Introduction

EE599-001 & EE699-010, Spring 2026

Hank Dietz

http://aggregate.org/hankd/

University of Kentucky

Course Overview

- You'll be learning about quantum computing, which is not yet a fully-developed technology
- This is a computer engineering course
  - You have some exposure to digital logic
  - You have done conventional programming
  - No quantum physics understanding required

Major Topics

- Course overview and introduction. Moore's law. The relationship between parallel and quantum computing. What is a quantum computer? Why do we care? Quantum computers as attached accelerators.
- Review of classical digital logic gates. Introduction to reversible logic. Ancilla. Logic optimization with and without fanout.
- Quantum superposition, entanglement, and measurement. Notations and models including the Bloch sphere, unitary matrix formulations, and wave models. Probability amplitude. The quantum circuit model and quantum gates.
- Quantum circuits and algorithms, the use of interference. Quantum superiority and RCS (Random Circuit Sampling). Algorithms including Deutsch-Jozsa, Grover's search, Simon's problem, QFT (Quantum Fourier Transform), and Shor's factoring algorithm.
- Quantum annealing.
- NISQ (Noisy Intermediate Scale Quantum) computer implementations. Superconducting, trapped ion, photonics, spins, etc. Qubit implementations. Qubit layout and operation scheduling issues. Error correction.
- Quantum-inspired methods. Randomized and probabilistic algorithms. Alternative representations for entangled superpositions: PBP (Parallel Bit Pattern computing, developed here at UK).
- The architecture of a computer system with an attached quantum coprocessor.
- Quantum programming above the Qubit level.
- Where is quantum going? Cryptography, machine learning, networking and teleportation. Scaling.

Schedule (very tentative!)

Lectures	Topic
3	Introduction, parallel and quantum
4	Classical & adiabatic logic (project)
3	Quantum concepts and models
3	Quantum circuits and algorithms (project)
1	Quantum annealing
1	*Midterm exam*
3	NISQ computer implementations (homework)
4	Quantum-inspired methods (project)
2	Architecture of computer with quantum coprocessor
2	Higher-level quantum programming (homework)
2	Where is quantum going?
2	*reserved for schedule slippage*
1	Review for final exam

Schedule Notes

- Homeworks/projects will involve some programming using simulation environments (probably Verilog, Qiskit, and C++)
- Some topics may be given more or less time depending on how students are doing
- I will be presenting research at the IS&T Electronic Imaging conference, so we will not have regular class meetings 3/3 & 3/5

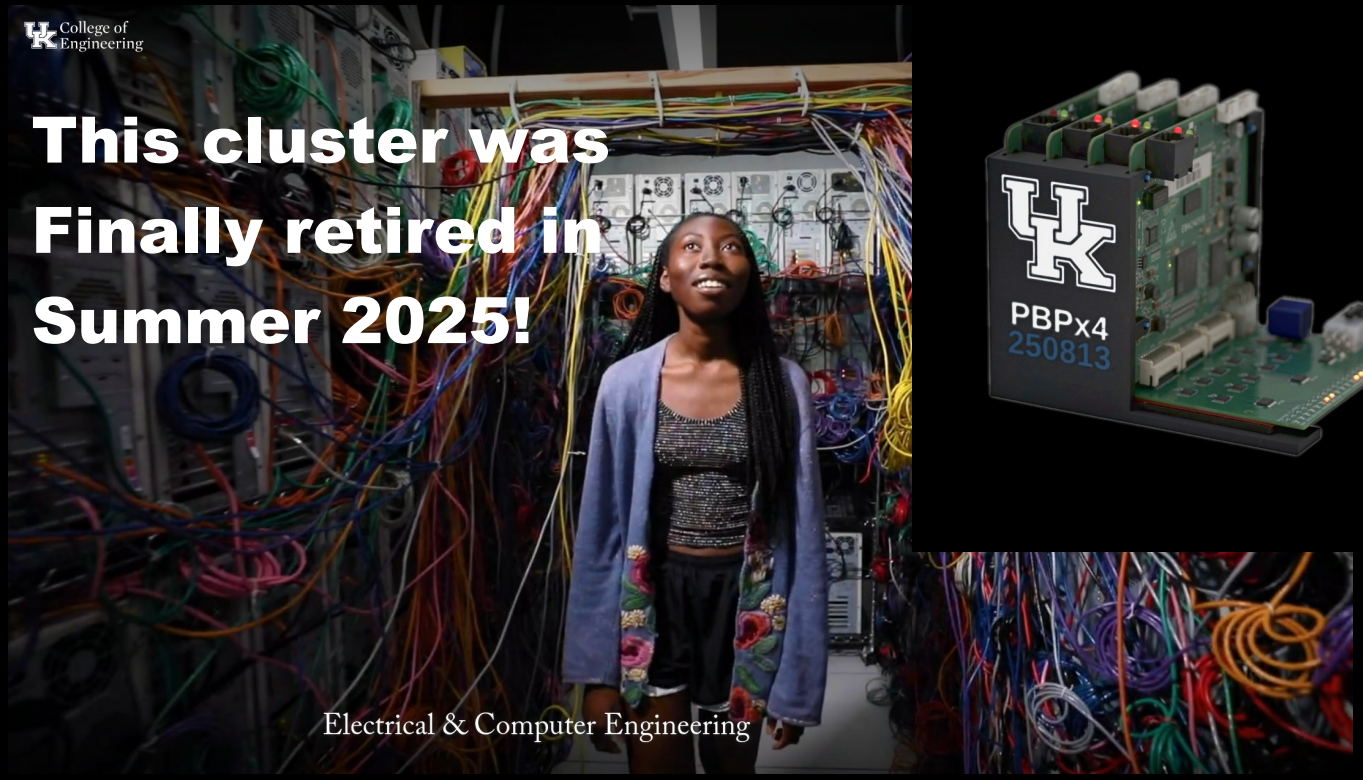
Grading & Such

- Several homeworks/projects, total 50%
  (a little extra on these for grad credit)
- In-person **Midterm exam**, 20%
- In-person **Final exam**, 30%
- Material from lectures, canvas, & course URL:
  `http://aggregate.org/QC`
- You are expected to **regularly attend class**
- I try not to curve much; always in your favor

Me (and why I'm biased)

- Hank Dietz, ECE Professor and
  James F. Hardymon Chair in Networking
- Office: 203 Marksbury
- Research in parallel compilers & architectures:
  - Built 1st Linux PC cluster supercomputer
  - Antlr, AFNs, SWAR, FNNs, MOG, ...
  - Various awards & world records for best price/performance in supercomputing
- Lab: 108/108A Marksbury – I have **TOYS**!


KREQC
Kentucky's Rotationally Emulated Quantum Computer
SEE WHAT'S
Wildly Possible
Power Supply
Control Logic
KREQC
Wave Function Generator


College of Engineering
This cluster was Finally retired in Summer 2025!
Electrical & Computer Engineering

PBPx4
250813


UK ECE

Am I A Quantum Expert?

- **Nope.** This is my 1st time teaching this…
- Two main aspects to quantum computing:
  - Quantum physics: many A+ folks, I'm C-
  - Computer engineering: most D or F, I'm A+
- My background in QC:
  - Many conferences, 1st QC publication in 2017
  - Built optimizing compilers for QC
  - Designed *quantum-inspired* HW/SW
  - Committee member for several QC PhDs

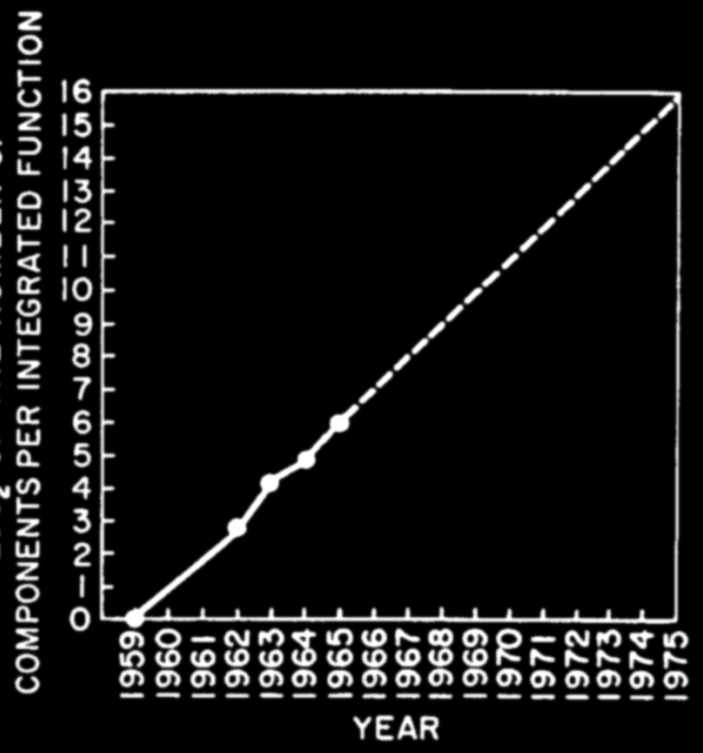
What Is This Course About?

- Using quantum phenomena in clever ways.
- Solving computational problems faster.†
- Using fewer resources while doing that.

† Especially problems that supercomputers can't feasibly solve.

How Computers Get Faster: Moore's Law

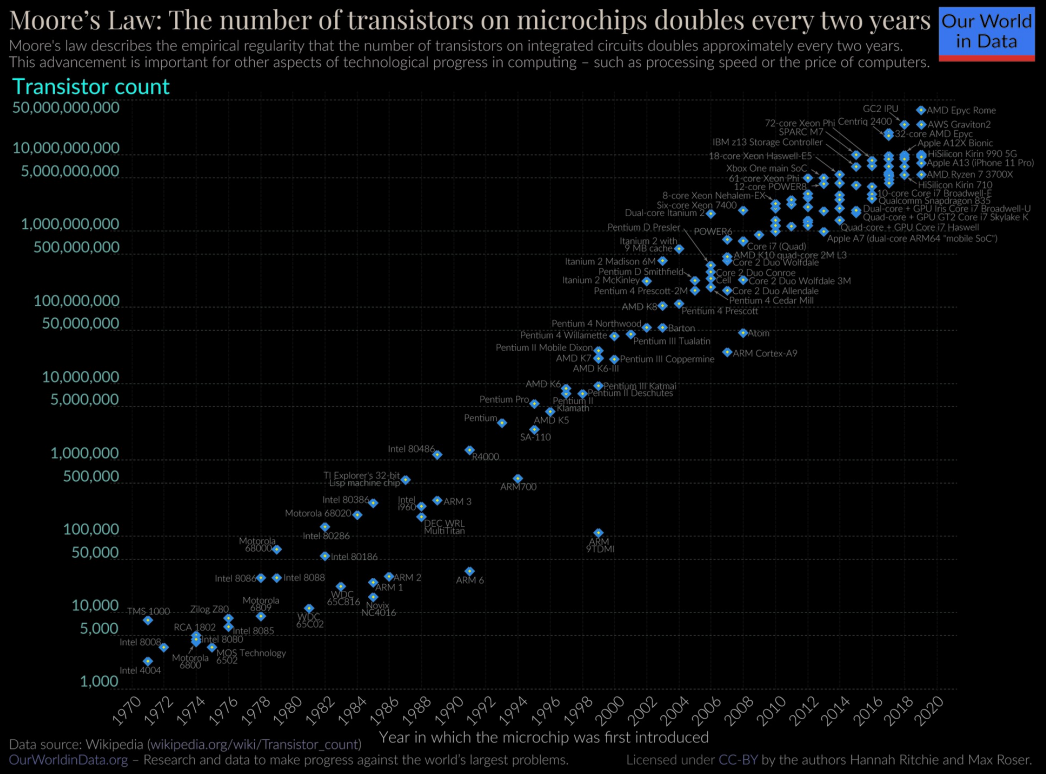
- 1965 prediction
  - Not about chip speed
  - Circuit complexity 2X every 18-24 months
- **Speedup** is mostly about **parallel processing**

