - \frac{k}{n}(1-p)$$

□

- **Optimizations** to maximize efficiency:
  - maximize p = probability branch is taken
  - minimize # of instructions in branch = k

- **Stalls**:
  - **Idea**: Concurrency is used to *remove stalls*
  - If a stall occurs, push instruction back onto instruction queue
  - Reschedule a different warp while the hazard / cause of stall is handled (e.g. data cache misses).

8.1.4 Memories

- Thread (local) registers: This is the fast form of memory on the SM. It is only accessible to the thread. However, registers are limited by the number of available registers in the sub (local) register file for the thread.
- Local memory: This memory is only available to the thread. It resides in global memory, hence is much slower than register or shared memory. Useful for scratch-pad memory, including call stack etc. Usually used to exploit spatial / temporal locality.
- Shared memory: Shared between warps in the SM. This allows communication between threads. Very fast (however, may be slower than registers).
- Constant Memory: Accessible to all threads on the device. Fully cached, and optimized for simultaneous reads (since the memory is read-only, many reads to a given address x may be optimized to a single read to x for a collection of threads ts). Limited, since it is read-only (on the SM side).
- Global Memory: *Much* slower than register or shared memory. Accessible from either the host or device. Accessible to all threads.

8.2 GPU Programming

- **Idea**: High-level languages for GPU $\implies$ increases portability and removes reliance on processor implementation details.
- 2 main languages: CUDA and OpenCL.

8.2.1 CUDA

- Developed by NVIDA. C-like language w/ additional GPU syntax.
- Program split into a collection of `host` and `device` procedures:
  - `__host__` procedures execute on CPU and interface with GPU
  - `__device__` (or `__global__`) procedures execute on GPUs. Each `__device__` procedure is a *kernel*.
- Memory allocation and transfer:
  - `cudaMalloc(void** dst_ptr_ptr, size_t` n`)`. Allocates n-bytes of GPU shared memory. Stores address in `*dst_ptr_ptr` in host memory.
  - `cudaMemcpy(void* dst, const void* src, size_t` n`, cudaMemcpyKind kind)`. Copies n-bytes from `src` to `dst`, where

```
typedef enum {
    cudaMemcpyHostToHost,
    cudaMemcpyHostToDevice,
    cudaMemcpyDeviceToHost,
    cudaMemcpyDeviceToDeivce
} cudaMemcpyKind;
```

- Kernels:
  - Kernel procedure invocations of the form:

```
kernel_procedure<<<gridDim,>>>
```

    `gridDim` is the number of blocks within the grid. `blockDim` is the number of threads within each block. (`grid_dim, block_dim` may be 1/2/3D).
  - CUDA variables:
    - `gridDim`: dimensions of grid of blocks
    - `blockDim`: dimensions of each block
    - `blockIdx`: index (1/2/3D indices) of block
    - `threadIdx`: index (1/2/3D indices) of thread
    - `warpSize`: Number of threads per warp.

8.2.2 OpenCL

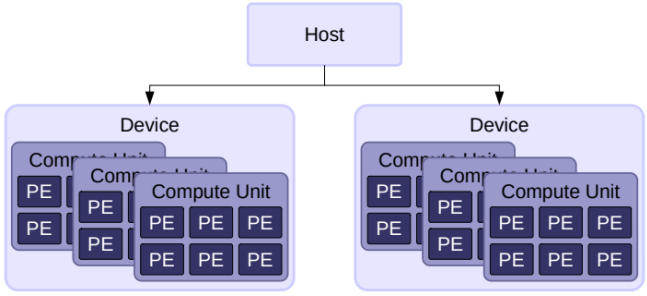
- Heterogeneous programming language
- Open standard, defines a specification. Applicable to CPUs, GPUs, FPGAs, etc.
- Defined 4 models in the OpenCL standard:

