2019

Computer Design

CAMBRIDGE COMPUTER SCIENCE TRIPOS PART IB, PAPER 5

ASHWIN AHUJA

Table of Contents

System Verilog 2
Operators 2
Arithmetic Operators 2
Unsigned and Signed Numbers 3
Bases and Bit-Widths 3
Constants 3
Conditionals 3
Functions 4
Enumerations 4
Registers 4
Gates to Processors 5
Introduction & Motivation 5
Logic Modelling, Simulation & Synthesis 5
Design for FPGAs 7
Testing 11
Instruction sets and introduction to processor architecture 13
RISC-V 13
CLARVI 15
History of Computer Architecture 16
RISC Processors 18
CISC, the Intel ISA, and Java VM 22
Java Virtual Machine 26
Memory Hierarchy 27
Hardware for OS Support 30
Pipelining 34
System-on-chip architecture 36
Definitions 36
Multicore Processors 39
Graphics Processing Units (GPUs) 43
Future Directions 50

System Verilog

```
/* Note:
   System Verilog uses C++/Java style comments */

// full-adder implementation called "full_adder"
module full_adder(
       // declare inputs:
       input[1:0]  Cin,
       // declare outputs:
       output Cout
       );

    // declare intermediate wires:
    wire ta,tb,tc;
    wire [15:0] data;

    // instantiate modules
    XOR myxor(Cin, Cout);
endmodule
```

Operators

```
assign Cout = (A && B) || (A && Cin) || (B && Cin);
    /*
    && is logical AND
    & is bitwise AND

    || is logical OR
    | is bitwise OR

    ^ is bitwise XOR

    ! is not

    == is logical equality

    != is logical inequality

    === is comparison to see if they are tristate or undefined (basically extends
equality to four value logic)
    !== is the same for non equality

    assign is a keyword which indicates continuous assignment ie used in
combinational circuits
    */
```

Arithmetic Operators

```
assign S = A + B; // also have '-', '*', '/' and '%'
// need to be careful about size of inputs and outputs - if A and B are 2 bits, S
is 3 bits, therefore we can do this:
```

```
assign sum = A + B;
assign Cout = sum[2];
assign S = sum[1:0];
```

Unsigned and Signed Numbers

```
// by default things are unsigned
input signed a;
wire signed [2:0] sum;
```

Bases and Bit-Widths

```
/*
b – binary
d – decimal
h – hex
o – octal
*/
4'b1010 == 4'd10 == 4'ha
// bit-widths specified by a width and a quote mark
-8'd3 == 8'b1111_1101 == 8'hfd

wire [3:0] ab;
wire [1:0] A;
wire [1:0] B;
...
assign ab = {A, B};
```

Constants

```
parameter x = a
'define height 1024
```

Conditionals

```
condition ? equation_if_condition_is_true : equation_if_condition_is_false
wire [2:0] sum;
sum = A + B;
assign S = sum[2] ? 2'b11 : sum[1:0];

// We can use this to encode a truth table directly, like so:
wire [3:0] ab = {A,B};
assign S = (ab==4'b0000) ? 2'b00:
           (ab==4'b0001) ? 2'b01:
           (ab==4'b0010) ? 2'b10:
           (ab==4'b0011) ? 2'b11:
           (ab==4'b0100) ? 2'b01:
           (ab==4'b0101) ? 2'b10:
           // default case of 2'b11 for (ab==4'b0110) and (ab==4'b0111)
           (ab==4'b1000) ? 2'b10:
           // default case for the rest of the table
           2'b11;
```

Functions

```
// Functions can only generate constants – have to be evaluated in compile time
// automatic keyword can be used for functions that reenter
function automatic [15:0] factorial;
  input [2:0] n;
  if(n <=1)
    factorial = 1;
  else
    factorial = n * factorial(n-1);
endfunction
```

Enumerations

```
typedef enum {red, green, blue} RGB; // by default this is a 32 byte type using an
int
typedef enum bit {TRUE, FALSE} MyBoolean;
typedef enum bit [1:0] {A, B, C} state;

typedef struct {
       bit [5:0] opcode;
       bit [4:0] rs, rt, rd;
} example_struct
```

Registers

```
reg r;
always_ff @(posedge clock or posedge reset)
    if(reset)
        r <= 0;
    else
        r <= !r;

reg [DATE_WIDTH-1:0] myram [0:MEM_SIZE-1]; // how to define a memory
/*
always_ff perform following actions when sensitivity list is true
    _ff implies all assignments need to be to registers and not to wires
<= non-blocking assignment – assigns the values in the block at the same time
= blocking assignment – completes before control passes on to the next statement
begin and end must be used to surround a block
*/

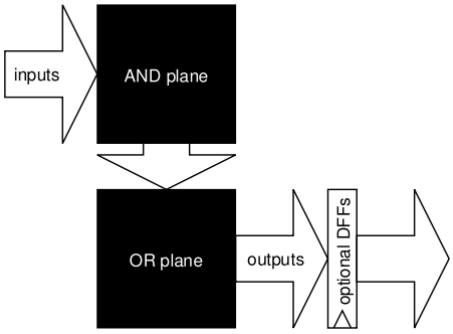
always_comb // always block which produces only combinational logic – only changes
wires

// can set the output of a module to be a register – wires the Q-output of the DFF
to the output of the module
```

Gates to Processors

Introduction & Motivation

- **PLA:** Programmable Logic Array
  - Cheap – low cost per chip and simple to use
  - Medium to low density integrated devices (not many gates per chip) so cost per chip is high