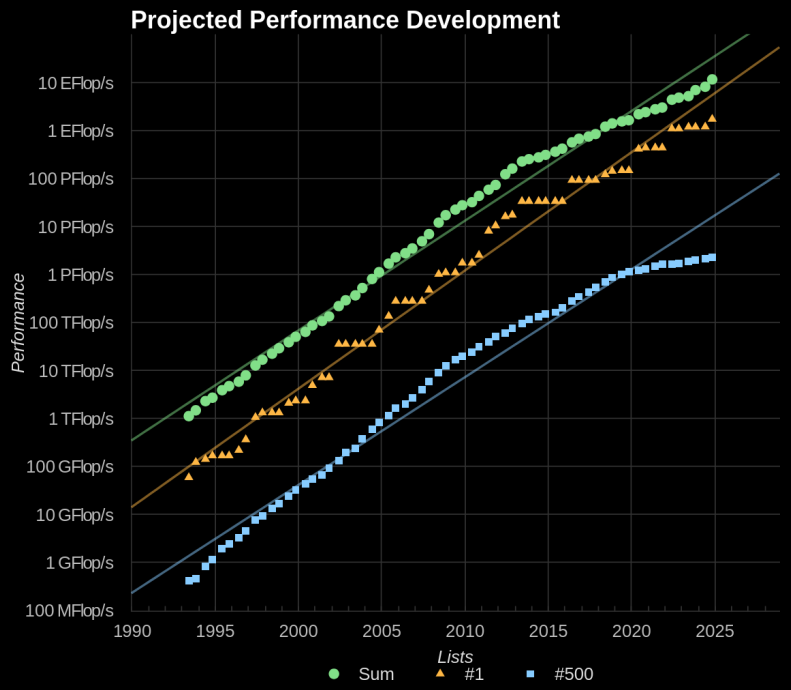


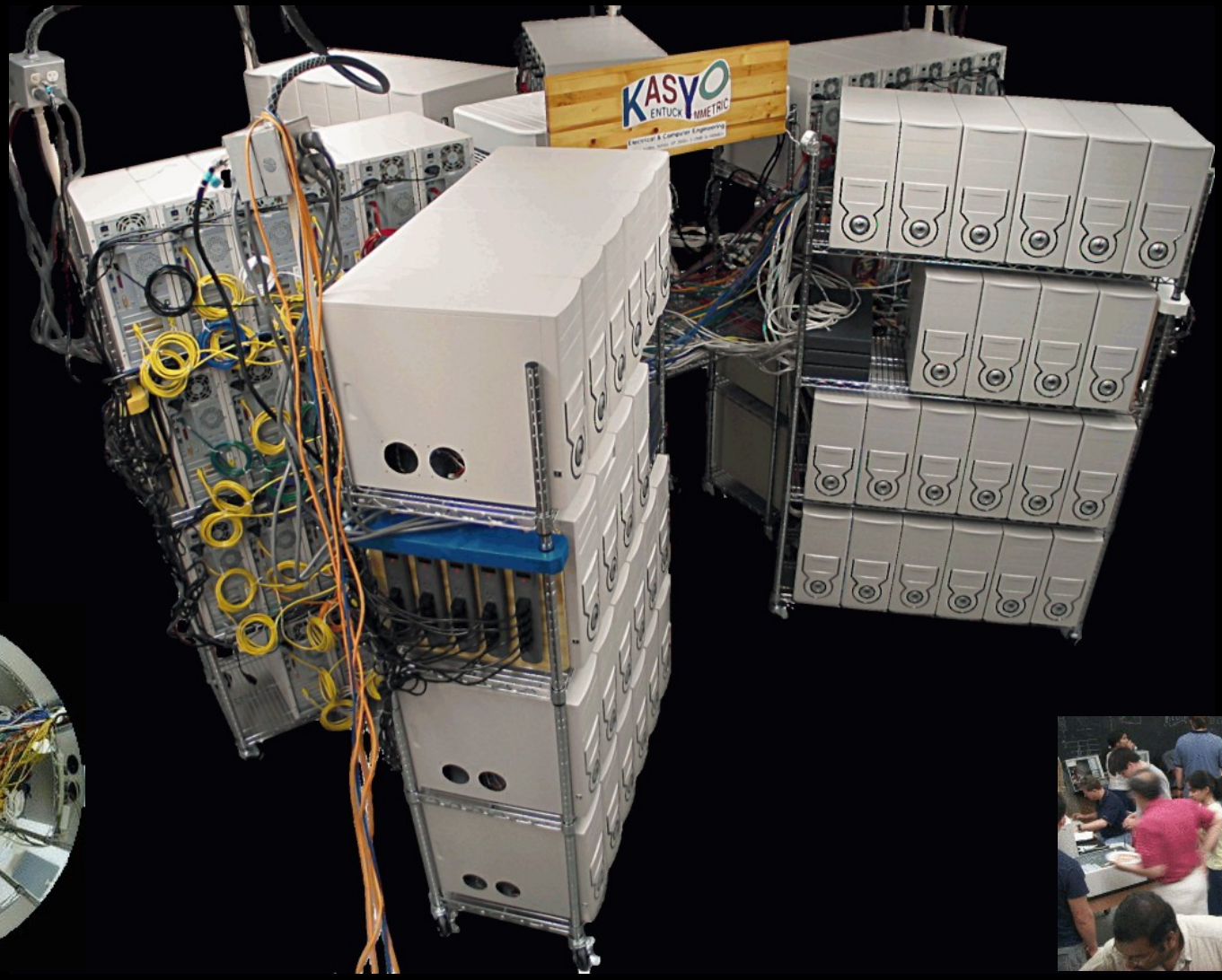
Parallel Processing

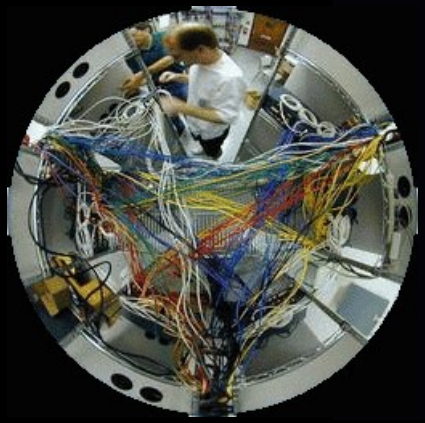
- Break program into ***N*** pieces that can execute simultaneously
  - **Scalable**: bigger ***N***, more speedup
  - **Modular hardware**
  - Can be **fault tolerant** using redundancy
- This scales up **forever**, **right?**

Moore’s Law is still sort-of OK…
(using tricks like multichip modules)









Kentucky ASYmmetric Zero supercomputer
Built July 16, 2003; 1st Teraflop/s system in KY
Cost $39K, **25kW + 5 Ton** air conditioner

El Capitan supercomputer:
11,039,616 cores, 2.821 Exaflop/s
Cost approx. $600M, **29.7 MW power**

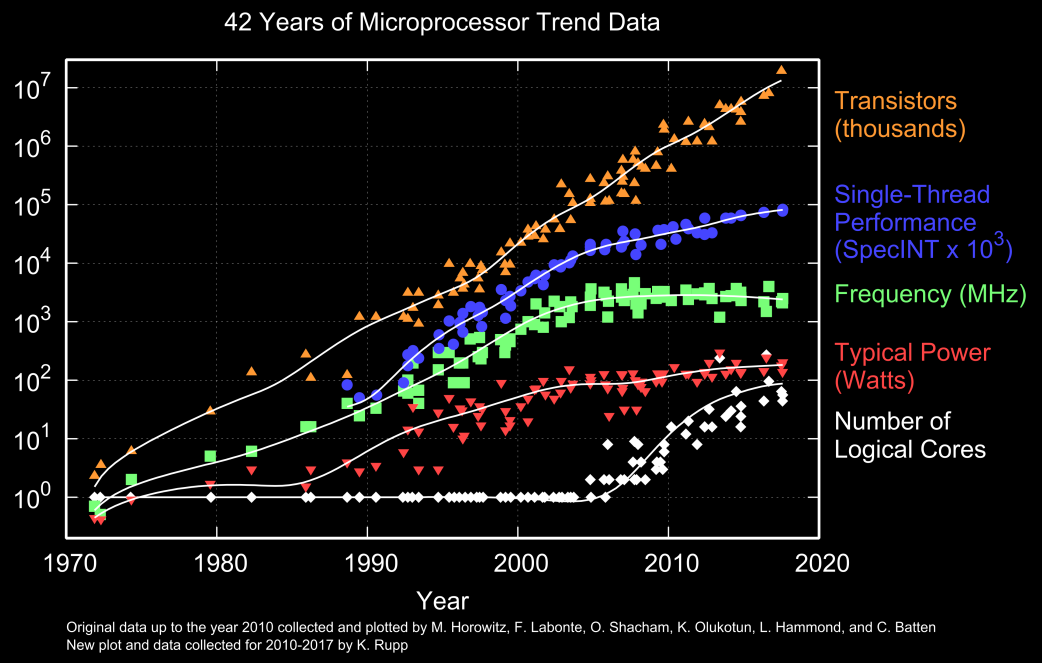
SI Terminology of Scale

1000^1	kilo	k	1000^-1	milli	m
1000^2	mega	M	1000^-2	micro	u
1000^3	giga	G	1000^-3	nano	n
1000^4	tera	T	1000^-4	pico	p
1000^5	peta	P	1000^-5	femto	f
1000^6	exa	E			

- 1000^x vs. *1024^x ;* e.g., giga vs. *giba*
- 1 Byte (B) is 8-10 bits (b), 4 bits in a Nybble
- Hertz (Hz) is frequency (vs. period)
- A flop is a floating-point op like add or multiply

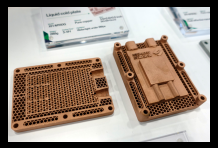
The Bad News

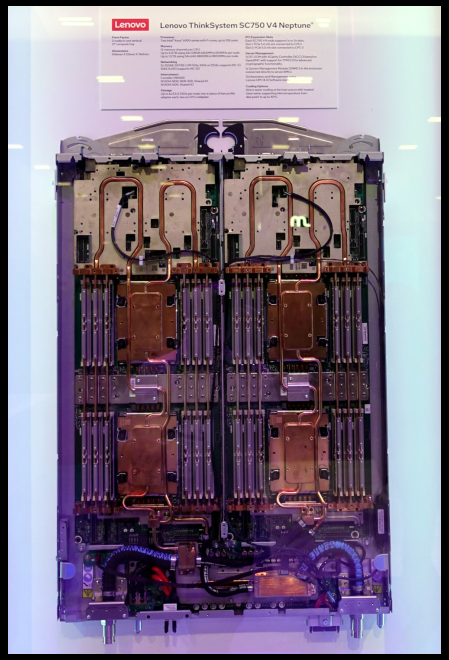
- Moore's Law **is slowing**
- Power/transistor ▼ **slower than** transistors/chip ▲
- **Individual ops not getting much faster**



Power & Heat

- Chips can have **dark silicon**, special-purpose function units normally unpowered, but…
- Remember my machine room? It has **170kW** and **30 Tons** of air conditioning, but there are *individual racks* it couldn't power & cool!
- **Colossus**, xAI's Memphis, TN supercomputer, in April 2025 had **422MW** gas turbine power!
- Exotic water cooling systems dominated SC25!




Lenovo
Lenovo ThinkSystem SC750 V4 Neptune


Valvoline
Hypertec


Secondary Side
OCP
Project Deschutes
Open CDU
DELTA | Google
Specifications:
• 2 MW @ 3 °C ATD
• 500 GPM (1890 LPM)
• 80 PSI available pressure
• N+1 sealless pumps
• Fully redundant power feeds for each pump circuit
• IEEE 519 ULHD variable frequency drives
• 230 VAC convenience port for fill cart
• 0.2 µm side stream filtration
DELTA | Google


Modular 1 MW OCP Cluster


NOMAD

Amdahl's Law

- In the best case, as parallelism width $N \rightarrow \infty$, runtime should go to 0, **right?**
- If *1/K* of the program's work ***cannot*** be run in parallel, the best possible speedup is only *K*

If a program spends 10% of its time sequentially reading a input file, you can't get more than a 10X speedup by parallelizing just the other 90%!
You need to parallelize everything possible.

