Computer Architecture

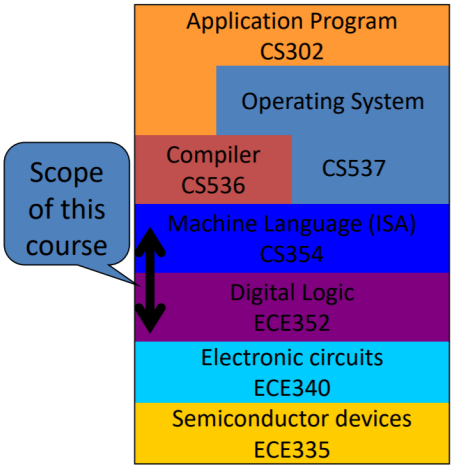
Author: Guanzhou (Jose) Hu 胡冠洲 @ UW-Madison CS552

Teachers: Karu Sankaralingam

Figures included in this note are from Prof. Sinclair's and Prof. Lipasti's slides.

Great Idea: Abstraction

The scope of this course:



One of the greatest ideas of computer architecture is **layers of abstraction**. An *abstraction* is a name and an *interface* we use to describe a component to hide *implementation* details of this component.

- *Interface*: What something does? E.g., 2:1 MUX, use select bit S to switch output F between inputs X and Y
- *Implementation*: How it does so? E.g., logic gates, pass transistors, registers, ...

Instruction Set Architecture (ISA)

Instruction Set Architecture (*ISA*) is a perfect example of abstraction which hides details of underlying hardware and provides an interface for software.

- The vocabulary that a processor understands
- Defines the interface - a set of *assembly instructions* - to the processor
- Implementations (hardware realization) of an ISA do vary. e.g., different Intel processors on x86

Different types of ISAs exist:

- *Reduced Instruction Set Computing* (RISC): MIPS, RISC-V
- *Complex Instruction Set Computing* (CISC): Intel x86 IA-32, amd64, ARM
- IBM PowerPC, Sun SPARC, ...

Hardware Description Languages (HDL)

Designing hardware by bottom-up modules is another sign of abstraction in this field. We will use this idea a lot when we design hardware blocks in *Verilog* to achieve a complex functionality. When we feel that a certain small functionality should be put in a separate module for easier reuse, we put them in a separate module. A complex module can instantiate sub-modules which may again reference smaller modules.

A **hardware description language** (*HDL*) is a high-level language that describes:

- What the hardware looks like
- What are its components
- How components are wired together
- ...

HDLs are NOT "programming languages"! It gets generated (compiled, mapped, synthesized) to a hardware chip, not to commands for processors to execute.

We will be using the *Verilog* HDL to design hardware blocks that achieves certain functionality. As a hardware programmer, when programing Verilog, keep the following in mind:

1. We are *describing* hardware:
    - Describing what the components are and how they are wired together
        - Hardware is all active at once
        - A simple incrementer can introduce a lot of hardware - program carefully
    - NOT programing a time-sequenced procedure like software programming
2. Think in *divide-and-conquer* way:

    - When you feel like a certain piece of logic should be put into a separate module for easier reuse and cleaner code, do that

Verilog itself has advanced features, enabling hardware programing in advanced models:

- *Structural* model
- Register Translation Language (RTL) *Dataflow* model
- *Behavioral* model

Some resources to reference for using Verilog in this course:

- Verilog tutorial slides
- Karu's Verilog cheatsheet

Performance Metrics

It is not so simple to answer the question: "How fast/good is a computer?" There are many factors that can influence the answer:

- *Workload* we are targeting at
    - Integer, FP, ...
    - Database - memory, storage & I/O
- Correct *measurement* and analysis
    - *Response time* (*Latency*)
    - *Throughput*
- *Cost*, ...

Response Time vs. Throughput

Response time (*Latency*) measures elapsed time for program to complete

- $= t_{end} - t_{start}$
- $=$ CPU time + I/O wait

Throughput measures completion rate

- $=$ delivery / sec

Throughput $=$ 1 / avg. response time only if jobs do not overlap. Otherwise, throughput will be larger than that. To improve performance:

- Build faster CPU core - helps both response time and throughput
- Add more cores (more *parallelism*) - helps throughput
- Do staged *pipelining* - helps throughput

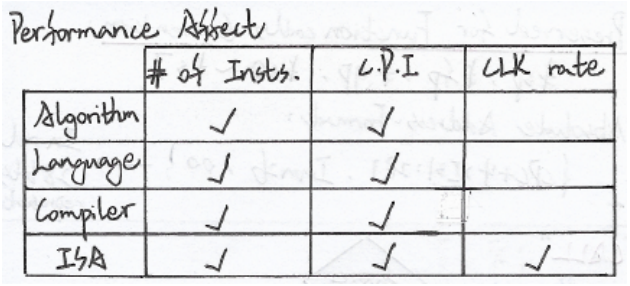
Iron Law

The Iron Law is the formal way of comparing processor performance.

$$\text{Time for program} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

- $\frac{\text{Instructions}}{\text{Program}} \rightarrow$ Compilers, determined by algorithm, compiler, & ISA
- $\frac{\text{Cycles}}{\text{Instruction}}$ (CPI) $\rightarrow$ Processor design, determined by ISA & CPU datapath
- $\frac{\text{Time}}{\text{Cycle}} = \frac{1}{\text{clock rate (frequency)}} \rightarrow$ Chip realization, determined by circuit design and manufacturing

Terms are inter-related. Do NOT only optimize for an isolate term (e.g., changing ISA to decrease instruction count) because that usually leads to a related term getting worse (e.g., making CPU datapath slower from this change). We should optimize for the overall performance.



> We say machine A is `20%` faster then machine B when $\frac{\text{Performance A}}{\text{Performance B}} = \frac{\text{Time B}}{\text{Time A}} = 1.2$.

MIPS & MFLOPS

MIPS (*million instructions per second*) $= \frac{\text{instructions}}{\text{time} \times 10^6}$

- Has serious shortcomings. See slide #2, page 16-17
- It is OK to use only when comparing processor implementations with the same compiled program and the same ISA

MFLOPS (*million floating-point operations per second*) $= \frac{\text{FP ops}}{\text{time} \times 10^6}$

- Assuming FP ops are independent of compiler and ISA, then safe for numeric codes

Beware when reading documents, especially if details are omitted. Beware of the language of "peak performance".

Benchmarking

What program should we use to (somewhat) fairly compare processor performance? We use a set of standard programs called as *benchmarks*. How to average the performance on multiple different programs?

- Use *weighted arithmetic mean* (AM) <u>of times</u> $= \frac{\sum(w_i \cdot \text{Time}_i)}{n}$ to get average time
  - If programs run equally often, may use unweighted arithmetic mean
- Use *harmonic mean* (HM) <u>of rates</u> $= \frac{n}{\sum(\frac{1}{\text{Rate}_i})} = \frac{n}{\sum(\text{Time}_i)}$ to get average rate
- Use *geometric mean* (GM) <u>of relative ratios</u> $= \sqrt[n]{\prod(\text{Ratio}_i)}$
  - Do NOT use arithmetic mean on relative ratios of machines! - Simple fact in math
  - GM of ratios ignores total time. It bias a machine if it has an advantage on programs with shorter execution time

> A CPU benchmark like SPEC2000 is meaningful only when your workload is NOT I/O bound.