- **FPGA:** Field Programmable Gate Array
  - Sea of logic elements – a 16x1 block of SRAM can provide any Boolean function of 4 variables with a D flip flop per logic element
  - Programmable interconnect: program bits enable / disable tristate buffers and transmission gates
  - Rapid prototyping, cost effective in low to medium volume
  - 15x to 25x bigger and slower than CMOS ASICs
- **CMOS ASICs:** Complementary Metal Oxide Semiconductor Application Specific Integrated Circuit
  - Full control over the chip design – some blocks fully custom with others being **standard cell** or generated by a macro.
- **Moore's Law:** Transistor density was doubling (and halving in price) every 18 to 24 months – generally been met but failing these days

Logic Modelling, Simulation & Synthesis

- **Four Valued Logic**

value	meaning
0	false
1	true
x	undefined
z	high impedance

AND	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

OR	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

NOT	output
0	1
1	0
x	x
z	x

BUFT	enable			
data	0	1	x	z
0	z	0	x	x
1	z	1	x	x
x	z	x	x	x
z	z	x	x	x

  - x generally appears in simulation for uninitialized state but generally does not exist when implemented.
  - Z is unconnected wires (for tristate buffer)

==	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

===	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

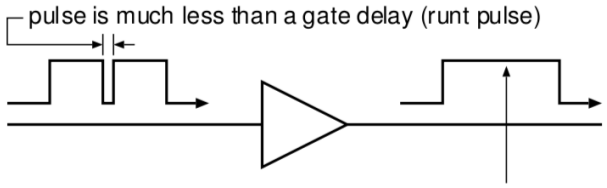
- **Tri-State Buffer**

```
module BUFT(
    output reg out,
    input      in,
    input      enable);

    // behavioural use of always which activates whenever
    // enable or in changes (i.e. both positive and negative edges)
    always @(enable or in)
        if(enable)
            out = in===1'bz ? 1'bx : in;
        else
            out = 1'bz; // assign high-impedance
endmodule
```

- **Further Logic Levels**
  - For capacitors, we can have weak_low and weak_high (when they aren't being driven but are charged or discharged respectively)
  - VHDL uses 46 state logic

- **Modelling Delays**
  - **Pure Delay:** Signals are delayed by some time constant
    - *assign #10 delayed_value = none_delayed_value*
  - **Inertial Delay:** Models capacitive delay – see example:



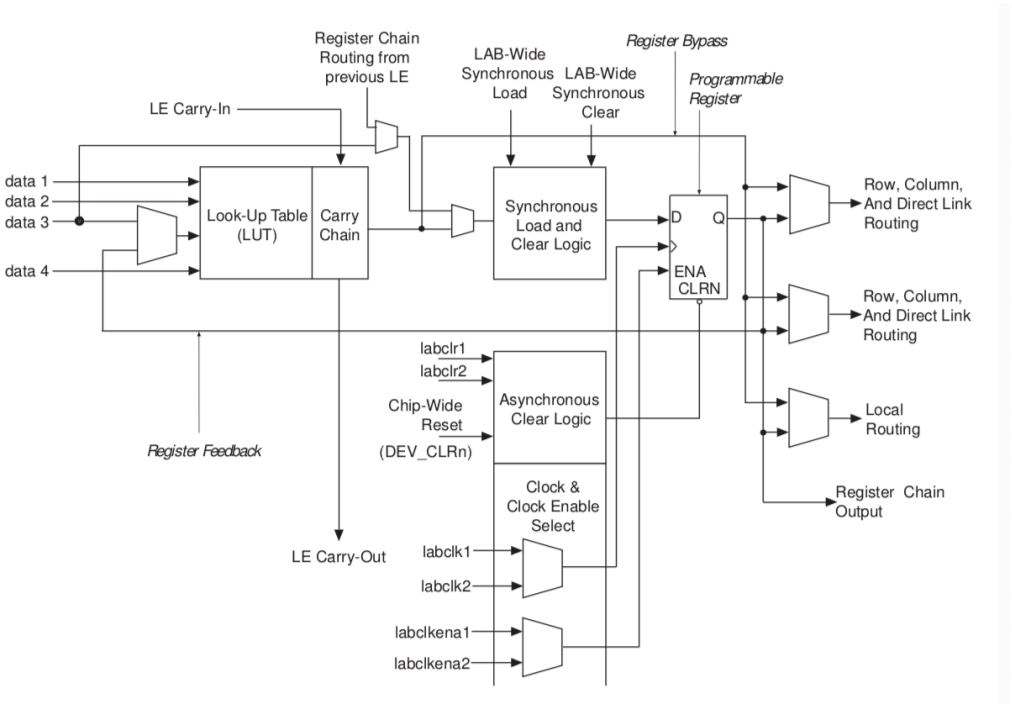
runt pulse has been filtered out by the gate capacitance

  - **How to obtain delay information**
    - Add up gate delays and capacitances of wire lengths once the design has been synthesized – can then be done through back annotation (can be useful for multiple other parts).
    - Many parts of the circuit are likely to be inactive at a given instant in time, therefore having to re-evaluate for each simulation cycle is incredibly expensive.
    - Delays have to be implemented as long strings of buffers which will slow things down.
  - **Discrete Event Simulation**
    - (1) Only changes in state cause gates to be reevaluated
    - (2) Gates are modelled as objects – state changes are passed as timed events or messages and these are inserted in a time ordered queue.
    - (3) Simulation loops around: (i) Pick least time event, (ii) pass event to the appropriate gate and (iii) gate evaluates and produces an event if its output has changed.