Flavors of Parallel Processing

- Bit-parallel within a word
- Pipelined
- Superscalar, VLIW, EPIC
- SWAR (SIMD Within A Register), Vector
- SMP (Symmetric MultiProcessor; multi-core)
- GPU (Graphics Processing Unit)
- Clusters, Farms, Grids, and Clouds

Automatic; Semi-Auto; Explicitly Programmed

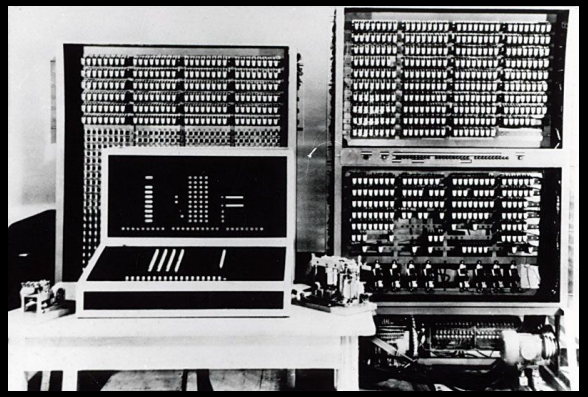
Flavors of Parallel Processing

- Bit-parallel within a word
- Pipelined
- Superscalar, VLIW, EPIC
- SWAR (SIMD Within A Register), Vector
- SMP (Symmetric MultiProcessor; multi-core)
- GPU (Graphics Processing Unit)

⇒ Quantum

- Clusters, Farms, Grids, and Clouds

Automatic; Semi-Auto; Explicitly Programmed



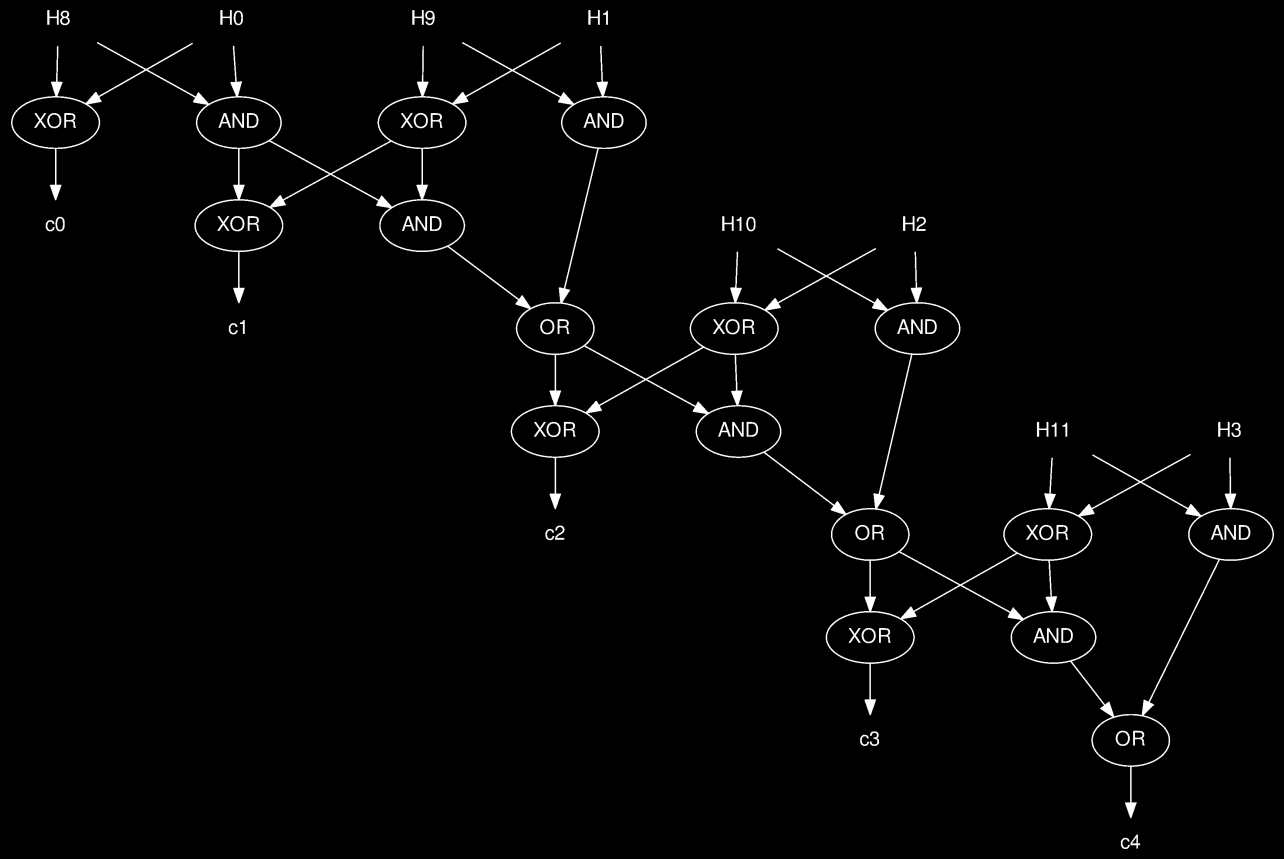
Bit-parallel within a word

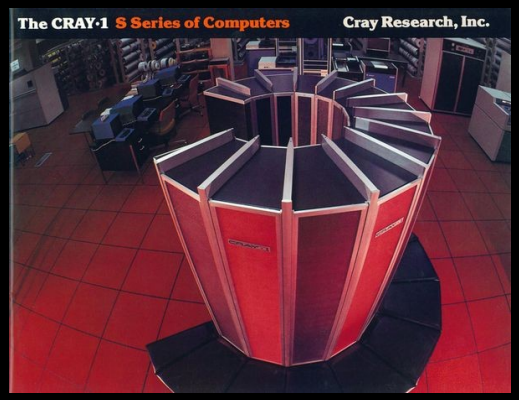
- Used by most computers (even Z3, ENIAC)
- Each machine has a word size
- Each operation produced a word result, with many bit-level operations happening in parallel; e.g., `A&B` *simultaneously* $\texttt{A}_0\texttt{\&B}_0$, $\texttt{A}_1\texttt{\&B}_1$, ...

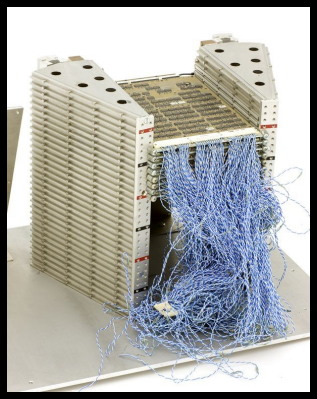
Word-Level Parallelism

- Consider adding 4-bit integers:

```
c = a + b
```






Vector parallelism across words

- Used by Cray 1, many others from the 1970s
- Applies same operation to all vector elements:
  `A[0..N-1] = B[0..N-1] + C[0..N-1]`
- A SIMD (Single Instruction, Multiple Data) model, but often implemented as a pipeline

Vector Pipeline Processing

- Consider the previous example:
  `A[0..N-1] = B[0..N-1] + C[0..N-1]`
- Suppose addition takes 3 clock cycles:

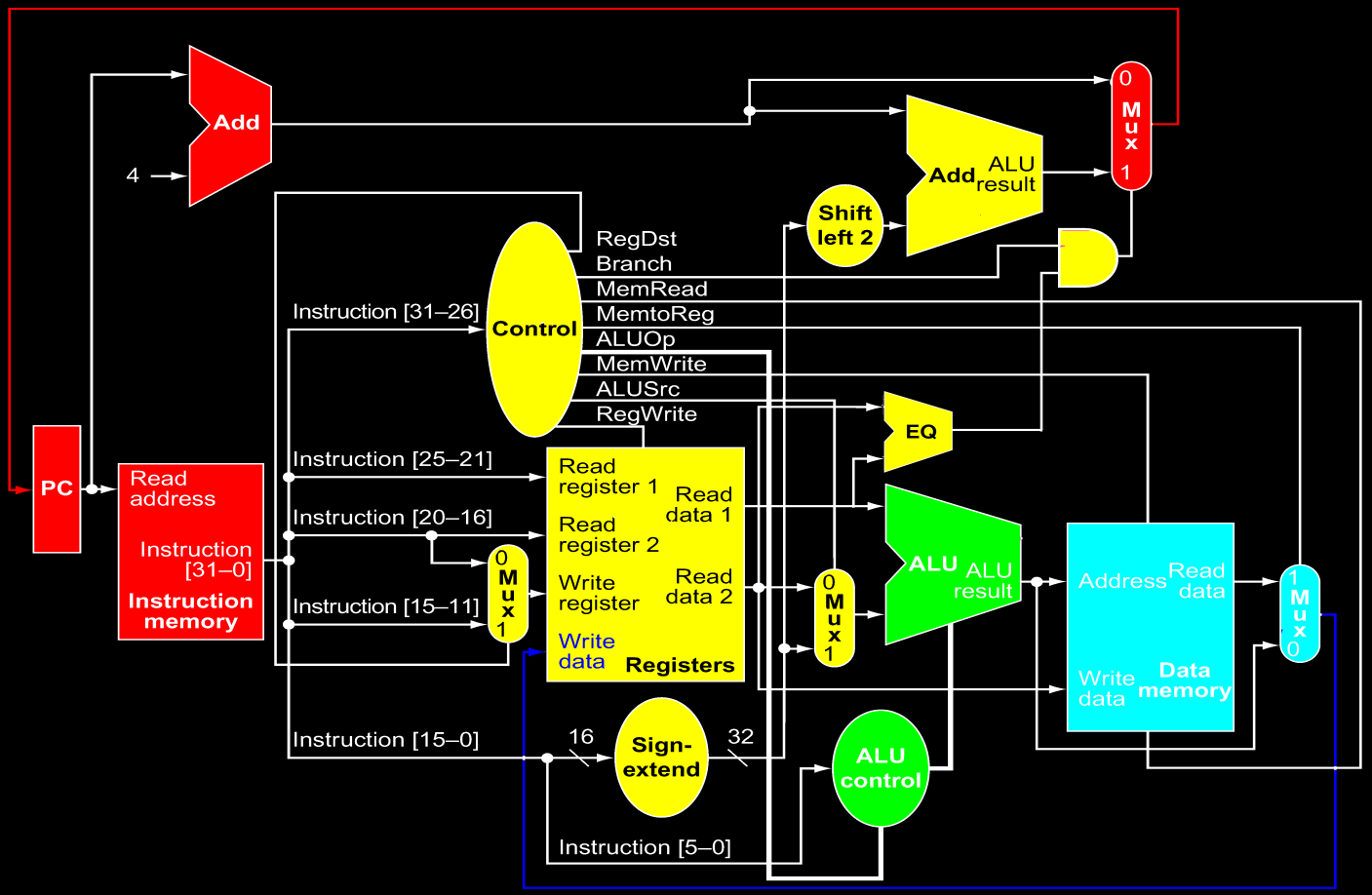
	Stage 0	Stage 1	Stage 2
Cycle 0:	`B[0]+C[0]`		
Cycle 1:	`B[1]+C[1]`	`B[0]+C[0]`	
Cycle 2:	`B[2]+C[2]`	`B[1]+C[1]`	`A[0]=B[0]+C[0]`
Cycle 3:	`B[3]+C[3]`	`B[2]+C[2]`	`A[1]=B[1]+C[1]`
Cycle 4:	`B[4]+C[4]`	`B[3]+C[3]`	`A[2]=B[2]+C[2]`

Modern Pipeline Processing

- Consider another example:
  `add a,b; sub c,d; mul e,a; xor f,a`
- 3-stage pipe (with bubble):

	Stage 0	Stage 1	Stage 2
Cycle 0:	`add a,b`		
Cycle 1:	`sub c,d`	`add a,b`	
Cycle 2:	`mul e,a`	`sub c,d`	`a = add a,b`
Cycle 3:	`mul e,a`	`nop`	`c = sub c,d`
Cycle 4:	`xor f,a`	`mul e,a`	`nop`

Where Is The Circuitry?



IF: Instruction Fetch EX: Execute WB: Write Back
ID: Instruction Decode MEM: Memory Access

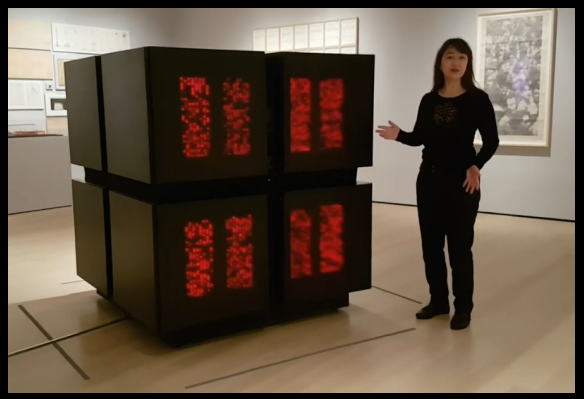
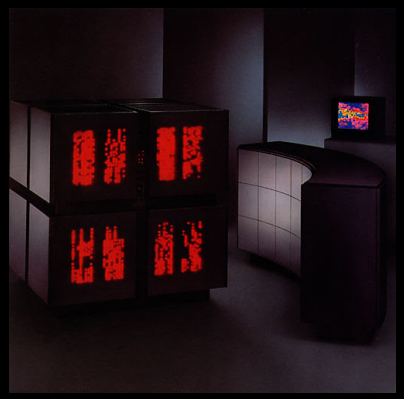
Where Is The Circuitry?