Amdahl's Law

Modern processors all exploit parallelism. **Amdahl's Law** gives a simple theoretical bound on possible speedup brought by parallelization.

$$\text{Speedup} = \frac{1}{(1-f) + \frac{f}{s}}, \text{ where}$$

- f is the fraction of your program that can be parallelized
- s is the number of cores

Insight: If f is small (your workload is inherently not parallelizable), then the maximum possible speedup is very limited even when you have infinitely many cores.

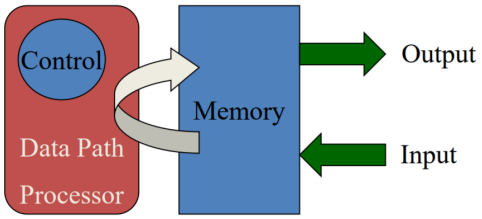
ISA Concepts - MIPS

An ISA defines a set of assembly instructions with the general forms:

```
    opcode operand(s)
```

- *Opcode* is the name (mnemonic) we give to specify an operation
- *Operand* is a *Register* or an "*Immediate*" number

An ISA assumes a *Von Neumann machine* model:



MIPS is a RISC ISA which composes of only very simple instructions. Check the attached MIPS data sheet for technical details. You may also find the WikiBook page and this page useful.

Registers & Alignment

MIPS has 32 exposed registers `$0 - $31` residing in the processor, each 32-bit (*word*) long. Some of them have an alias name, e.g., `$zero` for `$0`, `$ra` (return address) for `$31`.

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	No

Some of these registers are reserved for the assembler to use as temporary values. Some are reserved by the *Application Binary Interface* (ABI) and the *calling convention*. It is VERY dangerous if we arbitrarily use registers without paying attention to the context!

> - `$0` cannot be written and is always a 32-bit zero
> - `$at` is reserved for the assembler only, users should NEVER code it explicitly
> - `$t0 - $t7` registers are NOT preserved across a function call: The callee can use them without saving; After the call returns, words previously in them may change
> - `$s0 - $s7` registers SHOULD be preserved across a function call: The callee, by design, SHOULD save them if it uses them; After the call returns, the caller is safe to assume that words in them remain the same

Data alignment lengths are named as:

- 64-bit: *double word*
- 32-bit: *word*, `w`
- 16-bit: *halfword*, `h`
- 8-bit: *byte*, `b`

> *Endianness* refers to how bytes are addressed within a multi-byte datatype.
>
> - Little endian: LSB in lowest address
> - Big endian: MSB in lowest address
>
> MIPS is endian-mode selectable.

Core Instructions

Core instructions are those enabling the full functionality of MIPS. They generally take one of the following forms:

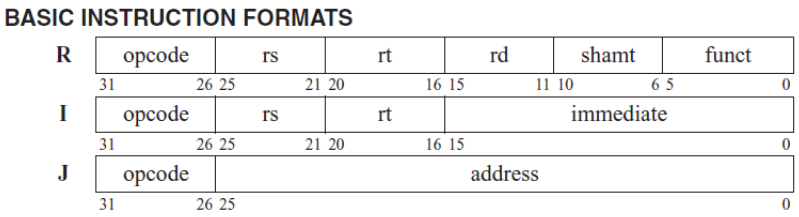
Type	Format	Instruction & Example
0.		`opcode` `int`
1.	`[R]`	`opcode $rs` `jr addr`
2.	`[J]`	`opcode Label` `j/jal addr`
3.	`[R]`	`opcode $rd, $rs, $rt` `add dst src1 src2`
4.	`[R]`	`opcode $rd, $rt, Shamt` `sll dst src shift`
5.	`[R]`	`opcode $rt, $rs` `div deno nom`

6.	`[I]`	`opcode $rt, $rs, Imm/Label`
		`addi dst src1 src2`
		`beq cmp1 cmp2 ofs`

7.	`[I]`	`opcode $rt, Imm` `lui dst const`
8.	`[I]`	`opcode $rt, Imm($rs)` `lw/sw data ofs addr`

- Basic arithmetics
  - Opcode:
    - `add`, `addiu`, `and`, `andi`, `sll`, `slt`, ...
    - `mult` & `div` give wider result and save results to fixed `hi` & `lo` registers
  - Operand:
    - 32-bit registers
    - 16-bit immediate constant
    - Shifts use a 5-bit "Shamt" immediate
  - Computation is done in the *Arithmetic Logic Unit* (ALU) for these basic integer arithmetics
- Memory operations: 32 words are certainly not enough of storage - we need to load from / store to memory
  - Opcode:
    - `lw $rt, Imm($rs)` for load word
    - `sw $rt, Imm($rs)` for store word
  - Operand:
    - A register for holding the word
    - A register holding the memory address
    - An immediate of address offset
  - Register is 32-bits long meaning that MIPS, by default, can address up to 4GB memory space
- Branches & Jumps: *control flow* instructions
  - Opcode:
    - `beq`, `bne`
    - `jr`, `j`, `jal`, ...
  - Operand:
    - Two registers for branch condition
    - A register, immediate, or *label* specifying the branch/jump distance/target
    - A label is coded as a name but translated to a PC-relative offset later
- Floating-Point instructions and miscellaneous instructions omitted here

A core instruction, according to its format, gets encoded into a 32-bit binary by the assembler:



> Some R-format instructions share the same opcode and have different "Funct" codes.

Pseudo-Instructions

Though core instructions are already functionally complete, we sometimes want an alias instruction to save the effort writing frequently-used snippets which may be too tedious in core instructions. MIPS thus includes a bunch of *pseudo-instructions* which get assembled into a snippet of core instructions in the first pass of assembling.

Commonly used pseudo-instructions:

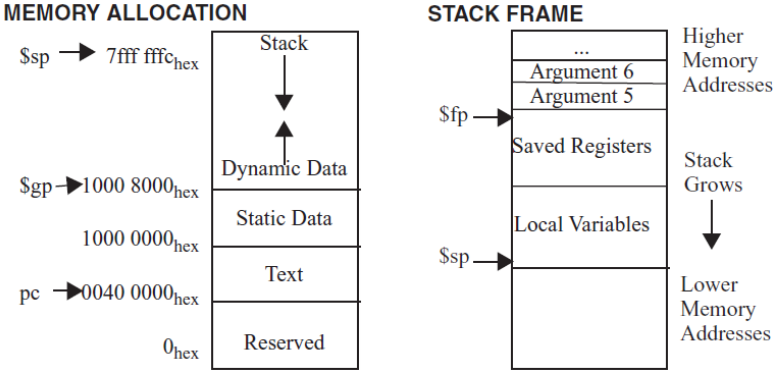
- `blt $rs, $rt, Label` ("branch if larger than") =
  1. `slt $1, $rs, $rt`
  2. `bne $1, $0, Label`
- `li $rd, Imm` ("load immediate") =
  1. `lui $at, UpperOfImm`
  2. `ori $rd, $at, LowerOfImm`
- `move $rd, $rs` =
  1. `add $rd, $rs, $0`

Procedure Calling Convention

Invoking a function means doing a *procedure call*. The **calling convention** is part of ABI and is what we should pay attention to when coding assembly.