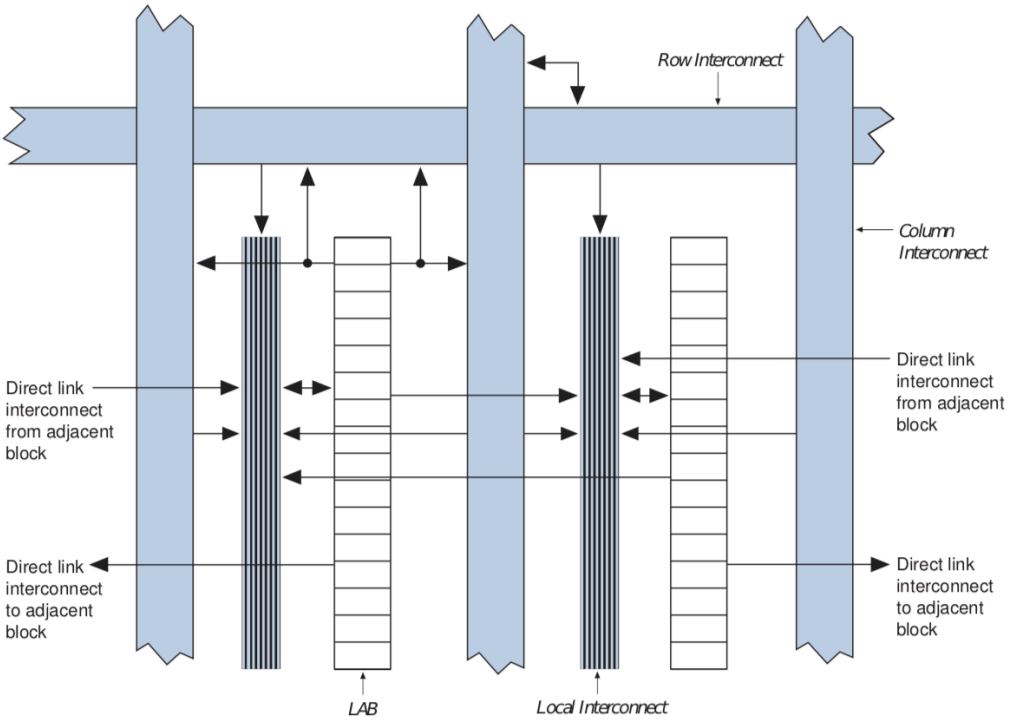
        - Cancelling runt pulses – event removal
        - Modelling wire delays
    - **SPICE: Simulation Program with Integrated Circuit Emphasis**
        - It is used for detailed analog transistor level simulation – models nonlinear components as a set of differential equations.
        - Simulation is very accurate but computationally very expensive
- **Logic Minimisation**
    - Karnaugh Maps and QM (see Digital Electronics, Michaelmas 2017)
    - If optimising over multiple outputs, with shared terms, there is a further algorithm – Putnam and Davis
    - Important to note that SOP or POS may not be simplest logic structure – multilevel logic structures could be more compact
    - Perhaps XOR gates?
    - It is important to note that 'don't care' states are very important for logic minimisation so when writing SystemVerilog should indicate these:
        - ```
          wire [31:0] my_alu = (opcode==’add) ? a + b :
                               (opcode==’not) ? ~a    :
                               (opcode==’sub) ? a - b : 32’bx;
          ```
    - Redundant logic can often reduce the amount of wiring required
- **Wire Minimisation**
    - Generally the job of the pick and place tool not the synthesis system
- **Finite State Machine Minimisation**
    - (1) State Minimisation
        - Remove duplicate / redundant / unreachable states
    - (2) State Assignment
        - Assign a unique binary code to each of the states – the decision of what to do here can reduce the amount of logic required there this can be done optimally.
    - But, SystemVerilog code is very explicit about register usage so little optimisation is possible – higher level behavioural SystemVerilog code would allow for implicit state machine
- **Retiming and D-Latch Migration:** Can move D latches earlier in the logic with little impact to compensate for long logic

Design for FPGAs

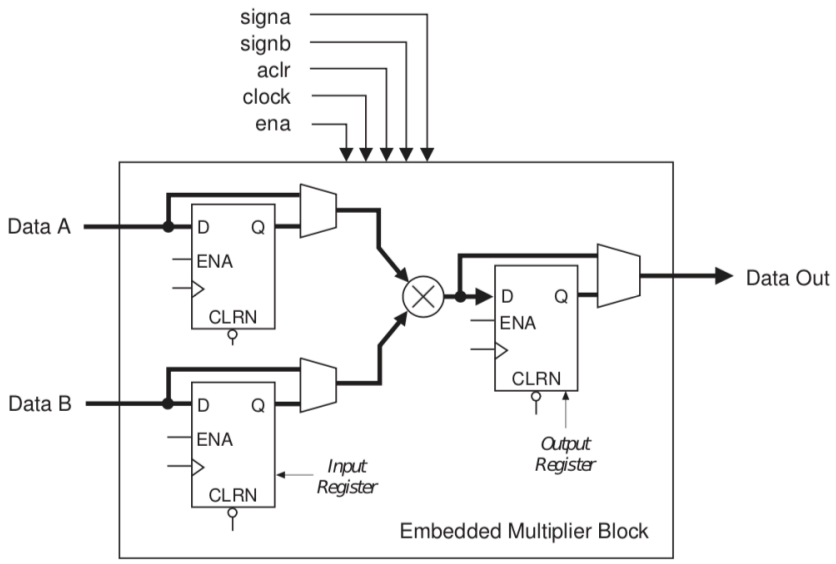
- **Architecture**
    - LUTs – look up tables (typically 4 inputs, one output) – can implement any 4-input logic function
    - LABs – LUT + DFF + muxes
    - Programmable wiring
    - Memory blocks
    - Digital processing blocks
    - I/O
          ```



