1 CPU Power Consumption

Chip Power Consumtion $= C * Vdd^2 * F$
$C =$ capacitance, $Vdd =$ voltage, $F =$ frequency
Dennard scaling: Transistors get smaller but their power density stays the same. The supply voltage of a chip can be reduced by $0.7x$ every generation
$Power = C * Vdd^2 * F_{0\to1} + Vdd * I_{leakage}$
Leakage gets worse with samller devices and lower Vdd, it also gets worse with higher temps
Amdahl's Law: We want to make the common case efficient, given an optimization x that accerlerates fraction f_x of the program by a factor S_x, the overall speedup is: $\frac{1}{(1-f_x)+\frac{f_x}{S_x}}$

Performance

Latency: how long it takes to do a task
Throughput: total work done per unit time
$ExeTime = \frac{Instrs}{Program}\frac{Clockcycles}{Instr}\frac{Sec}{ClockCycle}$
Relative Performance: define performance as $= 1/ExecutionTime$ "X is n times faster than Y" means $Perormance_x/Performance_y$

2 Instruction Set Architechture

CISC	RISC
Emphasis on Hardware	Emphasis on Software
Multi-cycle complex instructions	Simple (single clock) instructions
Memory-to-Memory load/store incorporated in instr.	Register-to-Register Separate load/store instructions
Small code size	Large code size
High CPI	Low CPI
Low clock frequency	High clock frequency
Variable length instructions	Same length instructions
Complex instruction decode	Simple instruction decode
HW difficult to implement	HW easy to implement

R-type

funct7	rs2	rs1	funct3	rd	opcode

Opcode: basic operation of instruction (7) — Rs1: Register source 1 operand (5)
Rd: Register destination operand (5) — Rs2: register source 2 operand (5)
Funct3: additional opcode field (3) — Funct7: additional opcode field (7)

Question: Why did RISC-V only define 32 registers?

I-type Question: What is an immediate?

immediate[11:0]	rs1	funct3	rd	opcode

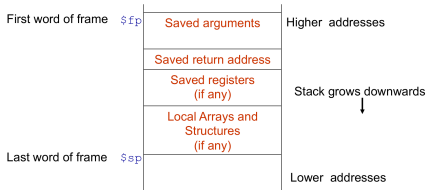
S/B-type Question: Why is the immediate split?

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

U/J-type

immediate[31:12]	rd	opcode

Constant == immediate == literal == offset
There are 32 registers in RISC-V **ld dst, offset(base)** the base is the starting address of the array the offset is the index
When using switch statement we can use a jump table, a jump table holds addresses in memory of where the code for the jump targets are.
Stack: The stack is a allocated in frames, it stores the state of a procedure for a limited time, the callee returns before the caller does. The things which can be saved on the stack are: local arrays, return addresses, saved registers, and nested call arguments



Callee Saved	Caller Saved
Saved registers ($s0-$s7)	Temporary registers ($t0-$t9)
Stack/frame pointer ($sp, $fp, $gp	Argument registers ($a0-$a3)
Return address ($ra)	Return values ($v0-$v1)

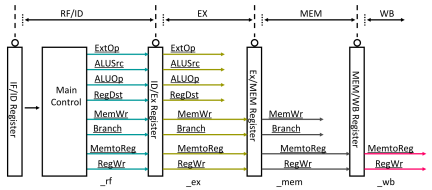
- Callee saved registers (preserved for caller)
  - Save register values on stack prior to use
  - Restore registers before return
- Caller saved registers (not preserved for caller)
  - Do what you please and expect callees to do likewise
  - Should be saved by the caller if needed after procedure call

Procedure Call Steps

1. Place parameters in a place where the procedure can access them
2. Transfer control to the procedure
3. Allocate the memory resources needed for the procedure
4. Perform the desired task
5. Place the result value in a place where the calling program can access it
6. Free the memory allocated in (3)
7. Return control to the point of origin

3 Pipelining

In pipelinging we overlap instructions in defferent stages



There are hazards, these include **structural hazards** where a required resource is busy, **data harzards** where we must wait previous instructions to produce/consume data, and **control hazards** where next PC depends on previous instruction.

Structural Hazards

two instructions are trying to use the same hardware within the same cycle, to solve this we can make all the instructions the same length

Data Dependencies

Dependencies for instruction j following instruction i

- Read after Write (RAW or true dependence)

  Instruction j tries to read before instruction i tries to write it

- Write after Write (WAW or output dependence)

  Instruction j tries to write an operand before i writes its value

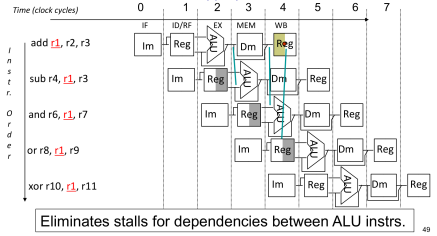
- Write after Read (WAR or (anti dependence))

  Instruction j tries to write a destination before it is read by i

Solutions for RAW Hazards: We can delay the reading of an instruction until data is available, to do this we can insert pipeline bubbles, can also write to the register file in the first half of a cycle and then read in the second half.

Forwarding: Another solution is forwarding or pushing the data to an appropriate unit.

We can also reorder instructions to deal with RAW hazards



- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd == ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd == ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

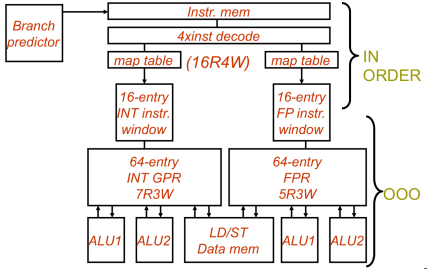
Control Hazards

A control hazards is like a data hazard on the PC, we cannot fetch the next instruction if we don't know the PC

Some solutions for control hazards are stalling on branches, predicting taken or not taken. We need to flush the pipeline if we predict wrong, in a 5-sage pipeline we only need to flush 1 instruction

4 Out of Order Execution and ILP

We want to avoid in-order stalls so we use out of order execution to re-order instructions based on dependencies



A **superscalar processor** is a CPU that implements a form of parallelism called instruction-level parallelism within a single processor. I.e we can launch multiple instructions every cycle.

There are some issues with multiple instructions executing at onc,e we need to double the amount of hardware, we introduce hazards, branch delay, & load delay

We can rename (map) architectural registers to physical registers in decode stage to get rid of false dependencies

Superscalar + Dynamic scheduling + register renaming

```
add  $t0_A, $t1, $t2    sub $t0_B, $t1, $t2
or   $t3, $t0_A, $t2    and $t5, $t0_B, $t2
```

There are some limits to **ILP** and pipelining:

- Limited ILP in real programs
- Pipeline overhead
- Branch and load delays exacerbated
- Clock cycle timing limits
- Limited branch prediction accuracy (85%-98%)
- Even a few percent really hurts with long/wide pipes
- Memory inefficiency
- Load delays + # of loads/cycle