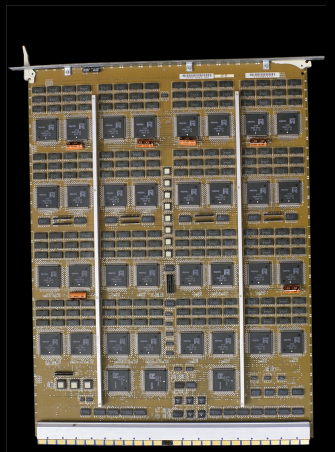
```
// Testbench options
`define RUNTIME 200     // How long simulator can run
`define CLKDEL 1        // Clock transition delay
`define TRACE   1       // enable simulation trace

// Types
`define WORD    [31:0]  // size of a data word
`define ADDR    [31:0]  // size of a memory address
`define INST    [31:0]  // size of an instruction
`define REG     [4:0]   // size of a register number
`define REGCNT  [31:0]  // register count
`define MEMCNT  [511:0] // memory count implemented
`define OPCODE  [5:0]   // 6-bit opcodes
`define ALUOP   [3:0]   // Simplified ALU codes

// Fields
`define OP      [31:26] // opcode field
`define RS      [25:21] // rs field
`define RT      [20:16] // rt field
`define RD      [15:11] // rd field
`define IMM     [15:0]  // immediate/offset field
`define SHAMT   [10:6]  // shift amnount
`define FUNCT   [5:0]   // function code (opcode extension)
`define JADDR   [25:0]  // jump address field
`define JPACK(R,O,J)        begin R`OP=O; R`JADDR=J; end
`define IPACK(R,O,S,T,I)    begin R`OP=O; R`RS=S; R`RT=T; R`IMM=I; end
`define RPACK(R,S,T,D,SH,FU)    begin R`OP=`RTYPE; R`RS=S; R`RT=T; R`RD=D; R`SHAMT=SH; R`FUNCT=FU; end
`define NOP     `OR     // Mult OPeration is or $0,$0,$0

// Instruction encoding
`define RTYPE   6'h00   // OP field for all RTYPE instructions
`define BEQ     6'h04   // OP field
`define ADDIU   6'h09   // OP field
`define SLTIU   6'h0b   // OP field
`define ANDI    6'h0c   // OP field
`define ORI     6'h0d   // OP field
`define XORI    6'h0e   // OP field
`define LUI     6'h0f   // OP field
`define LW      6'h23   // OP field
`define SW      6'h2b   // OP field
`define ADDU    6'h21   // FUNCT field
`define SUBU    6'h23   // FUNCT field
`define AND     6'h24   // FUNCT field
`define OR      6'h25   // FUNCT field
`define XOR     6'h26   // FUNCT field
`define SLTU    6'h2b   // FUNCT field

// Simplified ALU codes, default to lui
`define ALUADD  4'b0000
`define ALUSUB  4'b0001
`define ALUAND  4'b0010
`define ALUSUB  4'b0110
`define ALUSLT  4'b0111
`define ALULUI  4'b1000
`define ALUXOR  4'b1111

// Generic multi-cycle processor
module processor(halt, reset, clk);
output reg halt;
input reset, clk;
reg `WORD m `MEMCNT;
reg `WORD r `REGCNT;

// Initialize register file and memory
initial begin
    r[1] = 22; r[2] = 1; r[3] = 42;
    r[4] = 601; r[5] = 13013; r[6] = -1;
    `RPACK(m[0], 2, 3, 1, 0, `ADDU)
    `RPACK(m[1], 2, 3, 1, 0, `SLTU)
    `RPACK(m[2], 3, 5, 1, 0, `AND)
    `RPACK(m[3], 5, 3, 1, 0, `OR)
    `RPACK(m[4], 3, 5, 1, 0, `XOR)
    `RPACK(m[5], 4, 3, 1, 0, `SUBU)
    `IPACK(m[6], `ADDIU, 3, 1, -1)
    `IPACK(m[7], `SLTIU, 5, 1, 12345)
    `IPACK(m[8], `ANDI, 3, 1, 3)
    `IPACK(m[9], `ORI, 3, 1, 3)
    `IPACK(m[10], `XORI, 3, 1, 3)
    `IPACK(m[11], `LUI, 0, 1, 3)
    `IPACK(m[12], `LW, 2, 1, 1023)
    `IPACK(m[13], `SW, 0, 2, 1024)
    `IPACK(m[14], `ADDIU, 0, 0, 1)
    `IPACK(m[15], `BEQ, 0, 0, -2)
    n[16] = 0;
    n[256] = 22;
end

// IF registers
reg `ADDR IF_pc;
reg `INST IF_ir;

// ID registers
reg RegDst, Branch, MemRead, MemWrite, ALUSrc, RegWrite, Bad, ID_MemRead, ID_MemWrite, ID_Bad;
reg `ALUOP ALUOp, ID_ALUOp;
reg `WORD s, t, imm, ID_s, ID_t, ID_src;
reg `REG ID_rd;
reg `ADDR target;
reg squash;

// EX registers
reg EX_MemRead, EX_MemWrite, EX_Bad;
reg `WORD alu, EX_alu, EX_t;
reg `REG EX_rd;

// MEM registers
reg MEM_Bad;
reg `WORD v, MEM_v;
reg `REG MEM_rd;
```

```
// Running state?
wire running;
assign running = (!halt) && (!reset);

// Squash instruction fetched on a mispredicted branch
wire `INST squashed;
assign squashed = (squash ? `NOP : IF_ir);

// Are we blocked by a dependence on rs or rt?
// Also handles all forwarding
wire EX_dep1, EX_dep2, MEM_dep1, MEM_dep2, WB_dep1, WB_dep2, canfwds, canfwdt, dep, blocked;
wire `WORD fwds, fwdt;
assign ID_needs = (IF_ir `RS != 0);
assign ID_needt = (IF_ir `RT != 0);
assign EX_dep1 = (IF_ir `RS == ID_rd);
assign EX_dep2 = (IF_ir `RT == ID_rd);
assign MEM_dep1 = (IF_ir `RS == EX_rd);
assign MEM_dep2 = (IF_ir `RT == EX_rd);
assign WB_dep1 = (IF_ir `RS == MEM_rd);
assign WB_dep2 = (IF_ir `RT == MEM_rd);
assign canfwds = ID_needs && ((EX_dep1 && !MemRead) || ((!EX_dep1) && (MEM_dep1 || WB_dep1)));
assign canfwdt = ID_needt && ((EX_dep2 && !MemRead) || ((!EX_dep2) && (MEM_dep2 || WB_dep2)));
assign fwds = (EX_dep1 ? alu : (MEM_dep1 ? v : MEM_v));
assign fwdt = (EX_dep2 ? alu : (MEM_dep2 ? v : MEM_v));
assign dep = ((ID_needs && (EX_dep1 || MEM_dep1 || WB_dep1)) ||
              (ID_needt && (EX_dep2 || MEM_dep2 || WB_dep2)));
assign blocked = (dep && (((ID_needs && !canfwds) || (ID_needt && !canfwdt))));

// IF: Instruction Fetch stage
always @(posedge clk) if (running && (blocked)) begin
  IF_ir <= n[(squash ? target : IF_pc) >> 2];
  IF_pc <= (squash ? target : IF_pc) + 4;
end

// ID: Instruction Decode stage
always @(posedge clk) if (running && !ID_Bad) begin
  if (blocked) begin
    ID_s <= 0;
    ID_t <= 0;
    ID_src <= 0;
    ID_rd <= 0; // not writing
    ID_MemRead <= 0; // not reading
    ID_ALUOp <= 0;
    ID_MemWrite <= 0; // not storing
    ID_Bad <= 0;
  end else begin
    // Brute force decode of instruction
    case (squashed `OP)
      `RTYPE: begin
        case (squashed `FUNCT)
          `ADDU:   begin RegDst=1; Branch=0; MemRead=0; ALUOp=`ALUADD; MemWrite=0; ALUSrc=0; RegWrite=1; Bad=0; end
          `SUBU:   begin RegDst=1; Branch=0; MemRead=0; ALUOp=`ALUSUB; MemWrite=0; ALUSrc=0; RegWrite=1; Bad=0; end
          `AND:    begin RegDst=1; Branch=0; MemRead=0; ALUOp=`ALUAND; MemWrite=0; ALUSrc=0; RegWrite=1; Bad=0; end
          `OR:     begin RegDst=1; Branch=0; MemRead=0; ALUOp=`ALUOR;  MemWrite=0; ALUSrc=0; RegWrite=1; Bad=0; end
          `XOR:    begin RegDst=1; Branch=0; MemRead=0; ALUOp=`ALUXOR; MemWrite=0; ALUSrc=0; RegWrite=1; Bad=0; end
          `SLTU:   begin RegDst=1; Branch=0; MemRead=0; ALUOp=`ALUSLT; MemWrite=0; ALUSrc=0; RegWrite=1; Bad=0; end
          default: begin RegDst=0; Branch=0; MemRead=0; ALUOp=`ALUOR;  MemWrite=0; ALUSrc=0; RegWrite=0; Bad=1; end
        endcase
      end
      `BEQ:    begin RegDst=0; Branch=1; MemRead=0; ALUOp=`ALUSUB; MemWrite=0; ALUSrc=0; RegWrite=0; Bad=0; end
      `ADDIU:  begin RegDst=0; Branch=0; MemRead=0; ALUOp=`ALUADD; MemWrite=0; ALUSrc=1; RegWrite=1; Bad=0; end
      `SLTIU:  begin RegDst=0; Branch=0; MemRead=0; ALUOp=`ALUSLT; MemWrite=0; ALUSrc=1; RegWrite=1; Bad=0; end
      `ANDI:   begin RegDst=0; Branch=0; MemRead=0; ALUOp=`ALUAND; MemWrite=0; ALUSrc=1; RegWrite=1; Bad=0; end
      `ORI:    begin RegDst=0; Branch=0; MemRead=0; ALUOp=`ALUOR;  MemWrite=0; ALUSrc=1; RegWrite=1; Bad=0; end
      `XORI:   begin RegDst=0; Branch=0; MemRead=0; ALUOp=`ALUXOR; MemWrite=0; ALUSrc=1; RegWrite=1; Bad=0; end
      `LUI:    begin RegDst=0; Branch=0; MemRead=0; ALUOp=`ALULUI; MemWrite=0; ALUSrc=1; RegWrite=1; Bad=0; end
      `LW:     begin RegDst=0; Branch=0; MemRead=1; ALUOp=`ALUADD; MemWrite=0; ALUSrc=1; RegWrite=1; Bad=0; end
      `SW:     begin RegDst=0; Branch=0; MemRead=0; ALUOp=`ALUADD; MemWrite=1; ALUSrc=1; RegWrite=0; Bad=0; end
      default: begin RegDst=0; Branch=0; MemRead=0; ALUOp=`ALUOR;  MemWrite=0; ALUSrc=0; RegWrite=0; Bad=1; end
    endcase

    s = (canfwds ? fwds : r[squashed `RS]);
    t = (canfwdt ? fwdt : r[squashed `RT]);
    imm = {{16{squashed[15]}}, squashed `IMM};
    target <= IF_pc + (imm[29:0], 2'b00);

    squash <= (Branch && (s == t));
    ID_s <= s;
    ID_t <= t;
    ID_src <= (ALUSrc ? imm : t);
    ID_rd <= (RegWrite ? (RegDst ? squashed `RD : squashed `RT) : 0); // $0 if not writing
    ID_MemRead <= MemRead;
    ID_ALUOp <= ALUOp;
    ID_MemWrite <= MemWrite;
    ID_Bad <= Bad;
  end
end

// EX: EXecute stage
always @(posedge clk) if (running) begin
  case (ID_ALUOp)
    `ALUADD: alu = ID_s + ID_src;
    `ALUOR:  alu = ID_s | ID_src;
    `ALUADD: alu = ID_s + ID_src;
    `ALUSUB: alu = ID_s - ID_src;
    `ALUSLT: alu = ID_s < ID_src;
    `ALUXOR: alu = ID_s ^ ID_src;
    default: alu = (ID_src << 16);
  endcase

  EX_alu <= alu;
  EX_t <= ID_t;
  EX_rd <= ID_rd;
  EX_MemRead <= ID_MemRead;
  EX_MemWrite <= ID_MemWrite;
  EX_Bad <= ID_Bad;
end
```