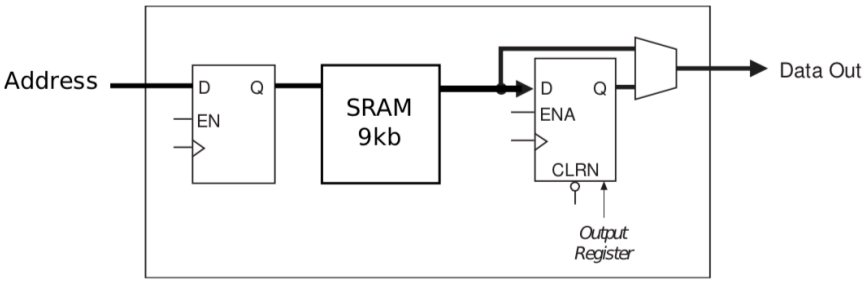
- Interconnection of Blocks



- **Multipliers**



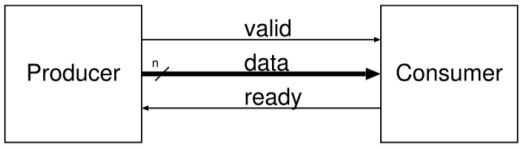
- **Single-Port Memory**



- **Design Flow**
  - Quartus does the synthesis:
  - (1) Compile: SystemVerilog to a netlist of gates and larger blocks
  - (2) Placement: of blocks onto the FPGA
  - (3) Route: wires over the programmable wiring to connect blocks up
  - (4) Timing Analysis: determines the worst-case signal paths
  - (5) PROGRAM THE FPGA: serially shift in the configuration bits which control the wiring switch boxes and the LUTs, etc.
- **Timing Analysis**
  - Finds the worst case delay paths through the circuit between outputs of DFFs and inputs of DFFs – defines the maximum clock frequency
  - There has to be a clock distribution via a special network to ensure that all DFFs receive a clock edge at about the same time
- In Quartus a qsf file specifies the mapping between pins (signals in design to physical pins)
- Sdc (design constraints) file tells us about timing analysis about input clock frequencies and delays on input and output pins.
- **Using Block RAM in SystemVerilog**
  - Sometimes has to be written in a particular style in order that the synthesis tool can easily identify BRAM – no clearing of all BRAM elements & one or two access ports (not three)
  -
  -

```
// BRAM where writes are immediately available
module single_clock_wr_ram(
    output reg [7:0] q,
    input [7:0]      d,
    input [6:0]      write_address, read_address,
    input            we, clk
    );
   reg [7:0]         mem [127:0];
   always_ff @(posedge clk) begin
      if (we)
        mem[write_address] = d;
      q = mem[read_address]; // q does get d in this clock cycle if
                             // we is high
   end
endmodule
```

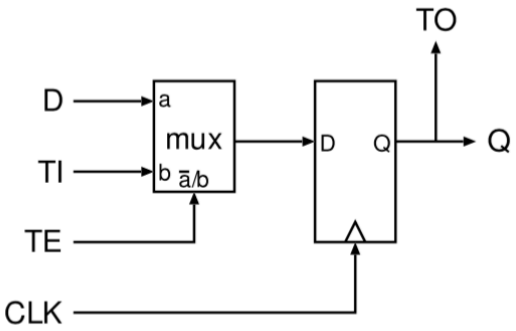
  - Blocking assignment imposes an ordering – writes propagate in 0 cycles
- **System-on-FPGA design with QSys**
  - **Why:** Make systems out of prebuilt blocks
  - **Drag and drop approach and allows us to graphically connect things together**
  - They use standardised interfaces:
    - Avalon Memory Mapped interfaces (processor / memory address + data + control bus)
    - Avalon Streaming Interfaces (point to point channels)



      - push protocol
        - producer pushes data out (data + valid=1)
        - producer sets valid=0 if busy
        - consumer indicates that it has the data by asserting ready=1
        - consumer uses ready=0 if it is busy
  - **Output:** Verilog which is then imported into Quartus
- Processors
  - Hard core: not made of soft FPGA logic
  - Soft processors also exist:
    - RISC-V core that were used in labs
- **SystemVerilog pitfalls**
  - (1) Automatic bus resizing
    - Buses of wrong width get truncated or padded with 0s
  - (2) Wires that get defined by default
    - If you don't declare a wire in an instance it will be a single bit wire by default
  - (3) Pass too few parameters – no error will occur
  - (4) Modules have a flat name space – cannot have multiple identically named modules.
  - (5) Parameter ordering errors – can use .clk(clk) etc though