- Caller:
  1. Save registers *that the caller is responsible to save* in current context
  2. Set up function arguments
  3. Jump to the procedure
  4. Callee does its work
  5. Get the results
  6. Restore context registers
- Callee:
  1. Save some more registers *that the callee is responsible to save*
  2. Do some work
  3. Set up the results where the caller knows to get from
  4. Restore registers
  5. Return

In MIPS ISA, stack grows from higher address to lower address. *Memory layout and stack frame layout*:



- Stack pointer `$sp` points to last word on stack `0($sp)`
- Frame pointer `$fp` points the first word of current function frame (callee's frame), and this first word SHOULD be the upper frame (caller's frame) frame pointer value

- First 4 word-long function arguments SHOULD be passed in registers `$a0 - $a3`
- More arguments SHOULD be ready in the caller's frame right above `$fp`

Example procedure which swaps the `k`-th and `(k+1)`-th elements in an array:

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

May compile to the following program in MIPS:

```
    ...
    move    $a0, <arr>
    li      $a1, <k>
    jal     swap
    ...

swap:
    # Prologue
    addi    $sp, $sp, -4
    sw      $fp, 0($sp)
    addi    $fp, $sp, 0

    # $t1 = arr + 4*k = &arr[k]
    sll     $t1, $a1, 2
    add     $t1, $a0, $t1

    # $t0 = arr[k]; $t2 = arr[k+1]
    lw      $t0, 0($t1)
    lw      $t2, 4($t1)

    # arr[k] = $t2; arr[k+1] = $t1
    sw      $t2, 0($t1)
    sw      $t0, 4($t1)

    # Epilogue & Return
    lw      $fp, 0($sp)
    addi    $sp, $sp, 4
    jr      $ra
```

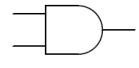
Arithmetic & Logic

The ALU is responsible for carrying out integer arithmetic and logic calculations.

A 32-bit binary can represent different things, depending on how we view it:

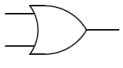
- Unsigned integer $\in [0, 2^{32} - 1]$
- Signed integer: *2's complement* $\in [-2^{31}, 2^{32} - 1]$
  - Unique zero: `0 ... 000`
  - To get $-A$, revert every bit of A, then $+1$
  - 2's complement makes hardware simpler than *sign-magnitude*
- Single-precision floating-point: *IEEE 754*
- A MIPS instruction

We use these *logic gate* symbols:



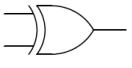
AND

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1



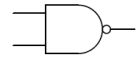
OR

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1



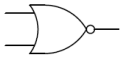
XOR

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0



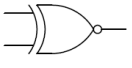
NAND

A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0



NOR

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0



XNOR

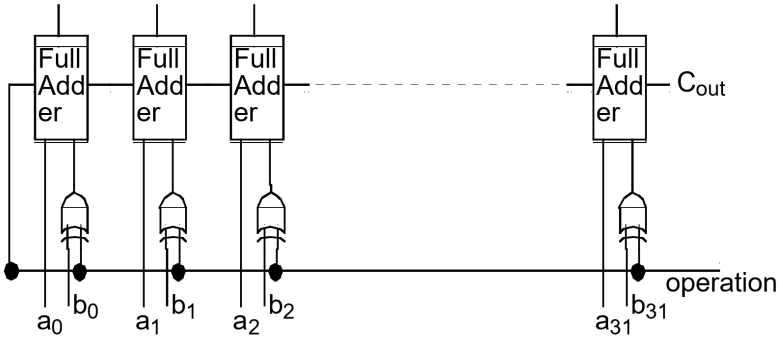
A	B	Output
0	0	1
0	1	0
1	0	0
1	1	1

Adder & Subtractor

A *full adder* is an adder adding two bits which takes care of *carry-in* & *carry-out*: $(a, b, c_{in}) \Rightarrow (c_{out}, s)$. The boolean table of a full adder is:

a	b	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Concatenating 32 full adders gives us a *ripple-carry adder*. Since subtracting B is equivalent to adding $-B$, a *ripple-carry subtractor* is a ripple-carry adder with all b bits negated and setting c_{in} to 1. A combined ripple-carry adder/subtractor:



> An XOR gate can be used as a "conditional" negation gate.

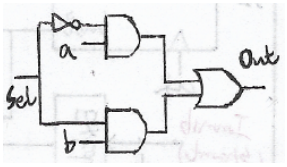
> In MIPS, the only difference between `add` and `addu` is that `add` causes overflow detection while `addu` does not. Overflowing can be detected by:
>
> - Signed add: c_{in} into MSB $\neq$ c_{out} out of MSB
> - Unsigned add: $c_{out} \neq 0$

Bitwise Boolean Logic

Doing 32-bit bitwise AND, OR, XOR, NAND, NORs are simple - just implement with 32 gates in parallel.

2-to-1 MUX

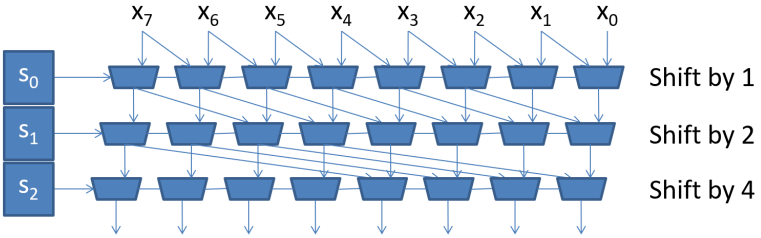
A MUX is a selector: using a select bit to switch the output between inputs. A *2-to-1 MUX* can be implemented as:



A *4-to-1 MUX* can be implemented with three 2-to-1 MUXs.

Shift & Rotation

A shifter shifts bits of a number to the left/right. A *rotator* wraps around those bits shifted out. An 8-bit shifter can be implemented using three layers of MUXs:

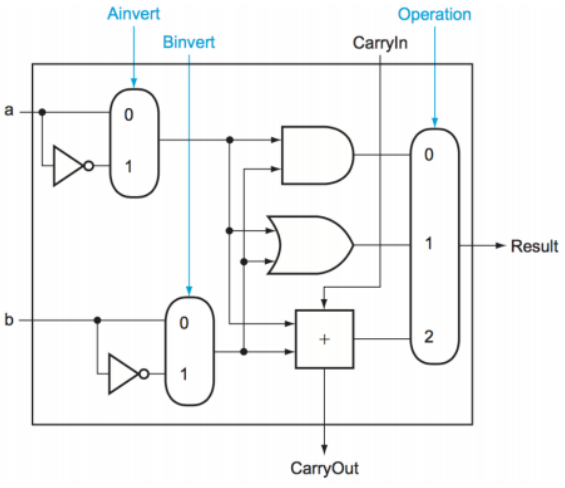


To rotate, just wrap-around instead of putting `0`s in missing inputs.