(Platform) The OpenCL platform model consists of $n \geq 1$ platforms, each platform p consisting of a host H_p and $m_p \geq 1$ devices, $D_1^p, \ldots, D_{m_p}^p$.

```
////////////////////////////////////////////////////////////////////
// Platform Discovery
////////////////////////////////////////////////////////////////////

cl_int n = 0;

// Obtain number of platforms available
if (clGetPlatformIDs(0, NULL, &n) != CL_SUCCESS) {
    // Handle error
}

// Allocate and clear platforms array
cl_platform_id platforms[n];
memset(&platforms[0], (uint8_t)0, sizeof(cl_platform_id) * n);

// Populate platforms array
if (clGetPlatformIDs(n, &platforms[0], NULL) != CL_SUCCESS) {
    // Handle error
}

////////////////////////////////////////////////////////////////////
// Device Discovery
////////////////////////////////////////////////////////////////////

// Arbitrary logic to select platform p
cl_platform_id p = f(n, &platforms[0]);

// Obtain number of GPUs on platform
```

```
cl_int m_p = 0;
if (clGetDeviceIDs(p, CL_DEVICE_TYPE_GPU, 0, NULL, &m_p) != CL_SUCCESS) {
    // Handle error
}

// Allocate and clear device array
cl_device_id devices[m_p];
memset(&devices[0], (uint8_t)0, sizeof(cl_device_id) * m_p);

// Populate devices array
if (clGetDeviceIDs(p, CL_DEVICE_TYPE_GPU, m_p, &devices[0], NULL)
    != CL_SUCCESS) {
    // Handle error
}
```

Each device split into compute units $C_1^{D_i^p}, \ldots C_k^{D_i^p}$, where k is the number of compute units in device D_i^p. Each compute unit is split into ℓ processing elements $PE_1^{C_j^{D_i^p}}, \ldots, PE_\ell^{C_j^{D_i^p}}$.



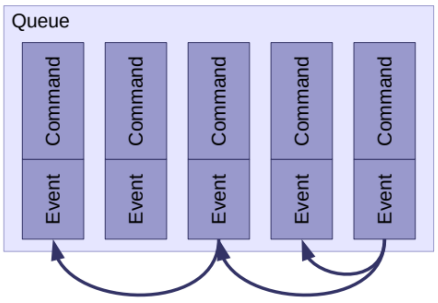
(Execution) The OpenCL execution model describes a context C, a platform p with a set of available devices (D_i^p) for the platform used to manage the interaction between host and device using *command queues* and memories.

```
// Creating a context
cl_int err_code = CL_SUCCESS;
cl_context context = clCreateContextFromType(NULL, CL_DEVICE_TYPE_GPU,
    NULL, NULL, &err_code);
if (err_code != CL_SUCCESS) // Handle error
```

```
// Creating a command queue q
cl_int err_code = CL_SUCCESS;
cl_command_queue q = clCreateCommandQueue(context, D, NULL, &err_code);
if (err_code != CL_SUCCESS) // Handle error
```

A command is an instruction from the host H_p to the device D_i^p within the context C, with one command queue per a device, storing commands. e.g. `clEnqueueNDRangeKernel`.

Each command is mapped to an event object, used to model dependencies between commands using *wait-lists*. Used for out-of-order queues, to allow re-ordering (for efficiency) without breaking consistency. Events objects also contain the state of the command.



(Programming) The OpenCL programming model states that an application consists of many kernels, each kernel is executed on a device.

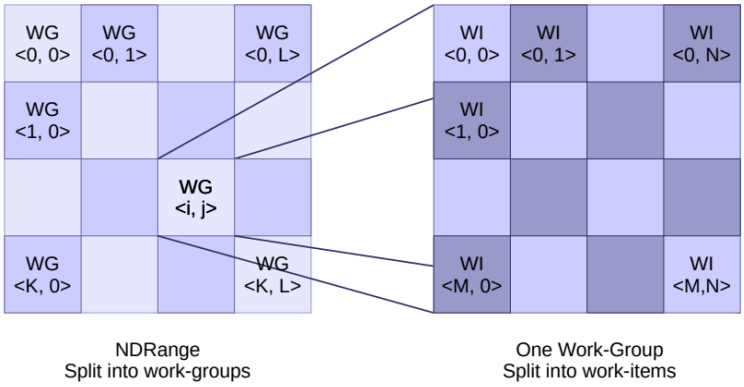
Kernels are compiled at runtime, allowing for device specific compilation and optimizations. This also increases portability.

```
cl_int err_code = CL_SUCCESS;
cl_program p = clCreateProgramWithSource(context, 1,
    &kernel_source, NULL, &err_code);
if (err_code != CL_SUCCESS) // Handle error

// Compile the program for device D
if (clBuildProgram(program, 1, &D,
    NULL, NULL, NULL) != CL_SUCCESS) // Handle error
```

```
// Compiled program is used to define a kernel
cl_int err_code = CL_SUCCESS;
cl_kernel k = clCreateKernel(p, "kernel name here", &err_code);
if (err_code != CL_SUCCESS) // Handle error

// setup kernel arguments using clSetKernelArg
...
```