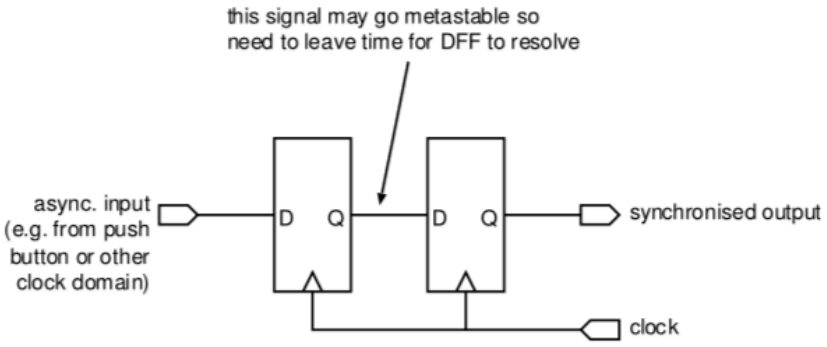
Testing

- **Production Testing:** To check if there are any manufacturing defects
- **Fault Coverage:** For consumer goods, given the cost of testing, something like 98% fault coverage is considered fine, whereas for safety-critical products, we need 100% coverage
- **Fault Models**
    - (1) Logical Faults
        - Stuck-at
        - CMOS stuck-open
        - CMOS stuck-on
        - Bridging faults
    - (2) Parametric Faults
        - Low / high voltage / current levels
        - Gate or path delay-faults
    - (3) Testing Methods
        - Parametric tests also detect stuck-on faults
        - Logical tests detect stuck-at faults
        - Transition tests detect stuck-open faults
        - Timed transition tests detect delay faults
- **Testability**
    - (1) Controllability: the ability to set and clear internal signals – it is particularly useful to be able to change the state in registers inside a circuit
    - (2) Observability: the ability to detect internal signals
- **Fault Reductions**
    - (1) Checkpoints: points where faults could occur
    - (2) Fault equivalence: remove the test for the least significant fault
    - (3) Fault dominance: if every test for fault f1 detects f2 then f1 dominates f2
- Test Patterns are sequences of (input values, expected result) pairs
- Path sensitisation: inputs required in order to make a fault visible on the outputs
- **Test Effectiveness:**
    - Undetectable fault
    - Redundant fault – undetectable fault whose occurrence doesn't affect circuit operation
    - Testability = number of detectable faults / number of faults
    - Effective faults = number of faults – redundant faults
    - Fault coverage = number of detectable faults / number of effective faults
    - Test set size = number of test patterns
    - The goal is to have 100% fault coverage with a minimum sized test set
- **Automatic Test Pattern Generation (ATPG)**
    - Trying to generate a sequence of test vectors (input vector, correct output vector) to identify faults
    - Input and output vectors are typically three valued (0, 1, X)
        - Simple combinatorial circuits can be supplied with test vectors in any order
        - Circuits with memory require sequences of test vectors in order to modify internal state
        - **Scan Path Testing**
            - Make all the D flip-flops in a circuit 'scannable' – add functionality to every DFF to enable data to be shifted in and out of the circuit



simple scan flip-flop

- Testing latches is now easy, and the flip-flops have slices the circuit into small combinatorial blocks which are usually nice and simple to test
  - **Boundary Scan:** just have a scan path around I/O pads of chip or macro-cell
- **JTAG Standard: IEEE 1149 international standard defines 4 wires for boundary scan**
  - tms – test mode select (high for boundary scan)
  - tdi – test data input
  - tck – test clock
  - tdo – test data output
- **Functional Testing:** Ensure the design is functionally correct
  - Simulation provides great visibility and testability and the implementation on the FPGA allows rapid prototyping and testing of I/O
  - Allows test benches to be written to check many cases
  - Gives good visibility of state
  - Is quick to do some tests since no place and route required
  - Slow if simulations need to be for billions of clock cycles
  - Difficult to test if complex I/O behaviour is required
  - **Functional Testing on FPGA**
    - Fast implementation
    - Connected to real I/O
    - Lack of visibility on signals
    - Difficult to test side cases
    - Have to wait for complete place and route between changes
- **Two-Flop Synchroniser**



- mean time between failure (MTBF) can be calculated:

$$MTBF = \frac{e^{\frac{t}{\tau}}}{f_d f_c T_w}$$

  - where:
    - t is the time allowed for first DFF to resolve, so allowing just a small amount of extra time to resolve makes a big difference since it is an exponential term
    - τ=gain of DFF
    - f_d and f_c are the frequencies of the data and clock respectively
    - T_w is the metastability time window
  -

- **SignalTap:** Provide analysis of state of running system
  - Automatically add extra hardware to capture signals of interest
  - Altera's tool to make this easier

Instruction sets and introduction to processor architecture

RISC-V

- **Importance of Instruction Set Architectures**
  - It is the most important interface in the computer system
  - There is a large cost to port and tune all the ISA dependent parts of a modern software stack
  - Hard to recompile / QA all supposedly ISA-independent parts of the stack
  - Most current large chips have multiple ISAs
- **RISC-V**
  - Open Source ISA
  - Developed at UC Berkeley
- Four base integer ISAs
  - RV32E, RV32I, RV64I, RV128I
  - RV32E is a 16-register subset of RV32I
  - Under 50 base hardware instructions
  - There are some standard extensions:
    - M: Integer multiplication / division
    - A: Atomic memory operations
    - F: Single-precision floating-point
    - D: Double-precision floating-point

    - G: General-purpose ISA
    - Q: Quad-precision floating-point
- Fixed length 32-bit instruction format

    -
    - Naturally aligned instructions – rd / rs1 and rs2 are in fixed locations
- Has 32 integer registers called x0 … x31 where x0 = 0 always
- Memory usage
    - Linear address space for both instructions and data
    - Additional page based virtualisation for protection and for simplified memory allocation.