> In MIPS, the only difference between `srl` and `sra` is that `srl` does logical shift (injecting `0`s to missing inputs) while `sra` does arithmetic shift (injecting the number's old MSBs).

The ALU

The ALU itself is a combination of arithmetic components listed above, plus a MUX selecting which output we are using. A simple 2-input ALU with addition/subtraction/bitwise boolean support looks like:



To do an `NOR(a, b)` for example, set `Ainvert = 1`, `Binvert = 1`, and `Operation = 0`.

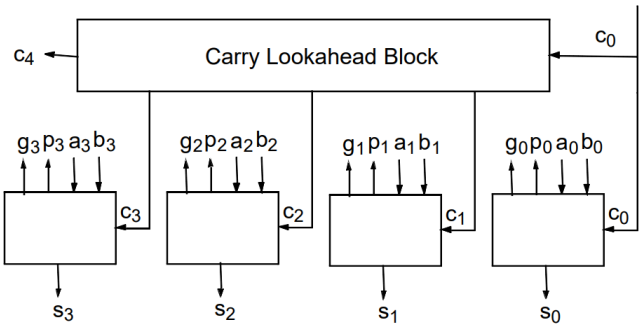
Carry Lookahead

The ripple-carry adder we mentioned above is very non-performant: it cascades 32 full-adders. A higher-bit adder needs to wait for the lower-bit to finish. A high-performance substitute is the *carry-lookahead* adder.

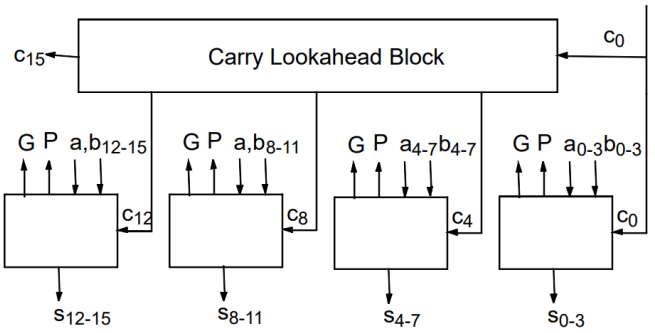
Looking at the two numbers we are adding, for each bit:

- Need both `1`s to generate carry: $g_i = a_i * b_i$
- Need one or both `1`s to propagate carry: $p_i = a_i + b_i$

Therefore, carry-in $c_{i+1} = g_i + p_i * c_i$. Expanding this up to 4 bits, we get a 4-bit carry-lookahead adder:



The carry-lookahead block is still doing cascading job. To amortize the workload and squeeze parallelism out of this, we build a tree-structured 16-bit carry-lookahead adder out of four 4-bit ones:

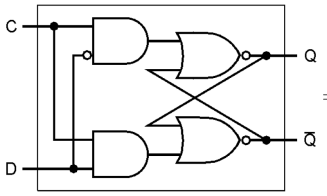


- $P = p_0 * p_1 * p_2 * p_3$
- $G = g_3 + g_2 * p_3 + g_1 * p_3 * p_2 + g_0 * p_3 * p_2 * p_1$
- C_{out} of a 4-bit group $= G + P * C_{in}$

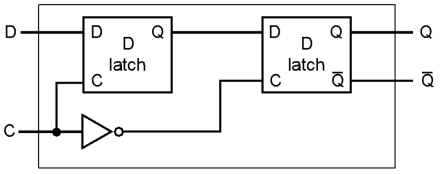
Processor Datapath & Control

Recall *sequential logic* using *D flip-flops* (to build *registers* `D -> Q`) + a *clock* signal `C`:

D latch:



⇒ Master/Slave D Flip-Flop:



- A latch is transparent during the whole period when clock is high
- A flip-flop, instead, ONLY propagates the value of `D` to `Q` at rising edge, and is opaque at any other time

Therefore, we use flip-flops to build registers for sequential logic to avoid *combinational loops*.

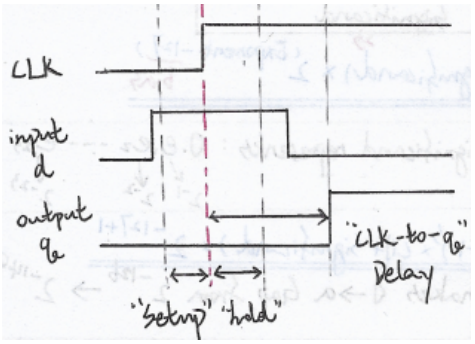
Fully Synchronous Design (FSD) Clocking

In this course, we will use *fully synchronous design* (FSD) clocking methodology:

- Only use flip-flops (FFs) in between logic pieces
- All with the same clock signal and all acting on the same edge, e.g., rising
- All logic pieces finish in one cycle

This simplifies our design and enables us to design the full datapath without worrying about clocks.

FFs have setup time & hold time requirements to function properly.



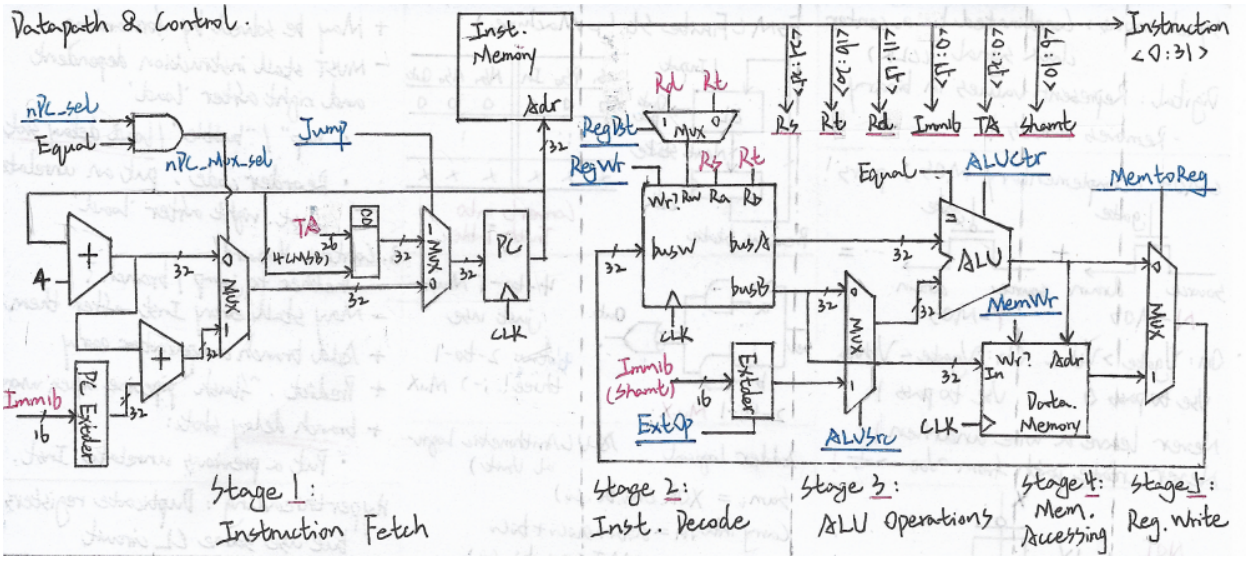
We further denote that clock cycle length is T_{clock}, and the combinational logic delay following the FF is T_{comb}. FSD clocking rule states that:

1. $T_{clock} > T_{clk\_to\_q} + T_{comb\_max} + T_{setup}$
2. $T_{clk\_to\_q} + T_{comb\_min} > T_{hold}$

> The combinational logic block having T_{comb_max} is called the *critical path*. It decides the upper-bound of clock frequency.