Color key: initialization
IF squashed ID blocked EX MEM WB debugging

```
// MEM: data MEMory access stage
always @(posedge clk) if (running) begin
  if (EX_MemRead) v = m[EX_alu >> 2]; else v = EX_alu;
  if (EX_MemWrite) m[EX_alu >> 2] <= EX_t;

  MEM_v <= v;
  MEM_rd <= EX_rd;
  MEM_Bad <= EX_Bad;
end

// WB: register Write Back stage
always @(posedge clk) if (running) begin
  if (MEM_rd) r[MEM_rd] <= MEM_v;
  if (MEM_Bad) halt <= 1;
end

// Reset
always @(posedge clk) if (reset) begin
    halt<=0;
    r[0]<=0;
    squash <= 0; target <= 0;
    IF_ir<=`NOP; IF_pc<=0;
    ID_s<=0; ID_t<=0; ID_src<=0; ID_rd<=0; ID_MemRead<=0; ID_ALUOp<=0; ID_MemWrite<=0; ID_Bad<=0;
    EX_alu<=0; EX_t<=0; EX_rd<=0; EX_MemRead<=0; EX_MemWrite<=0; EX_Bad<=0;
    MEM_v<=0; MEM_rd<=0; MEM_Bad<=0;
end

`ifdef TRACE
// State-by-state trace
always @(posedge clk) if (running) begin
  `define Fs      (canfwds ? (EX_dep1 ? "EX " : (MEM_dep1 ? "MEM " : "WB ")) : "")
  `define Ft      (canfwdt ? (EX_dep2 ? "EX " : (MEM_dep2 ? "MEM " : "WB ")) : "")
  $display("\n    running=%d blocked=%d squash=%d t=%5d squash=%id", running, blocked, `Fs, IF_ir `RS, `Ft, IF_ir `RT, squash);
  $display("IF  ir=%x pc=%id", IF_ir, IF_pc);
  case (IF_ir `OP)
    `RTYPE: begin
      case (IF_ir `FUNCT)
        `ADDU:   $display("IF  addu $%id,$%id,$%id", IF_ir `RD, IF_ir `RS, IF_ir `RT);
        `SUBU:   $display("IF  subu $%id,$%id,$%id", IF_ir `RD, IF_ir `RS, IF_ir `RT);
        `AND:    $display("IF  and $%id,$%id,$%id", IF_ir `RD, IF_ir `RS, IF_ir `RT);
        `OR:     $display("IF  or $%id,$%id,$%id", IF_ir `RD, IF_ir `RS, IF_ir `RT);
        `XOR:    $display("IF  xor $%id,$%id,$%id", IF_ir `RD, IF_ir `RS, IF_ir `RT);
        `SLTU:   $display("IF  sltu $%id,$%id,$%id", IF_ir `RD, IF_ir `RS, IF_ir `RT);
        default: $display("IF  OP=%x RS=%id RT=%id RD=%id SHAMT=%id FUNCT=%x", IF_ir `OP, IF_ir `RS, IF_ir `RT, IF_ir `RD, IF_ir `SHAMT, IF_ir `FUNCT);
      endcase
    end
    `BEQ:    $display("IF  beq $%id,$%id,offset=$%id", IF_ir `RT, IF_ir `RS, IF_ir `IMM);
    `ADDIU:  $display("IF  addiu $%id,$%id,$%id", IF_ir `RT, IF_ir `RS, IF_ir `IMM);
    `SLTIU:  $display("IF  sltiu $%id,$%id,$%id", IF_ir `RT, IF_ir `RS, IF_ir `IMM);
    `ANDI:   $display("IF  andi $%id,$%id,$%id", IF_ir `RT, IF_ir `RS, IF_ir `IMM);
    `ORI:    $display("IF  ori $%id,$%id,$%id", IF_ir `RT, IF_ir `RS, IF_ir `IMM);
    `XORI:   $display("IF  xori $%id,$%id,$%id", IF_ir `RT, IF_ir `RS, IF_ir `IMM);
    `LUI:    $display("IF  lui $%id,$%id", IF_ir `RS, IF_ir `IMM);
    `LW:     $display("IF  lw $%id,$%id($%id)", IF_ir `RT, IF_ir `IMM, IF_ir `RS);
    `SW:     $display("IF  sw $%id,$%id($%id)", IF_ir `RT, IF_ir `IMM, IF_ir `RS);
    default: $display("IF  OP=%x RS=%id RT=%id IMM=%x", IF_ir `OP, IF_ir `RS, IF_ir `RT, IF_ir `IMM);
  endcase
  if (ID_Bad) $display("ID  illegal instruction");
  else $display("ID  s=%id t=%id src=%id rd=%id MemRead=%b ALUOp=%b MemWrite=%b", ID_s, ID_t, ID_src, ID_rd, ID_MemRead, ID_ALUOp, ID_MemWrite);
  if (EX_Bad) $display("EX  illegal instruction");
  else $display("EX  alu=%id t=%id rd=%id MemRead=%b MemWrite=%b", EX_alu, EX_t, EX_rd, EX_MemRead, EX_MemWrite);
  if (MEM_Bad) $display("MEM illegal instruction");
  else begin
    if (EX_MemWrite) $display("MEM m[%id]=%id v=%id rd=%id", EX_alu, EX_t, MEM_v, MEM_rd);
    else $display("MEM v=%id rd=%id", MEM_v, MEM_rd);
  end
  if (MEM_rd) $display("WB  r[%id]=%id", MEM_rd, MEM_v);
end
`endif
endmodule

// Testbench
module bench;
wire halt;
reg reset = 1;
reg clk = 0;

processor PE(halt, reset, clk);

initial begin
  #`CLKDEL clk = 1;
  #`CLKDEL clk = 0;
  reset = 0;
  while (($time < `RUNTIME) && !halt) begin
    #`CLKDEL clk = 1;
    #`CLKDEL clk = 0;
  end
end
endmodule
```

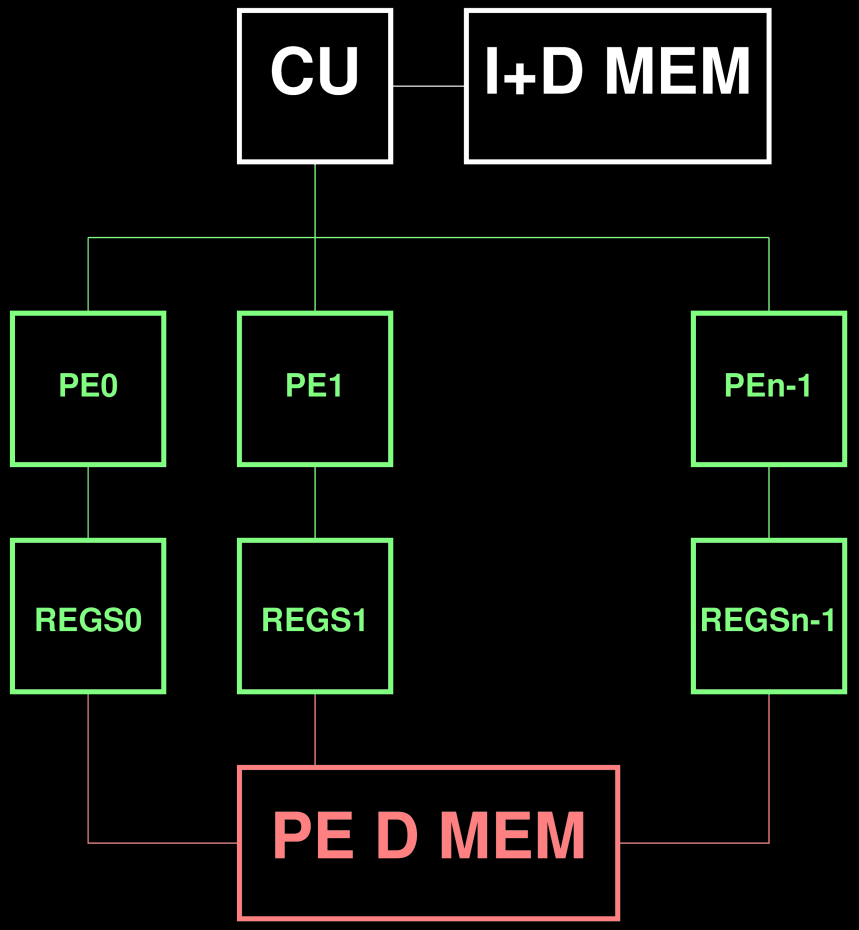


SIMD parallel across PEs (processing elements)

- E.g.: Thinking Machines, MasPar
- Each PE either applies same operation or is disabled for that operation
- 1980s examples were bit-serial, but parallel across massive number of PEs (e.g., 64K)

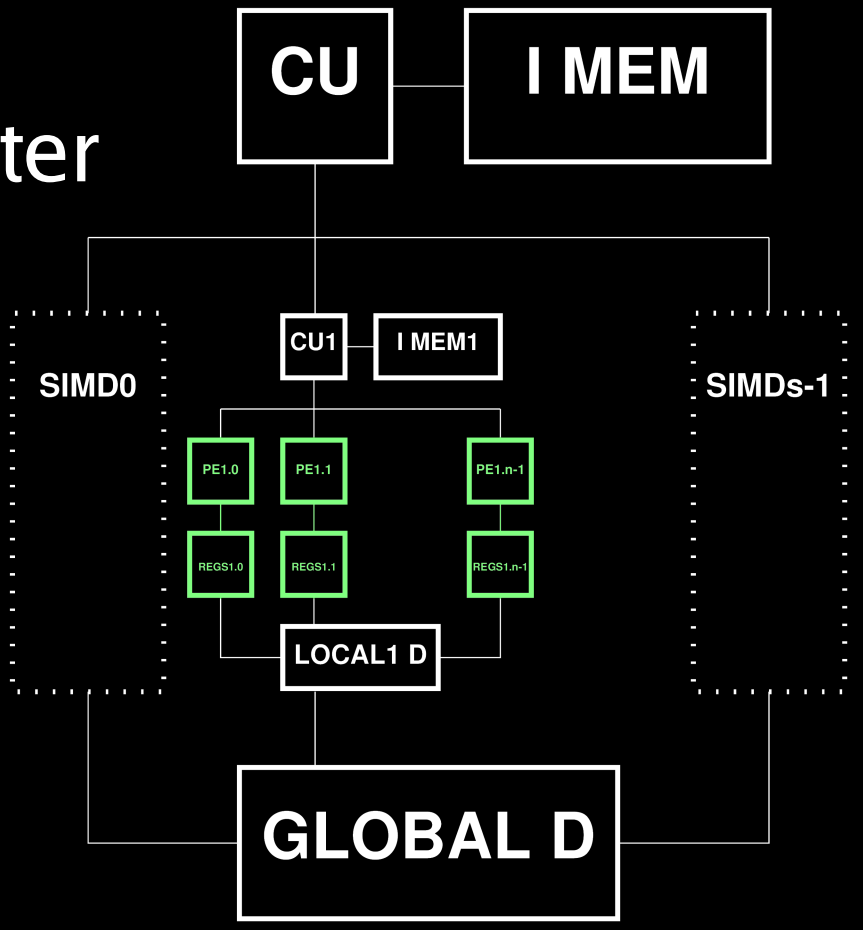
Generic SIMD Architecture

- Control Unit
  conventional processor…
- Processing Elements
  ALUs + local data
- PE Data Memory
  Optional; coherent?
- **GPUs *nest*** this structure within each PE+REGS



Generic GPU Architecture

- Graphics Processing Unit attached to a Host computer
- Processing Elements have 2D PE numbering
- Virtualized PEs behave like vector pipelines
- If PE*x.y* for all *y* are disabled, CU*x* can skip



Generic SIMD Programming

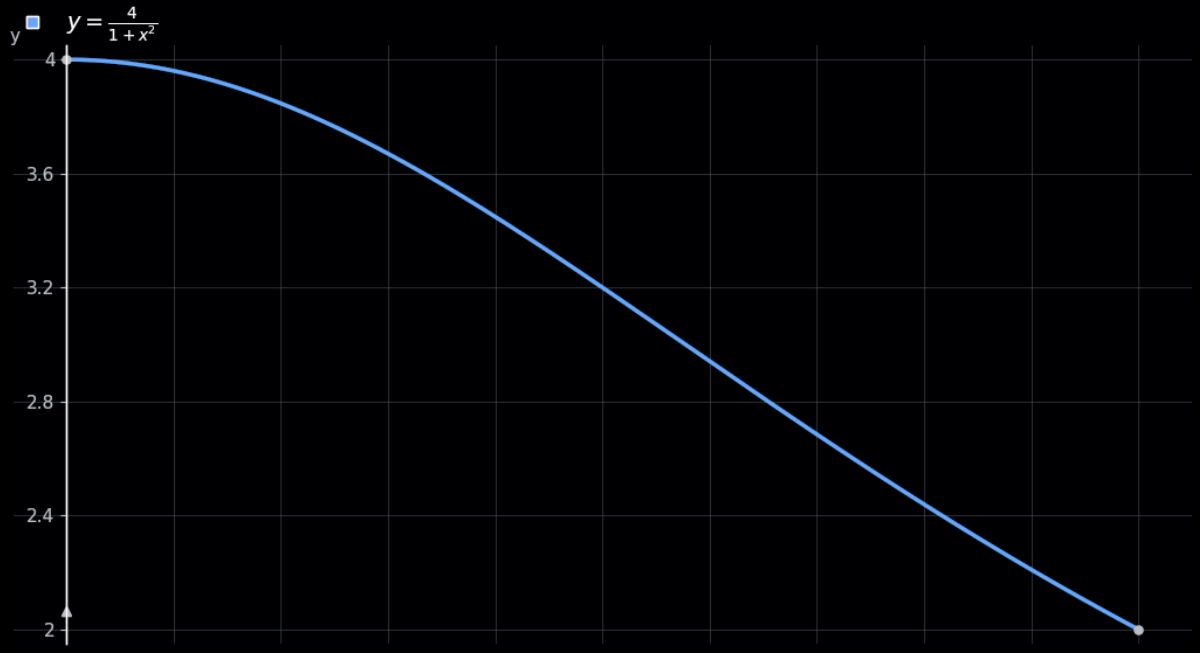
- A program has two parts:
  - Sequential (singular) part run on CU or Host
  - Parallel (plural) part run on PEs
- Parallel code says what happens to an element, hardware applies that to all elements on PEs that are not disabled
- SIMD: inter-PE communication is cheap
  GPU: inter-CU*x* communication is expensive, but lower fanout allows a higher clock speed

Sample Algorithm: Compute π

- Area of a circle is πr^2, so r=1 area is π
- Area under $y=4/(1+x^2)$ over x=0..1 is also π