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base	+RV{64,128}
Loads	Load Byte	I	LB rd,rs1,imm	
	Load Halfword	I	LH rd,rs1,imm	
	Load Word	I	LW rd,rs1,imm	L{D|Q} rd,rs1,imm
	Load Byte Unsigned	I	LBU rd,rs1,imm	
	Load Half Unsigned	I	LHU rd,rs1,imm	L{W|D}U rd,rs1,imm
Stores	Store Byte	S	SB rs1,rs2,imm	
	Store Halfword	S	SH rs1,rs2,imm	
	Store Word	S	SW rs1,rs2,imm	S{D|Q} rs1,rs2,imm
Shifts	Shift Left	R	SLL rd,rs1,rs2	SLL{W|D} rd,rs1,rs2
	Shift Left Immediate	I	SLLI rd,rs1,shamt	SLLI{W|D} rd,rs1,shamt
	Shift Right	R	SRL rd,rs1,rs2	SRL{W|D} rd,rs1,rs2
	Shift Right Immediate	I	SRLI rd,rs1,shamt	SRLI{W|D} rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRA{W|D} rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI rd,rs1,shamt	SRAI{W|D} rd,rs1,shamt
Arithmetic	ADD	R	ADD rd,rs1,rs2	ADD{W|D} rd,rs1,rs2
	ADD Immediate	I	ADDI rd,rs1,imm	ADDI{W|D} rd,rs1,imm
	SUBtract	R	SUB rd,rs1,rs2	SUB{W|D} rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm	
	Add Upper Imm to PC	U	AUIPC rd,imm	
Logical	XOR	R	XOR rd,rs1,rs2	
	XOR Immediate	I	XORI rd,rs1,imm	
	OR	R	OR rd,rs1,rs2	
	OR Immediate	I	ORI rd,rs1,imm	
	AND	R	AND rd,rs1,rs2	
	AND Immediate	I	ANDI rd,rs1,imm	
Compare	Set <	R	SLT rd,rs1,rs2	
	Set < Immediate	I	SLTI rd,rs1,imm	
	Set < Unsigned	R	SLTU rd,rs1,rs2	
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm	
Branches	Branch =	SB	BEQ rs1,rs2,imm	
	Branch ≠	SB	BNE rs1,rs2,imm	
	Branch <	SB	BLT rs1,rs2,imm	
	Branch ≥	SB	BGE rs1,rs2,imm	
	Branch < Unsigned	SB	BLTU rs1,rs2,imm	
	Branch ≥ Unsigned	SB	BGEU rs1,rs2,imm	
Jump & Link	J&L	UJ	JAL rd,imm	
	Jump & Link Register	UJ	JALR rd,rs1,imm	
Synch	Synch thread	I	FENCE	
	Synch Instr & Data	I	FENCE.I	
System	System CALL	I	SCALL	
	System BREAK	I	SBREAK	
Counters	ReaD CYCLE	I	RDCYCLE rd	
	ReaD CYCLE upper Half	I	RDCYCLEH rd	
	ReaD TIME	I	RDTIME rd	
	ReaD TIME upper Half	I	RDTIMEH rd	
	ReaD INSTR RETired	I	RDINSTRET rd	
	ReaD INSTR upper Half	I	RDINSTRETH rd	

RV Privileged Instructions

Category	Name	RV mnemonic
CSR Access	Atomic R/W	CSRRW rd,csr,rs1
	Atomic Read & Set Bit	CSRRS rd,csr,rs1
	Atomic Read & Clear Bit	CSRRC rd,csr,rs1
	Atomic R/W Imm	CSRRWI rd,csr,imm
	Atomic Read & Set Bit Imm	CSRRSI rd,csr,imm
	Atomic Read & Clear Bit Imm	CSRRCI rd,csr,imm
Change Level	Env. Call	ECALL
	Environment Breakpoint	EBREAK
	Environment Return	ERET
Trap Redirect	to Supervisor	MRTS
	Redirect Trap to Hypervisor	MRTH
	Hypervisor Trap to Supervisor	HRTS
Interrupt	Wait for Interrupt	WFI
MMU	Supervisor FENCE	SFENCE.VM rs1

Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC	RVI equivalent
Loads	Load Word	CL	C.LW rd',rs1',imm	LW rd',rs1',imm*4
	Load Word SP	CI	C.LWSP rd,imm	LW rd,sp,imm*4
	Load Double	CL	C.LD rd',rs1',imm	LD rd',rs1',imm*8
	Load Double SP	CI	C.LDSP rd,imm	LD rd,sp,imm*8
	Load Quad	CL	C.LQ rd',rs1',imm	LQ rd',rs1',imm*16
	Load Quad SP	CI	C.LQSP rd,imm	LQ rd,sp,imm*16
Stores	Store Word	CS	C.SW rs1',rs2',imm	SW rs1',rs2',imm*4
	Store Word SP	CSS	C.SWSP rs2,imm	SW rs2,sp,imm*4
	Store Double	CS	C.SD rs1',rs2',imm	SD rs1',rs2',imm*8
	Store Double SP	CSS	C.SDSP rs2,imm	SD rs2,sp,imm*8
	Store Quad	CS	C.SQ rs1',rs2',imm	SQ rs1',rs2',imm*16
	Store Quad SP	CSS	C.SQSP rs2,imm	SQ rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD rd,rs1	ADD rd,rd,rs1
	ADD Word	CR	C.ADDW rd,rs1	ADDW rd,rd,imm
	ADD Immediate	CI	C.ADDI rd,imm	ADDI rd,rd,imm
	ADD Word Imm	CI	C.ADDIW rd,imm	ADDIW rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP x0,imm	ADDI sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN rd',imm	ADDI rd',sp,imm*4
	Load Immediate	CI	C.LI rd,imm	ADDI rd,x0,imm
	Load Upper Imm	CI	C.LUI rd,imm	LUI rd,imm
	MoVe	CR	C.MV rd,rs1	ADD rd,rs1,x0
	SUB	CR	C.SUB rd,rs1	SUB rd,rd,rs1
Shifts	Shift Left Imm	CI	C.SLLI rd,imm	SLLI rd,rd,imm
Branches	Branch=0	CB	C.BEQZ rs1',imm	BEQ rs1',x0,imm
	Branch≠0	CB	C.BNEZ rs1',imm	BNE rs1',x0,imm
Jump	Jump	CJ	C.J imm	JAL x0,imm
	Jump Register	CR	C.JR rd,rs1	JALR x0,rs1,0
Jump & Link	J&L	CJ	C.JAL imm	JAL ra,imm
	Jump & Link Register	CR	C.JALR rs1	JALR ra,rs1,0
System	Env. BREAK	CI	C.EBREAK	EBREAK