MIPS 5-Stage Datapath

A classic 5-staged MIPS processor datapath design is:



It can be divided into 5 stages (for future pipelining optimization):

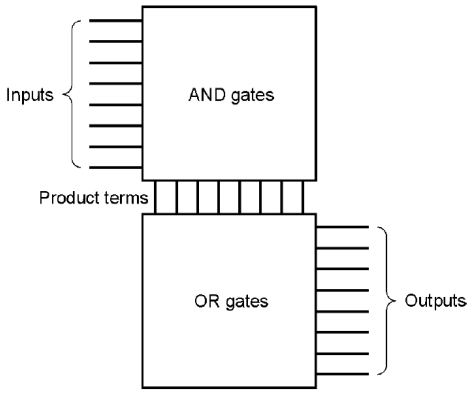
1. Instruction Fetch (IF, F)
2. Instruction Decode (ID, D)
3. ALU Operations/Execution (EX, X)
4. Memory Accessing (MEM, M)
5. Register Write (WB, W)

I draw instruction memory separately from data memory, meaning that this is a *Harvard architecture*. (On a *Von Neumann architecture*, code shares the same physical memory with data.) The block in the middle is a *register file*, i.e., the collection of on-chip registers.

Signals marked red are constants decoded from the instruction, and those marked blue are *global control* signals, which we need to compute in the *global control unit* from the `opcode` & `funct` fields decoded from the instruction (not shown in the figure).

Global Control PLA

Those global control signals have to be computed from the `opcode` & `funct` fields. A common solution is to use a PLA unit:



, where each input is an `opcode` / `funct` bit, each AND product term represents whether this is a certain type of instruction, and the OR gates combine this information to compute all global control signals.

One design goal of an ISA (e.g., MIPS) is to carefully design the opcodes and control signals to minimize the combination logic need in this PLA.

Ideal Pipelining

The problem of a single-cycle CPU design is that it delivers very poor throughput. Consider the `lw` instruction. It needs to go through all five stages (F + D + X + M + W) to complete. Though other instruction might need fewer stages, the global clock frequency is bounded by the critical path instruction.

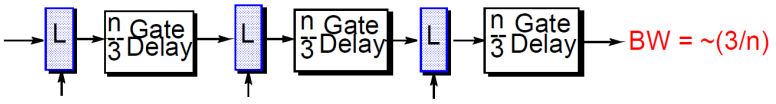
Recall that we measure time for a program as:

$$\text{Time for program} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

- CPI = 1
- Time per cycle is large

Modern processors use *multi-cycle* design with pipelining. **Pipelining** can be applied when:

- Each task uses multiple resources (here, different combinational blocks on chip) in an order of stages
- When the previous task reaches stage 2, the next task can begin processing its stage 1; they are multiplexing different parts of one datapath
- Will improve total throughput (by #stages maximum, if all stages are even); will not improve latency of a single instruction (will actually slightly worsen that because of extra registers)

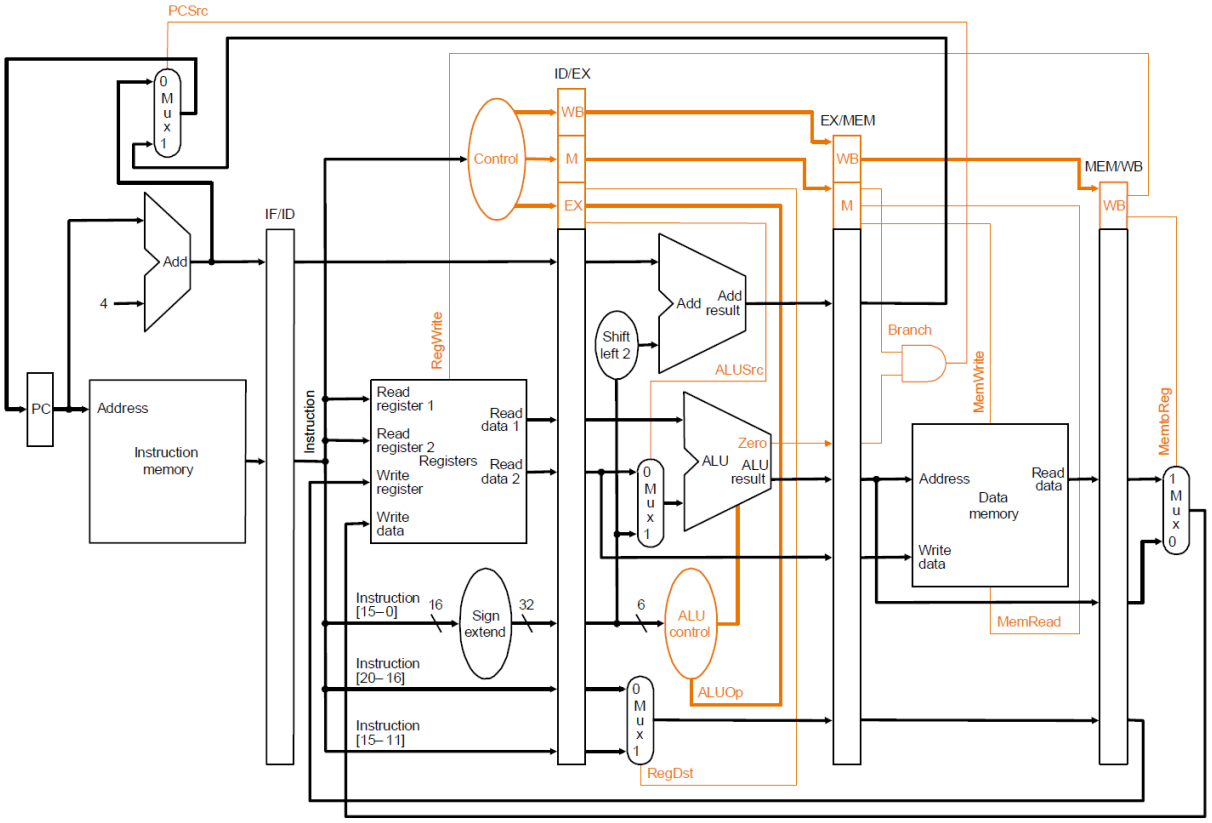


In an ideal 5-staged pipeline:

Cycle: Instr:	1	2	3	4	5	6	7	8	9	10	11	12	13
i	F	D	X	M	W								
i+1		F	D	X	M	W							
i+2			F	D	X	M	W						
i+3				F	D	X	M	W					
i+4					F	D	X	M	W				