$y = \frac{4}{1+x^2}$ for $x \in [0, 1]$

Graph of a rational function showing a decreasing curve



π in C as an integral

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv) {
 int i, intervals = atoi(argv[1]);
 double width = 1.0 / intervals, sum = 0;
 // sum curve height at each interval
 for (i=0; i<intervals; ++i) {
  double x = (i + 0.5) * width;
  sum += 4.0 / (1.0 + x * x);
 }
 sum *= width; // multiply by width to get area
 printf("Pi is roughly %f\n", sum);
 return(0);
}
```

π in MPL (MasPar's SIMD C)

```
#include <mpl.h> // MPL support defines nproc, etc.

int main(int argc, char **argv) {
 int intervals = atoi(argv[1]);
 double width = 1.0 / intervals, sum = 0;
 plural double dsum = 0;
 plural int i;
 // sum curve height at each interval
 for (i=iproc; i<intervals; i+=nproc) {
  plural double x = (i + 0.5) * width;
  dsum += 4.0 / (1.0 + x * x);
 }
 sum = reduceAddd(dsum); // tree summation of dsum from all PEs
 sum *= width; // multiply by width to get area
 printf("Pi is roughly %f\n", sum);
 return(0);
}
```

π in MPL (MasPar's SIMD C)

```
#include <mpl.h> // MPL support defines nproc, etc.

int main(int argc, char **argv) {
 int intervals = atoi(argv[1]);
 double width = 1.0 / intervals, sum = 0;
 plural double dsum = 0;
 plural int i; int j;
 // sum curve height at each interval
 for (i=iproc; i<intervals; i+=nproc) {
  plural double x = (i + 0.5) * width;
  dsum += 4.0 / (1.0 + x * x);
 }
 // tree summation of dsum from all PEs
 for (j=nproc/2; j>0; j>>=1) dsum += router[iproc+j].dsum;
 sum = proc[0].dsum;
 sum *= width; // multiply by width to get area
 printf("Pi is roughly %f\n", sum);
 return(0);
}
```

π in CUDA (final sum in Host)

```
#include <stdio.h>
#include <cuda.h> // CUDA support
#define INTERVALS 1000000
#define BLOCKS 4 // number of thread blocks (little SIMDs)
#define THREADS 192 // number of threads per block (virtual PEs per little SIMD)

// Kernel that executes on the CUDA device (the GPU)
__global__ void summer(float *sum, int intervals, float width, int nproc) {
 float x; int i, iproc = blockIdx.x*blockDim.x+threadIdx.x; // iproc is global PE number
 for (i=iproc; i<intervals; i+=nproc) {
  x = (i+0.5)*width; sum[iproc] += 4.0/(1.0+x*x); // sum[iproc] is sum computed by PE iproc
 }
}

// Main routine that executes on the Host
int main(void) {
 dim3 dimGrid(BLOCKS,1,1); dimBlock(THREADS,1,1); // Grid and Block dimensions
 float pi = 0, width = 1.0/INTERVALS;
 size_t size = BLOCKS*THREADS*sizeof(float); // array memory size in bytes
 float *sumDev, *sumHost = (float *)malloc(size); // allocate array on Host
 cudaMalloc((void **) &sumDev, size); cudaMemset(sumDev, 0, size); // allocate & zero array on GPU
 summer <<<dimGrid, dimBlock>>> (sumDev, INTERVALS, width, THREADS*BLOCKS); // run GPU kernel
 cudaMemcpy(sumHost, sumDev, size, cudaMemcpyDeviceToHost); // copy sum array from GPU to Host
 for(int i=0; i<THREADS*BLOCKS; ++i) pi += sumHost[i]; // sum partial sums in Host (sequential!)
 pi *= width; // multiply by width to get area
 printf("Pi is roughly %f\n",pi);
 free(sumHost); cudaFree(sumDev); // free data structures (would happen at exit anyway)
 return(0);
}
```

Shared Memory MIMD (Multiple I, Multiple D)

- E.g.: SGI Origin, Multi-Core processors
- Each PE runs a process or thread
- Coherent memory doesn't easily scale...

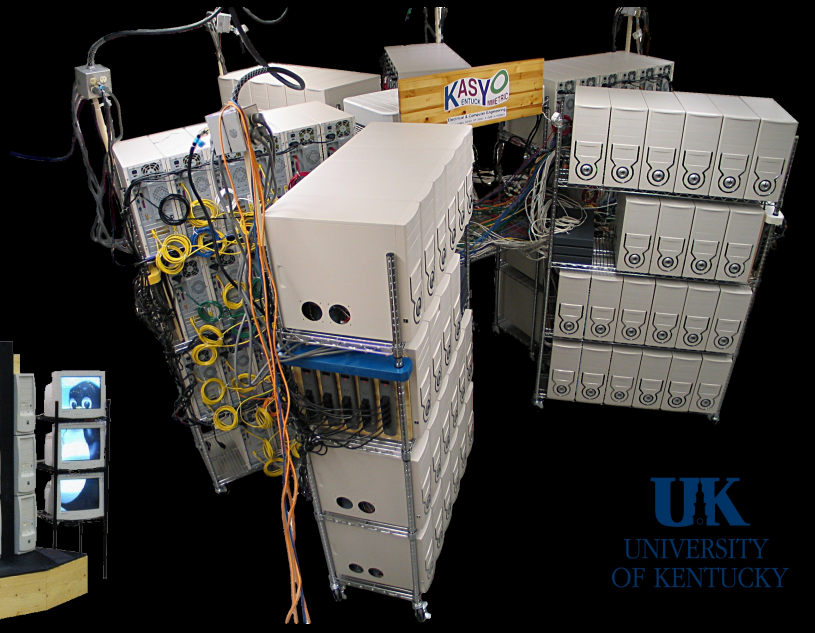
π in C using PThread Library

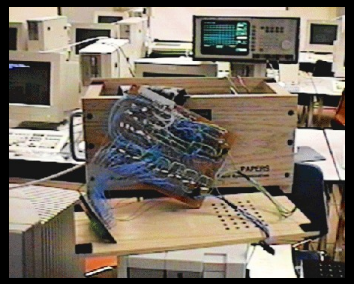
```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
volatile double pi = 0.0; // approximation to pi (shared)
pthread_mutex_t pi_lock; // lock for unique access to pi
volatile double intervals; // how many intervals?
#define NPROC 8

void *process(void *arg) {
 double localsum = 0, width = 1.0 / intervals;
 int i, iproc = *((int *) arg);
 for (i=iproc; i<intervals; i+=NPROC) { // partial summation on this PE
  double x = (i + 0.5) * width; localsum += 4.0 / (1.0 + x * x);
 }
 pthread_mutex_lock(&pi_lock); // wait to have unique access to pi
 pi += localsum; // I'm the only PE here now!
 pthread_mutex_unlock(&pi_lock); // we are done with pi
 return(0);
}

int main(int argc, char **argv) {
 pthread_t thread[NPROC]; void * retval; int ip[NPROC];
 intervals = atoi(argv[1]);
 pthread_mutex_init(&pi_lock, NULL); // initialize the lock on pi to unlocked
 for (int i=0; i<NPROC; ++i) ip[i] = i; // initialize iproc values for threads
 for (int i=0; i<NPROC; ++i) pthread_create(&thread[i], NULL, process, &ip[i]); // make threads
 for (int i=0; i<NPROC; ++i) pthread_join(thread[i], &retval);  // join (collapse) threads
 pi *= width; // everybody joined, so multiply by width to get area
 printf("Pi is roughly %f\n", pi);
 return(0);
}
```

π in C using OMP Directives

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h> // OMP support

int main(int argc, char **argv) {
 int i, intervals = atoi(argv[1]);
 double width = 1.0 / intervals, sum = 0;
 // sum curve height at each interval
#pragma omp parallel for reduction(+: sum) schedule(static)
 for (i=0; i<intervals; ++i) {
  double x = (i + 0.5) * width;
  sum += 4.0 / (1.0 + x * x);
 }
 sum *= width; // multiply by width to get area
 printf("Pi is roughly %f\n", sum);
 return(0);
}
```





Message-Passing MIMD & Clusters

- E.g.: ASCI Red, Beowulf, KLAT2, KASY0 ...
- Each PE runs a process or thread
- Coherent memory doesn't easily scale...

π in C using PVM Library

```
#include <stdlib.h>
#include <stdio.h>
#include <pvm3.h> // Parallel Virtual Machine library
#define NPROC 4 // set number of PE processes

int main(int argc, char **argv) {
 int intervals = atoi(argv[1]);
 double width = 1.0 / intervals, sum = 0;
 int tids[NPROC]; tids[0] = pvm_mytid(); // enroll in PVM
 int iproc = pvm_joingroup("pi"); // join a group
 if (iproc == 0) { // first PE process in group creates the rest
  pvm_spawn("pvm_pi", &argv[1], 0, NULL, NPROC-1, &tids[1]);
 }
 pvm_barrier("pi", NPROC); // ensure all PEs exist
 for (i=iproc; i<intervals; i+=NPROC) {
  double x = (i + 0.5) * width; sum += 4.0 / (1.0 + x * x);
 }
 // tree reducton of sum from all PEs
 pvm_reduce(PvmSum, &sum, 1, PVM_DOUBLE, 4, "pi", 0);
 sum *= width;
 if (iproc == 0) printf("Pi is roughly %f\n", sum); // only PE0 prints
 // check all PEs are here, leave group, and exit PVM
 pvm_barrier("pi", NPROC); pvm_lvgroup("pi"); pvm_exit();
 return(0);
}
```

π in C using AFAPI Library

```
#include <stdlib.h>
#include <stdio.h>
#include "afapi.h" // Aggregate Function API

int main(int argc, char **argv) {
 int intervals = atoi(argv[1]);
 double width = 1.0 / intervals, sum = 0;
 if (p_init()) exit(1); // check in with AF network hardware
 for (i=IPROC; i<intervals; i+=NPROC) {
  double x = (i + 0.5) * width; sum += 4.0 / (1.0 + x * x);
 }
 sum = p_reduceAdd64f(sum); // AF network sums from all PEs
 sum *= width;
 if (IPROC == CPROC) { // only AFAPI console PE prints
  printf("Pi is roughly %f\n", sum);
 }
 p_exit(); // check out with AF network hardware
 return(0);
}
```

π in C using MPI Messages

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h> // MPI support

int main(int argc, char **argv) {
 int i, intervals = atoi(argv[1]);
 double width = 1.0 / intervals, sum = 0, lsum = 0;
 int nproc, iproc; MPI_Status s;
 if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1); // check in
 MPI_Comm_size(MPI_COMM_WORLD, &nproc); // how many PEs?
 MPI_Comm_rank(MPI_COMM_WORLD, &iproc); // who am I?
 for (i=iproc; i<intervals; i+=nproc) {
  double x = (i + 0.5) * width; lsum += 4.0 / (1.0 + x * x);
 }
 lsum *= width;
 if (iproc != 0) {
  MPI_Send(&lsum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD); // send to PE0
 } else {
  sum = lsum;
  for (i=1; i<nproc; ++i) { // add lsum from each of PE1..PEnproc
   MPI_Recv(&lsum, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &s);
   sum += lsum;
  }
  printf("Pi is roughly %f\n", sum);
 }
 MPI_Finalize(); return(0); // check out and exit
}
```

π in C using MPI Collectives

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h> // MPI support

int main(int argc, char **argv) {
 int i, intervals = atoi(argv[1]);
 double width = 1.0 / intervals, sum = 0, lsum = 0;
 int nproc, iproc; MPI_Status s;
 if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1); // check in
 MPI_Comm_size(MPI_COMM_WORLD, &nproc); // how many PEs?
 MPI_Comm_rank(MPI_COMM_WORLD, &iproc); // who am I?
 for (i=iproc; i<intervals; i+=nproc) {
  double x = (i + 0.5) * width; lsum += 4.0 / (1.0 + x * x);
 }
 lsum *= width;
 // collectively tree reduce lsum values into a single sum by adding
 MPI_Reduce(&lsum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
 if (iproc == 0) {
  printf("Pi is roughly %f\n", sum);
 }
 MPI_Finalize(); return(0); // check out and exit
}
```

What is a Quantum Computer?

Is this a Quantum Computer?

Is this a Quantum Computer?

Yup!