32-bit Instruction Formats

	31	30 25	24 21	20	19 15	14 12	11 8	7	6 0
R	funct7		rs2		rs1	funct3	rd		opcode
I	imm[11:0]				rs1	funct3	rd		opcode
S	imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode
SB	imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode
U	imm[31:12]						rd		opcode
UJ	imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode

16-bit (RVC) Instruction Formats

	15 14 13	12	11 10	9 8 7	6 5 4 3 2	1 0
CR	funct4		rd/rs1		rs2	op
CI	funct3	imm	rd/rs1		imm	op
CSS	funct3		imm		rs2	op
CIW	funct3		imm		rd′	op
CL	funct3	imm	rs1′	imm	rd′	op
CS	funct3	imm	rs1′	imm	rs2′	op
CB	funct3	offset	rs1′		offset	op
CJ	funct3		jump target			op

CLARVI

- **Processing one instruction**
  - (1) Instruction fetch
    - PC used as an address to do memory access
    - Takes time to do the memory access
    - Result placed in an instruction register
    - PC incremented

- (2) Decode
  - Expands the instruction into a more usable state
    - What kind of instruction is it – does it involve a branch / jump
    - What arithmetic to do
    - Sources and destination registers
- (3) Register fetch
  - Find the source registers
- (4) Branch
  - May be done with a decode or execute
  - This is optional
  - Return address may be stored in a register
- (5) Execute
  - Perform integer or logical operation using an ALU
- (6) Memory access (optional)
  - Use the address to either load data or store data
- Write-back any result to destination register

History of Computer Architecture

- **Hardwired Programming Digital Computers**

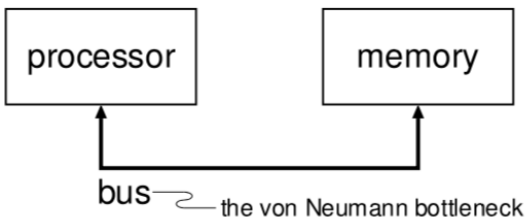
Colossus

Started/completed:	1943/1943
Project leader:	Dr Tommy Flowers
Programmed:	pluggable logic & paper tape
Speed:	5000 operations per second
Power consumption:	4.5 KW
Footprint:	360 feet^2 (approx) + room for cooling, operators
Notes:	broke German codes, info. vital to success of D-day in 1944

ENIAC

Started/completed:	1943/1945
Project leaders:	John Mauchly and J. Presper Eckert.
Programmed:	plug board & switches
Speed:	5000 operations per second
Footprint:	1000 feet^2

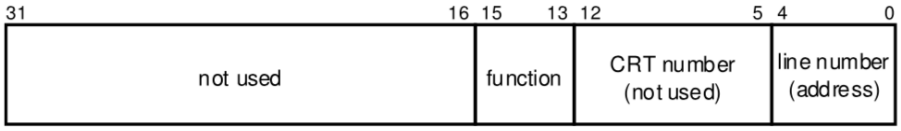
- Birth of a Stored Program Computer
  - John von Neumann – 1945
  - Control-flow Model



  - Summer School of 1946 – UPenn stimulated post war construction of stored-program computers
- Manchester Mark 1 (The Baby)

Demonstrated:	June 1948
Project leaders:	Tom Kilburn and F C (Freddie) Williams
Input/Output:	buttons + memory is visible on Williams tube
Memory:	William Tube 32 × 32 bit words
Logic Technology:	valves (vacuum tubes)
Add time:	1.8 ms
Footprint:	medium room

- Just 7 instructions to subtract, store and conditional jump



function code	*name*	*description*
000	JMP	sets CR to the contents of LINE
100	JRP	adds the contents of LINE to CR
010	LDN	gets the contents of LINE, negated, into ACC
110	STO	stores the contents of ACC into LINE
001 or 101	SUB	subtracts the contents of LINE from ACC, putting the result in ACC
011	TEST	if the contents of ACC are less than zero, add 1 to CR (skip next instruction)
111	STOP	halt the machine

- Williams Tube
  - Used phosphor persistence on a CRT to store information
- EDSAC

Start/End:	1947/1949
Project leader:	Maurice Wilkes
Input/Output:	paper tape, teleprinter, switches
Memory:	mercury delay lines 1024 × 17 bits
Logic Technology:	valves (vacuum tubes)
Speed:	714 operations/second
Footprint:	medium room

  - 18 instructions including add, subtract, multiply, store and conditional jump
- Cray-1
  - Fastest computer in 1976 - $8.8mn
  - 1MB memory
  - 64-bit data, 24-bit address
- Xerox PARC
  - First personal computer to use a bit-mapped graphics and mouse to provide a windows user interface
  - Ethernet
  - Laser printers
- Issues with early machines
  - Technology limited memory size, speed, reliability
  - Instruction set did not support
    - Subroutine calls
    - Floating point operations