A work-item is the unit of parallelism (corresponds to a thread). The number of work-items mapped to a kernel is given by a n-dimensional range (NDRange) (a grid). Work-items in an NDRange are grouped into work-groups (a CTA), each work-group is allocated to a compute-unit.

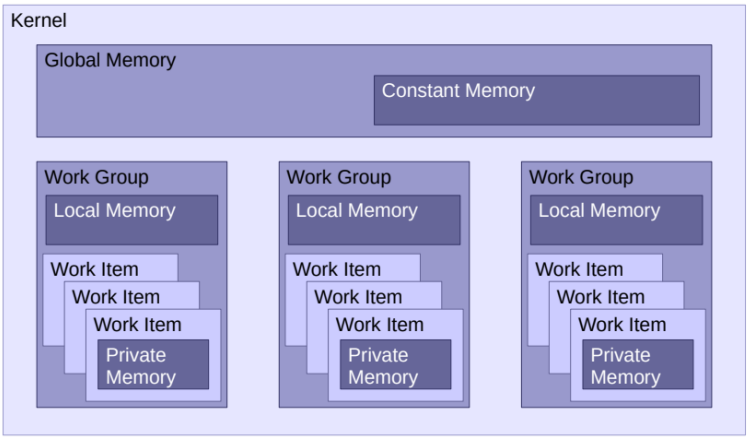
```
// global_work_size is nD array w/ dimensions of CTA.
// local_work_size is nD array w/ dimensions of work-group
cl_event e;
if (clEnqueueNDRangeKernel(q, k, n, NULL,
    global_work_size, local_work_size,
    0, NULL, &e) != CL_SUCCESS) // Handle error
```



(Memory) The OpenCL memory model states that memory is either host memory or device memory. The specification doesn't define host memory. Device memory is split into the following types:

* Global memory. Visible to all work-items
* Constant memory. Visible to all work items. Provides simultaneous access.

    * Local memory. Visible to all work items within a work group.
    * Private memory. Visible to each work-item,



    The standard also defines standard *memory objects*:

    * Buffers. A collection of elements stored as a contiguous sequence of bytes. `clCreateBuffer`
    * Pipes. An ordered sequence of elements, similar to a FIFO queue. `clCreatePipe`
    * Images. A collection of pixels. Defined specifically for image processing. `clCreateImage`

- Queue synchronization:
    - `clFinish(`q`)` *blocks* until all previously queued commands in q have been issued to the associated device **and** have completed. Used for synchronization barrier.
    - `clFlush(`q`)` *issues* all previously queued commands in q to the associated device (eventually executing the commands). **Non-blocking**.

OpenCL	CUDA
OpenCL is an open standard that can be used to program CPUs, GPUs, and other devices from *different vendors*	CUDA is a specific language to Nvidia GPUs
Unit of execution is a *kernel*	-
Kernel compiled at runtime for portability. Adds to the OpenCL running time, however, the runtime compilation allows for device-specific optimizations	Kernels compiled at compile-time into Nvidia's proprietary PTX low level parallel-thread execution ISA. PTX is compiled at runtime (by the GPU) to SASS, a low-level assembly language. This allows for GPU-specific optimizations, while allowing a target ISA for all Nvidia devices.
Both have a similar device memory model of private, local, shared and global (with constant) memory	-
Similar mapping of kernels to processing elements (threads)	-
Kernel initialization / execution sequence vastly differs	-