- CPI may seem to be 5, but after the pipeline has been "filled", CPI actually = 1 (have 1 instruction complete per cycle)
- Time per cycle is $\frac{1}{5}$ plus some overhead
- We add *stage registers* in between stages; we pass down any info needed by a later stage through these registers, including data & control signals

Putting them together, a pipelined datapath + control looks like:



Pipeline Hazards

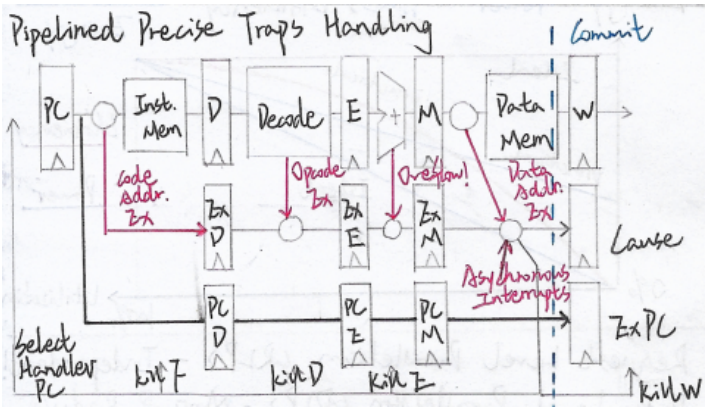
The 5 stages, however, are not completely independent. Sometimes, program instructions have dependences:

- True dependence (read after write, RAW)
- Anti-dependence (write after read, WAR)
- Output dependence (write after write, WAW)

In these situations, a later instruction cannot proceed until a previous instruction has finished its certain stage. These are called *pipeline hazards*:

1. *Structural hazard*: busy resources shared among stages
    - E.g., register file is accessed by many stages
    - Can always be solved by splitting/adding hardware:
        - Splitting instruction/data memory
        - Register file access halves to read + write in one clock cycle
2. *Data hazard*: data flow backwards, written reg used right after
    - E.g., an instruction following an `lw` uses the loaded word immediately
    - Can be solved by inserting a `nop` "bubble" to stall after the load
        - Can reorder code and put an unrelated instruction right after that load to avoid wasting that slot
    - Can be solved by explicitly *forwarding* the memory input to ALU input and add a mux to control which ALU input to use - this can handle most data hazards that are not memory loads
3. *Control hazard*: branch/jump must execute to determine which instruction to fetch next
    - Can be solved by inserting a `nop` "bubble" to stall after branch/jump
        - Can also reorder code and put an unrelated instruction there
    - Can predict which way we go and "flush" the pipeline when we know we were wrong (called *speculative* execution)

A pipelined datapath must also handle *exceptions* in each stage (e.g., illegal instruction, ALU overflow, invalid memory address, ...). A common design is to include exceptions in stage registers and commit only at M stage.



Revisit Amdahl's Law. If a pipeline has g fraction of time fully filled and $(1-g)$ fraction of time not fully filled (stalled), then even if we have infinite number of stages, a g slightly below 100% will significantly limit the upper bound of performance. Stalls must be minimized as much as possible.

Instruction-Level Parallelism (ILP)

With the emerging idea of *multi-core* processors, people start to utilize fine-grained instruction-level parallelism (ILP).

- In the 1980's, pipelining brought CPI from $5.0 \rightarrow 1.15$
- In the 1990's, superscalar brought CPI from $1.15 \rightarrow 0.5$ in best case

The idea of **superscalar** means to dispatch multiple instructions per cycle to achieve IPC > 1. It exploits *instruction-level parallelism* (ILP) and is done dynamically by the processor. Very Long Instruction Word (VLIW) relies on data parallelism and is done at compile time.

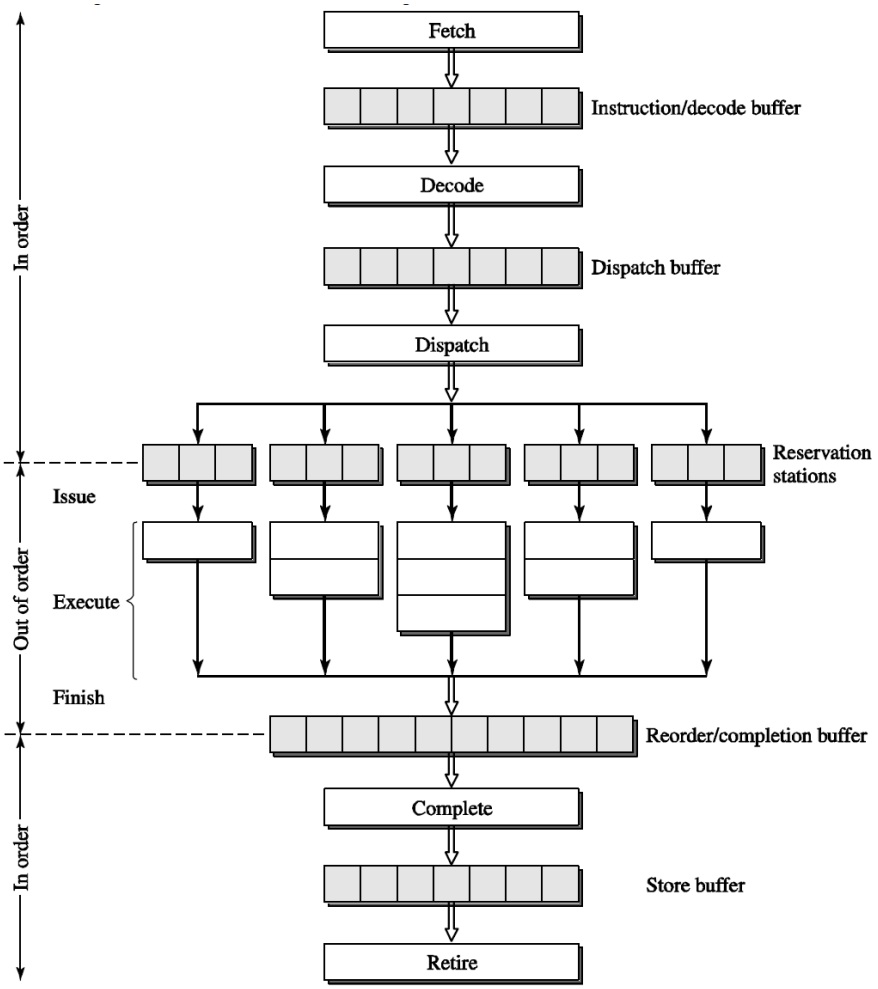
A classification of ILP machines by Jouppi & DECWRL can be found in Slide 11, page 20 -25:

1. Pipelined scalar processor (baseline)
2. *Superpipelined*: exploits parallelism further in time
3. *Superscalar*: exploits parallelism in space
4. *VLIW*: exploits static data parallelism in space
5. Superpipelined-superscalar: exploits both temporal & spatial parallelism!