Google
Sycamore

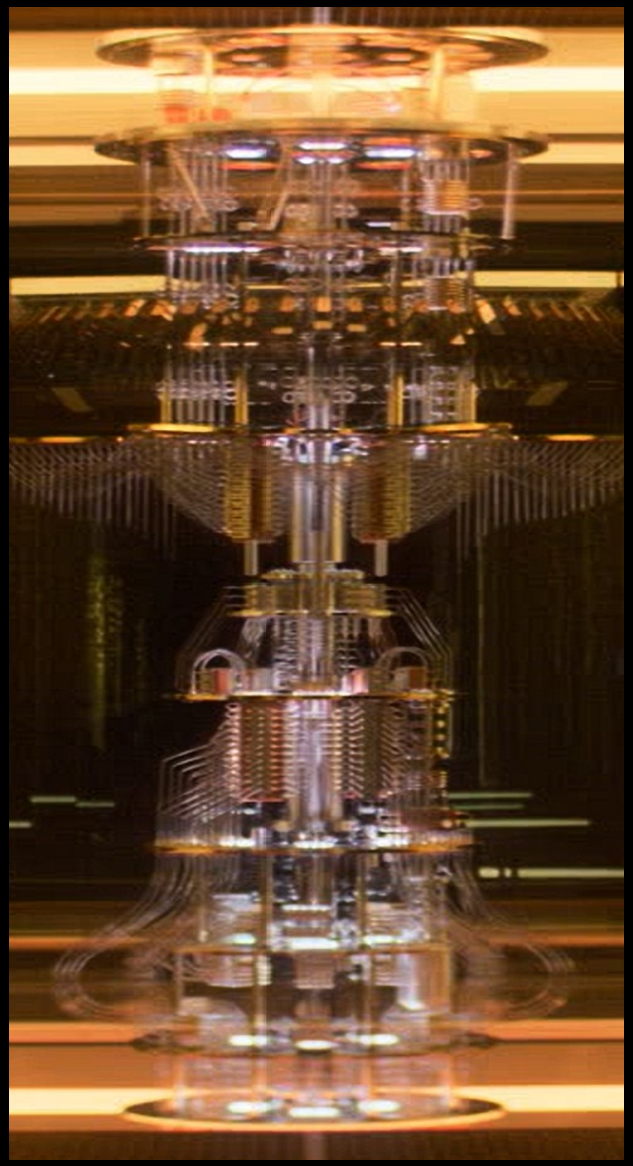
Is this a Quantum Computer?

Is this a Quantum Computer?

Yup!

SpinQ Gemini Mini
~$8K

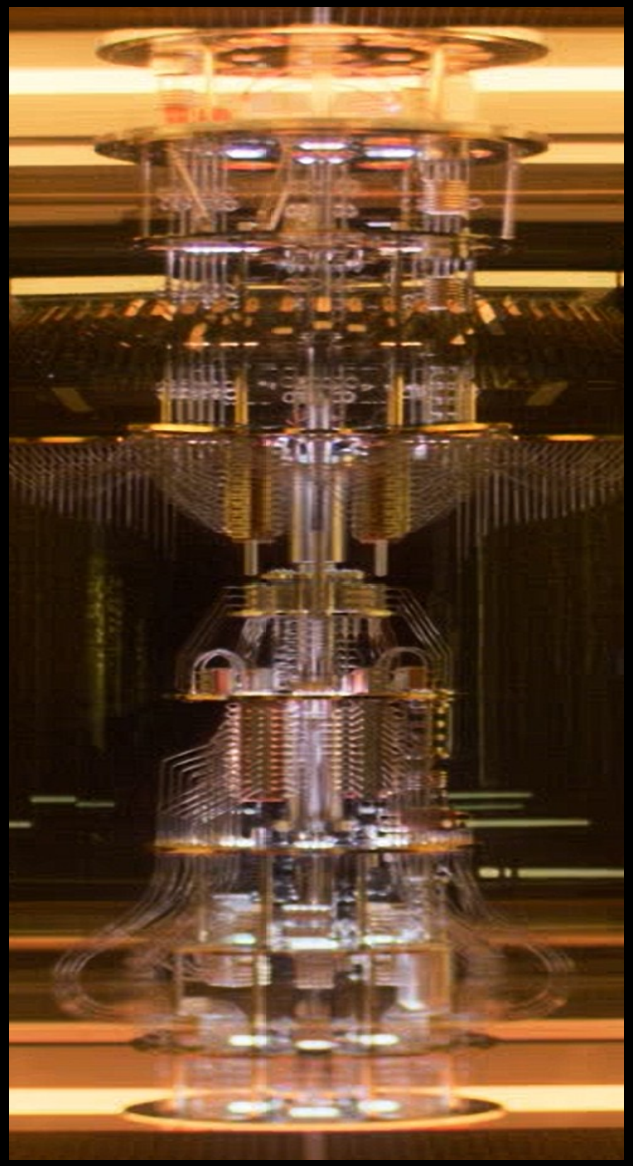
Is this a Quantum Computer?



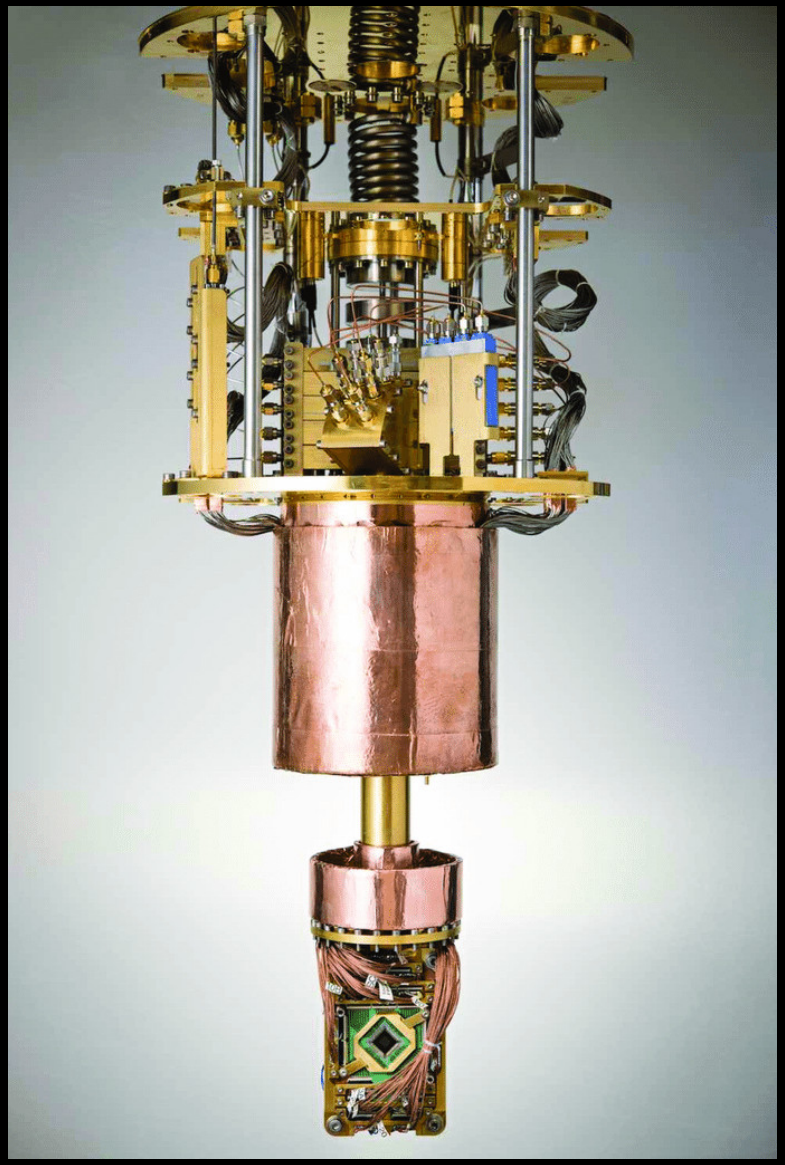
Is this a Quantum Computer?

Nope!

It is from the 2020 TV mini series **DEVS**



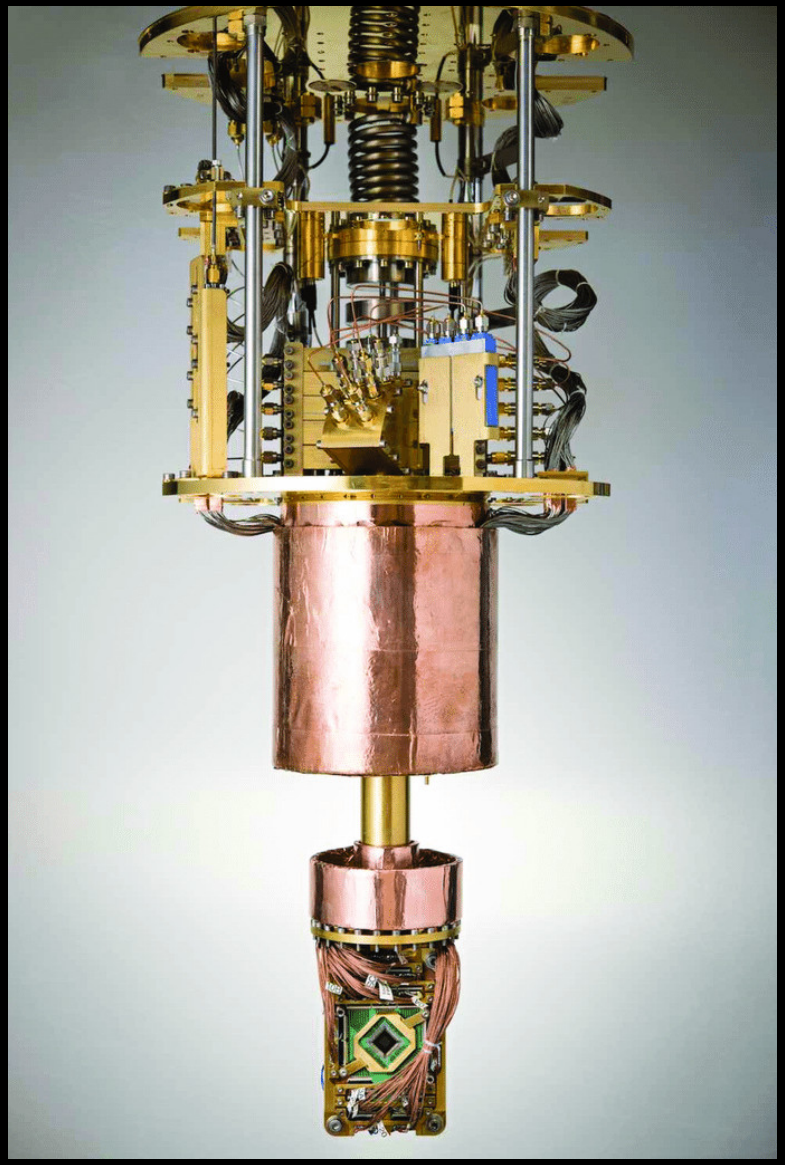
Is this a Quantum Computer?



Is this a Quantum Computer?

Yup!

It is a
D-Wave 2000Q



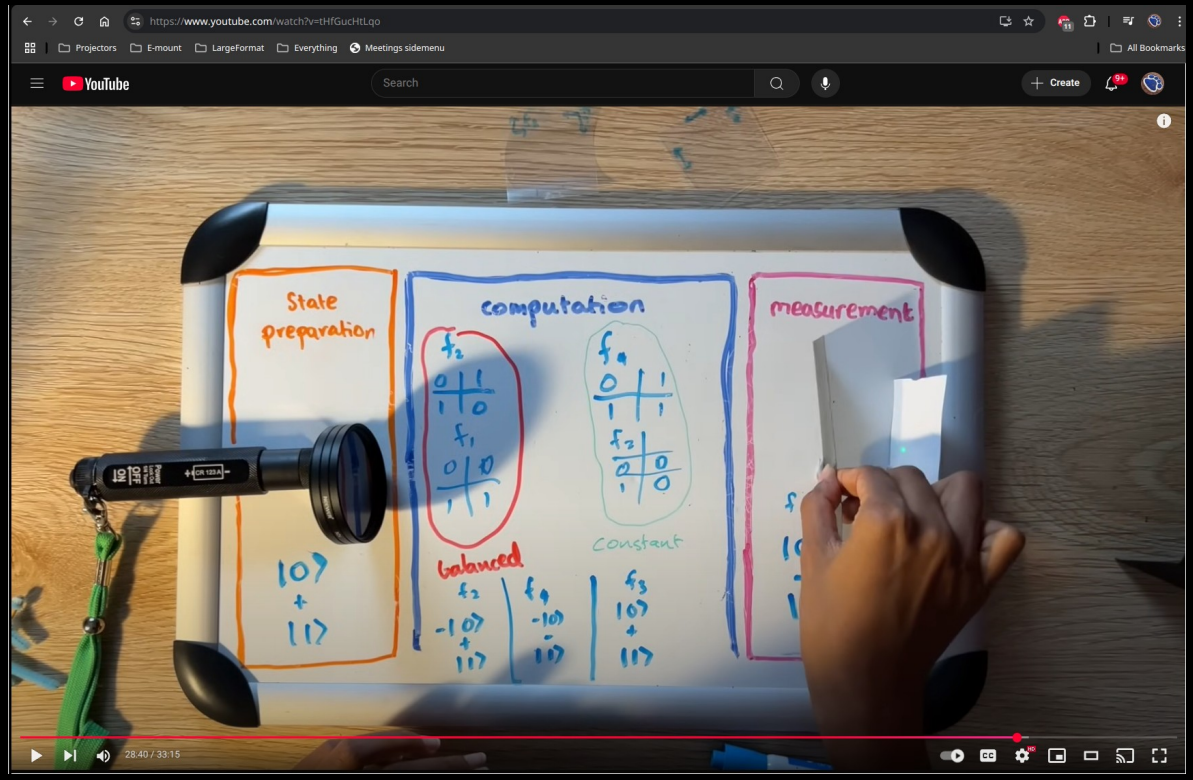
Is this a Quantum Computer?

Is this a Quantum Computer?

Not quite.

1/2-scale model of **Fujitsu**

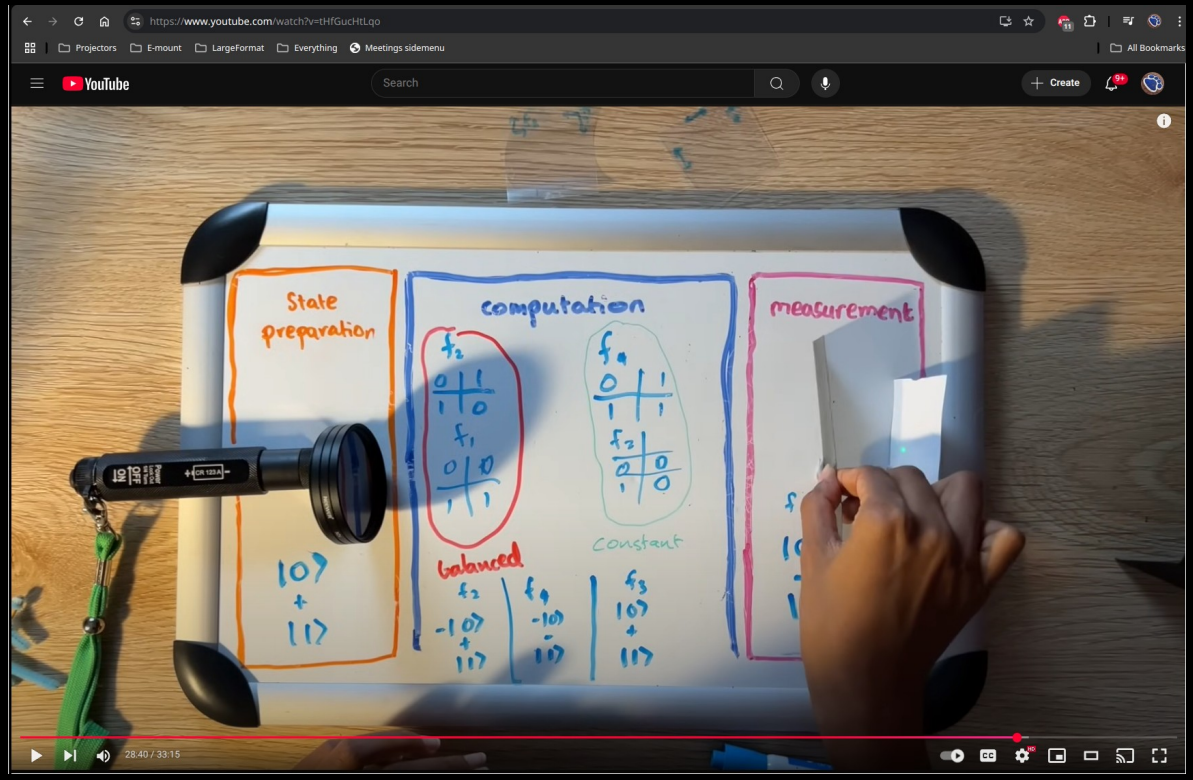
Is this a Quantum Computer?



Is this a Quantum Computer?

Yup.

A photonic qubit using polarization



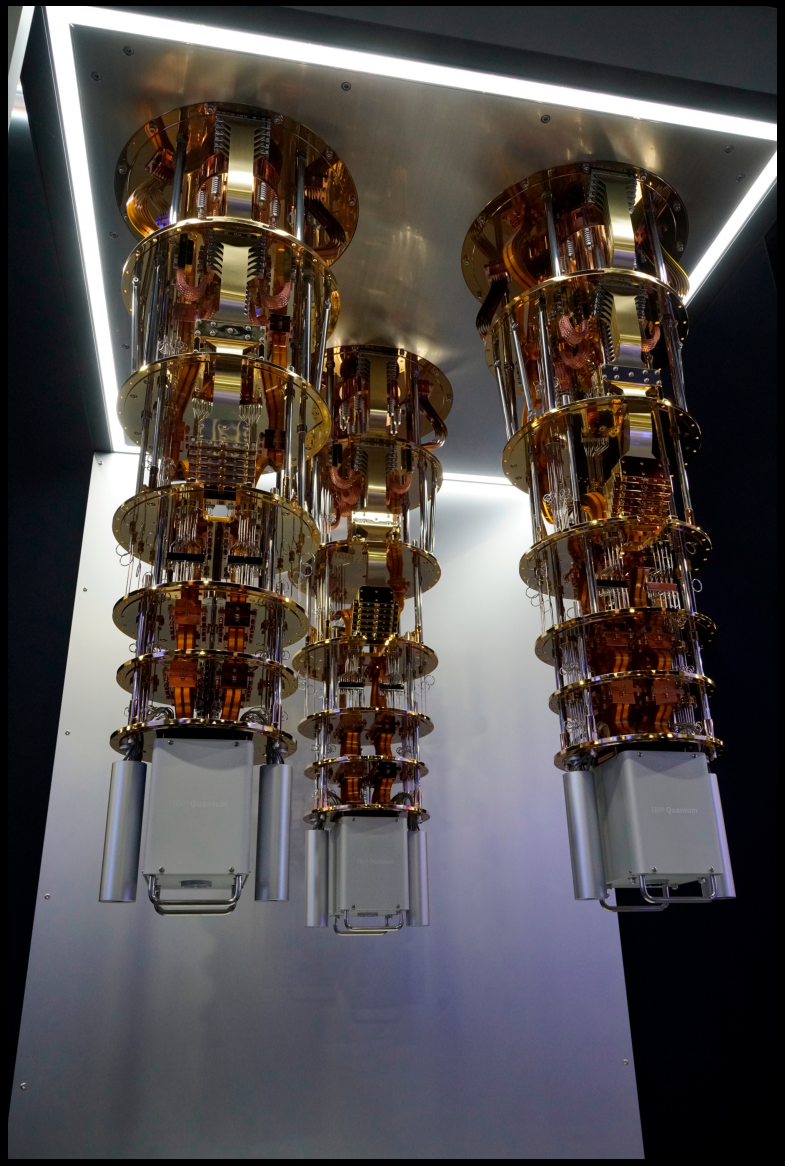
Is this a Quantum Computer?

Is this a Quantum Computer?

Yup!

IBM Q

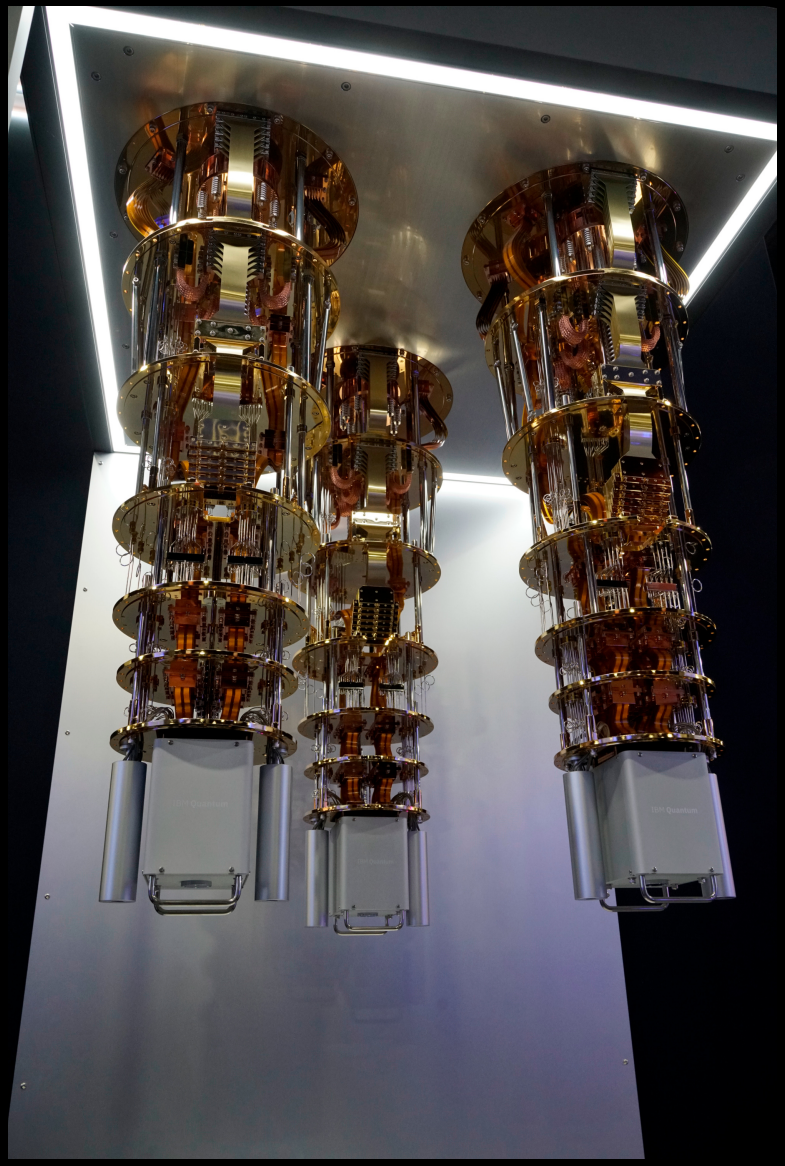
Is this 3 Quantum Computers?



Is this 3 Quantum Computers?

Not quite.

80% scale
model of
IBM Quantum
System Two



Is this a Quantum Computer?

Is this a Quantum Computer?

Nope!

BLUEFORS
This is a dummy; they make cooling, not computers

Is this a Quantum Computer?

Is this a Quantum Computer?

Not quantum;
quantum-inspired.

KREQC is
Kentucky's
Rotationally
Emulated
Quantum
Computer

What is a Quantum Computer?

- Operates on **Qubits** rather than bits
  - A bit can be either 0 or 1
  - A qubit can be 0, 1, or a **Probability Density Function** over {0,1}
  - A gate-level op on a qubit alters the PDF
- What is a **probability density function**?
  - *Encodes* probability of 0 or 1
  - Not probability because it has more than 1D
- A qubit is really a wave. Isn't everything? ;-)

What is a Quantum Computer?

- Two or more qubits can be **Entangled**
  - Entangled things have their waves aligned
  - This implies ***E* entangled qubits can hold a PDF over all 2^E possible *E*-bit values**
  - Entangled things are coupled, creating what Einstein called ***Spooky action at a distance***
- Measuring a qubit's value always gives 0 or 1, and makes that qubit's value what you read: **measurement collapses superposition**… or does it? Maybe you just get entangled too? ;-)

What is a Quantum Computer?

- We think everything in the Universe operates on quantum mechanics, so there are lots of ways to implement qubits and operations on them
- The catch is that it is very difficult to keep qubits from interacting with other stuff
  - Noise corrupts/collapses superposition
  - *Performing an operation **is** injecting noise*
- Only **thermodynamically reversible** ops are viable, and which depends on mechanism used

What is a Quantum Computer?

- Sequential code could work if we had enough qubits, etc., but we don't
- Using superposed values to simulate vectors could compute the local sums, but then how do we sum across all of them?
- No obvious way to do different operations on individual superposed value components at the same time, so it doesn't match the MIMD model
- Need a somewhat different algorithm...

π in a Quantum Computer?

- Sequential code could work if we had enough qubits, etc., but we don't
- Using superposed values to simulate vectors could compute the local sums, but then how do we sum across all of them?
- No obvious way to do different operations on individual superposed value components at the same time, so it doesn't match the MIMD model
- Need a somewhat different algorithm...

π in C using `int` sampling

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv) {
 int i, j, intervals = atoi(argv[1]);
 int w = intervals * intervals, sum = 0;
 for (i=0; i<intervals; ++i) {
  int h = w / (intervals + ((i*i) / intervals));
  for (j=0; j<intervals; ++j) {
   if (h > j) ++sum;
  }
 }
 printf("Pi is roughly %f\n", (4.0 * sum) / w);
 return(0);
}
```

π in PBP, `pint` sampling

```
#include "pbp.h" // PBP classes and support

int main(int argc, char **argv) {
 int bits = atoi(argv[1]); // number of pbits
 pint intervals(1 << bits); // intervals in pbits
 pint w(1 << (2 * bits)); // int scaling factor
 pint x = pint(0).Had(bits); // all x values
 pint y = pint(0).Had(bits,bits); // all y values
 pint h = w / (((x * x) >> bits) + intervals);
 pint r = (h > y); // r is 1 where below curve
 // count 1s; quantum would sample probability
 double pi = (4.0 * r.Pop()) / (1 << REWAYS);
 printf("Pi is roughly %f\n", pi);
 return(0);
}
```