Advantages of using superscalar pipelines:

- Paralleled pipelines theoretically allow CPI to go below 1
- Specialized pipelines solves the inefficiency of one single unified pipeline
- Enables *out-of-order execution* and *speculative execution*, improves the rigid stall policy

A dynamic superscalar processor architecture looks like:



- Upper section is *instruction flow*
  - Branch prediction
  - Fetch alignment
  - Instruction cache (I-cache) misses
- Middle section is out-of-order execution and involves *register data flow*
  - Register renaming to solve all RAW/WAR/WAW conflicts
- Bottom section is *memory data flow*
  - In-order stores for WAR/WAW
  - Store queue for RAW
  - Data cache (D-cache) misses

Cache Architecture

Off-chip DRAM accesses are very slow compared to on-chip SRAM or even registers. With each single memory access in a cycle going to an off-chip DRAM, our process will yield very poor performance. A table briefly comparing different types of storage from the perspective of the processor:

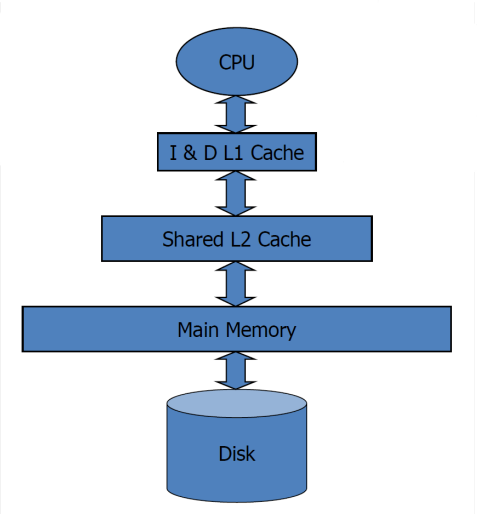
Type	Size	Speed	Cost/bit
Register	< 1KB	< 1ns	$$$$
On-chip SRAM	8KB-6MB	< 10ns	$$$
Off-chip SRAM	1Mb – 16Mb	< 20ns	$$
DRAM	64MB – 1TB	< 100ns	$
Disk	40GB – 1PB	< 20ms	~0

Memory Hierarchy

Good news is, memory accesses tend to have **locality**:

- *Temporal locality*: the recently accessed bytes are likely to be repeatedly accessed in a short period of time; E.g., a counter for a loop
- *Spacial locality*: Bytes near the recently accessed ones are likely to be accessed shortly; E.g., an array being looped

By building up a *memory hierarchy*, we *cache* recently accessed data in much smaller, much faster upper-level memory to boost up performance. The typical memory hierarchy looks like:



Some key issues we must solve when introducing caching:

- Hardware technology: how bits are stored in each layer?
- Placement: where are the bits stored and how to address them?
- Identification: how to maintain the upper-to-lower mapping?
- *Replacement (Eviction) policy*: what to evict for new entries if cache is full?
- *Write-allocation policy*: how to handle writes which will update the bits?

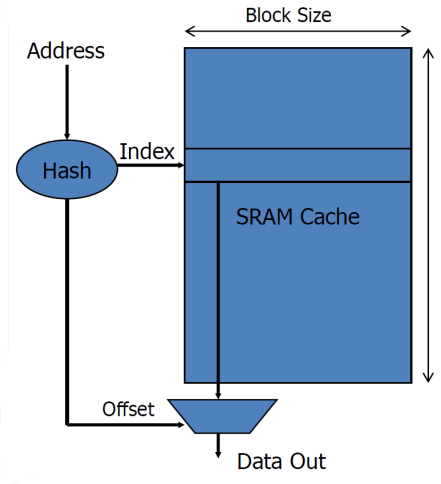
Address Mapping & Associativity

Consider an on-chip SRAM as a cache for an off-chip DRAM. The DRAM address space is much larger than the cache, so we need a way to *hash* (map) a DRAM physical address to some place on cache. Multiple DRAM addresses will map to the same cache address: they will then evict each other if necessary.

There are three types of cache address mapping:

1. *Direct-Mapping*: one extreme - a fixed region of physical address used as cache *index*
    - Physical address decomposition:

    - Cache layout:

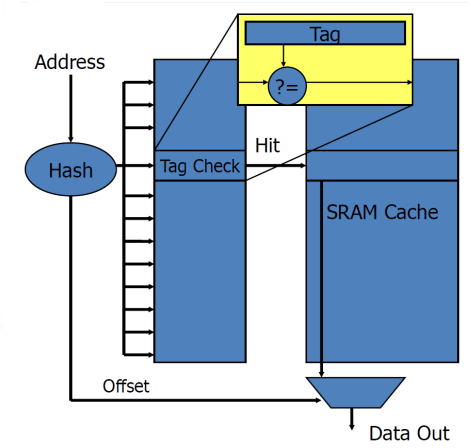


    - Properties:
        - Block size defines #bits for `offset`, e.g., 4 bytes blocks → 2-bits `offset`
        - Blocks in cache defines #bits for `index`, e.g., 256 blocks → 8-bits `index`

    - All addresses with the same `index` field will be mapped to the same *cache line*:
        - ↑ fastest translation
        - ↓ evictions will be common

2. *Fully-Associative*: the other extreme - a block can be mapped to any cache line
    - Physical address decomposition:

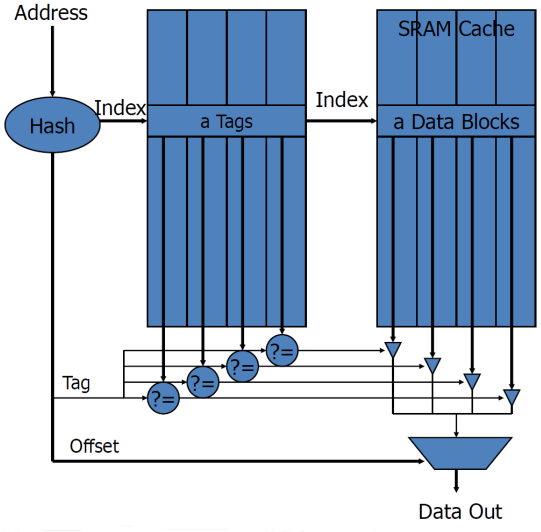
    - Cache layout:



    - Properties:
        - Aligned physical blocks can be mapped to any cache line, identified by its `tag`
            - ↑ the full cache space can be utilized, new entry can get in any empty slot
            - ↓ looking up & locating a block needs a linear-time tag comparison
            - ↓ extra space for storing long tags

3. *Multi-Way Set-Associative*: a hybrid solution
    - Physical address decomposition:

    - Cache layout:



    - Properties:
        - Blocks are first mapped by its `index`, and then each `index` on cache can hold multiple physical blocks, differentiated by the `tag`
        - Columns (how many cache lines for the same `index`) is called the number of *ways*
        - Rows (how many different indexes on the cache) is called the number of *sets*
        - Now, evictions are more flexible than direct-mapping, while tag comparison much faster (in-parallel) than fully-associative cache
            - Ways = 1 & #Sets = #cache-lines means direct-mapping
            - Sets = 1 & #Ways = #cache-lines means fully-associative
    - Some simple calculations:
        - Length of `offset` $o = \log_2(\text{block size})$

- Length of `index` $i = \log_2(\text{number of sets})$
- Length of `tag` $= 32 - o - i$

Replacement Policy

Cache replacement (eviction) policies are slightly out-of-scope of this course. They are researched in much more fancty details in lower-level caching (e.g., for disks), where caches are mostly fully-associative.

For high-speed processor caches, #ways are mostly small and we do not want to introduce large overheads for doing eviction caculations. So, only several choices are available:

- FIFO: simplest, does not always work well
- LRU: cheap and easy for fewer ways, works well in most cases
- NMRU (not most recently used): approximates LRU
- Pseudo-random: works pretty good for high-level caches, because locality does not always perfectly exist!

Write-Allocation Policy

This is again slightly out-of-scope of this course. Details can be found in OS materials & this blog post.

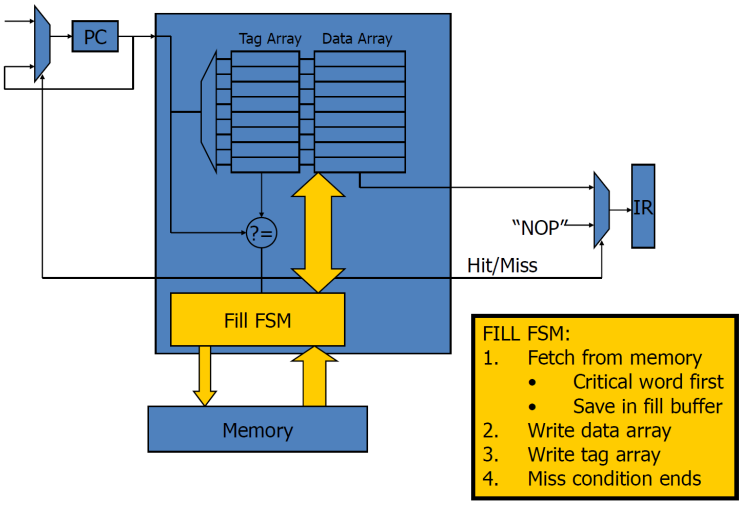
For high-level high-speed processor cache, we usually do *write-back* since its the fastest and we do not care about consistency / losing data - the DRAM itself is volatile anyway.

On multi-core processors or I/O devices with *direct memory access* (DMA), write-back policy will introduce the *cache coherence* problem - multiple cores access the same physical block when it has been cached *dirty* in one of the core's cache. We will get to that later.

Cache + Pipeline Integration

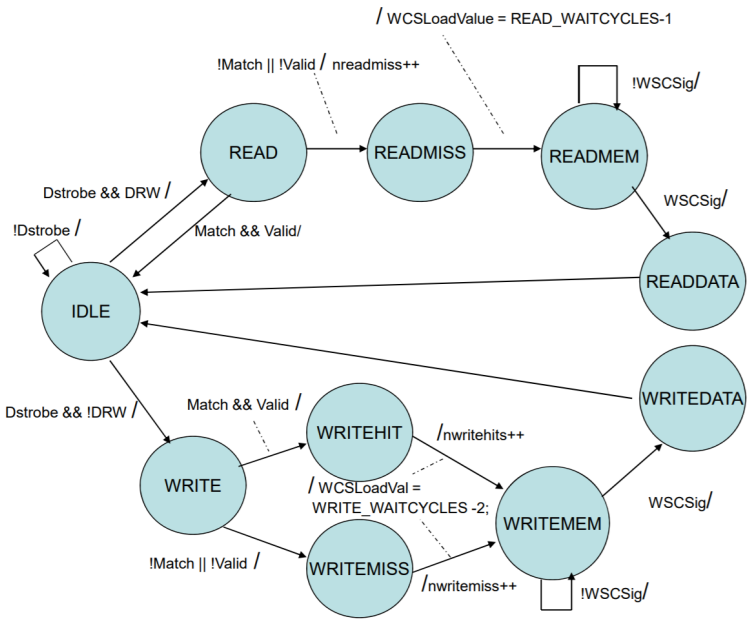
To integrate caches into the processor datapath pipeline, we often assume that a cache *hit* finishes within the cycle. However, when we have a cache *miss*, we have to stall the pipeline and wait for the slow off-chip DRAM access to fetch the instruction / data we want.

- Instruction cache (*I-cache*): read-only; stalls on a miss



- Data cache (*D-cache*):
  - Much more complicated - wires are too slow for sending a global stall from MEM stage back to IF stage; instead, we have to cancel/flush the pipeline on a miss
  - For cache writes, even more complicated:
    - Write-through + no write-allocation:
      - DRAM write proceeds in parallel with tag check
      - Invalidate an entry on write miss
      - Simplest implementation but bad performance
    - Using a *store buffer* (SB):
      - Store only performs a tag check in MEM stage, then `<val, addr, way-id>` placed in SB
      - When the next store comes to MEM stage, we write SB to data cache in the background
      - Loads now MUST first check SB; load misses should also flush SB first

The cache control state machine FSM looks like:



Cache Performance

Latency performance of processor cache can be calculated as:

$$L_{\text{high}} = \text{hit\_rate} \cdot L_{\text{high\_hit}} + (1 - \text{hit\_rate}) \cdot L_{\text{miss\_penalty}}$$

$$\text{CPI} = \frac{\text{cycles}_{\text{hit}}}{inst} + L_{\text{miss\_penalty}} \times \text{miss\_rate} = \frac{\text{cycles}_{\text{hit}}}{inst} + \sum_{\text{level } i=1}^{n} (L_{\text{miss\_penalty}_i} \times \text{global\_miss\_rate}_i)$$

$$\text{global\_miss\_rate}_k = \prod_{\text{level } i=1}^{k} \text{local\_miss\_rate}_i$$

- $L_{\text{high\_hit}}$ is the cache hit latency
  - Determined by the cache layout
  - Typically 1~3 cycles
- $\text{hit\_rate} = 1 - \text{miss\_rate}$ is the ratio of cache hits over all memory accesses
  - Determined by the workload
  - Affected by cache layout, eviction policy, and write policy as well
  - We expect hit rate to be high enough for the cache to actually bring benefit; Improving hit rate is a continuing topic in the computer science field
- $L_{\text{miss\_penalty}}$ is the *miss penalty*, which is very high
  - Composed of L_{low} + detect miss overhead + eviction overhead + ...
  - On multi-level hierarchy, L_{low} can be in a similar form - this is a recursive calculation; Hence, the sum in the second formula
  - Can be 6 cycles to 100 secs

Cache misses *3-C's* classification:

1. *Compulsory*: First-ever reference to a block
2. *Capacity*: Working set exceeds cache capacity, kicking off useful entry out
3. *Conflict*: (Capacity within set) On non-fully-associative caches, useful block kicked out by one with the same index

Other Topics

Some topics not included in this note:

- Virtual memory - shouuld be elaborated extensively in OS classes
- *Error-correcting code* (ECC) - see this note
- Multicore & shared memory - should be elaborated extensively in parallel computing classes
- Advanced hardware arithmetics - see this slide