        - Little AND, OR, etc
        - No functions (interrupts, exceptions, memory management)
        - Instructions have only one operand
- **Logic Technologies**
    - Valves (Vacuum Tubes)
    - Transistors
        - Bell Labs – Bardeen, Shockley, Brattain – 1947
        - Silicon Junction ($2.50) – 1954
        - First integrated circuit – 1958
        - Resistor Transistor Logic (RTL) – first monolithic chip – 1961
        - CMOS circuits – 1967
- Primary Memory Technologies
    - (1) Mercury delay lines
    - (2) William's tube
    - (3) Magnetic drum
    - (4) Core memory
    - (5) Solid state memories (DRAM, SRAM)
- Secondary Memory Technologies
    - (1) Punched paper cards
    - (2) Magnetic drums, disks, tape
    - (3) Optical (CD, etc)
    - (4) Flash memory
- **Computing Markets**
    - **Servers**
        - High fault tolerance
        - High throughput
        - Low power usage
    - **Desktop Computing**
        - Price vs performance
    - **Embedded Computing**
        - High power efficiency
        - Real-time performance requirements

RISC Processors

- Design Space
    - Complex set of inter-related problems ranging from physical constraints to market forces
    - Hardware is highly parallel
    - Data has spatial and temporal characteristics
- **Design Goals**
    - **(1) Amdahl's Law and the Quantitative Approach to Processor Design**

$$\text{speedup} = \frac{\text{performance for the entire task without using the enhancement}}{\text{performance for entire task using the enhancement when possible}}$$

Amdahl's version: if an optimisation improves a fraction f of execution time by a factor of a then:

$$\text{speedup} = \frac{T_{old}}{((1-f)+f/a)T_{old}} = \frac{1}{(1-f)+f/a}$$

- Eliminating the Semantic Gap
  - In order to improve performance, minimise semantic gap between HLLs and Assembler in order to improve on performance
  - This resulted in CISC

- **MIPS Processor**
  - **Market:** Originally workstations, but now generally embedded / low power systems
  - **Instruction Set:** RISC
  - **Registers:** 32 registers
  - Register File
    - Used to localise intermediate results which improves performance
    - 32x32 bit registers
    - HI and LO registers for the results of mult and div operations
  - Instruction Formats
    - rs: index of first operand register
    - rt: index of second operand register
    - rd: index of destination register
    - shamt: shift amount, used only in shift operations
    - imm: 16-bit signed immediate
    - addr: memory address\

R-type instruction

31:26	25:21	20:16	15:11	10:6	5:0
opcode	rs	rt	rd	shamt	funct

I-type instruction

31:26	25:21	20:16	15:0
opcode	rs	rt	imm

J-type instruction

31:26	25:0
opcode	addr

R-type instructions - sorted by funct
all have operands rs, rt, rd and shamt

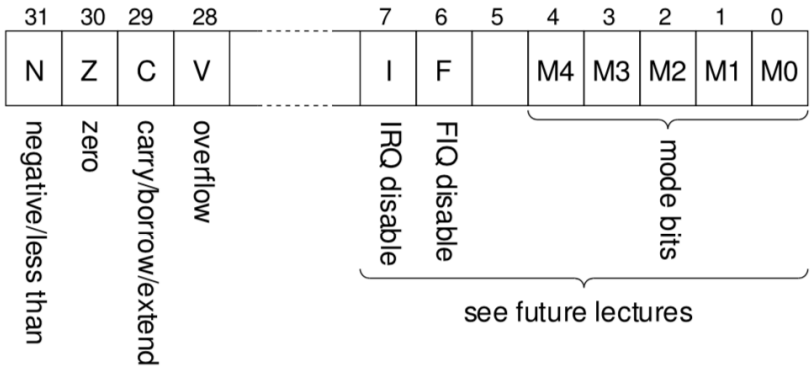
Funct	Mnemonic	Operation
0000 00	SLL	$rd = $rt << shamt
0000 01	SRL	$rd = $rt >> shamt
0000 11	SRA	$rd = $rt >>> shamt
0001 00	SLLV	$rd = $rt << $rs[4:0]
		assembly: sllv rd rt rs
0001 10	SRLV	$rd = $rt >> $rs[4:0]
		assembly: srlv rd rt rs
0001 11	SRAV	$rd = $rt >>> $rs[4:0]
		assembly: srav rd rt rs
0010 00	JR	PC = $rs
0010 01	JALR	$ra = PC + 4; PC = $rs
0100 00	MFHI	$rd = $hi
0100 01	MTHI	$hi = $rs
0100 10	MFLO	$rd = $lo
0100 11	MTLO	$lo = $rs
0110 00	MULT	{$hi, $lo} = ($rs × $rt)
0110 01	MULTU	{$hi, $lo} = ($rs × $rt)
0110 10	DIV	$lo = $rs / $rt
		$hi = $rs % $rt
0110 11	DIVU	$lo = $rs / $rt
		$hi = $rs % $rt
1000 00	ADD	$rd = $rs + $rt
1000 01	ADDU	$rd = $rs + $rt
1000 10	SUB	$rd = $rs − $rt
1000 11	SUBU	$rd = $rs − $rt
1001 00	AND	$rd = $rs & $rt
1001 01	OR	$rd = $rs | $rt
1001 10	XOR	$rd = $rs ⊕ $rt
1001 11	NOR	$rd = ¬ ($rs | $rt)
1010 10	SLT	$rs < $rt ? $rd = 1 : $rd = 0
1010 11	SLTU	$rs < $rt ? $rd = 1 : $rd = 0

MIPS instruction set - sorted by opcode

Opcode	Mnemonic	Operands	Function
0010 00	ADDI	rs rt imm	$rt = $rs + SignImm
0010 01	ADDIU	rs rt imm	$rt = $rs + SignImm
0010 10	SLTI	rs rt imm	$rs < SignImm ? $rt = 1 : $rt = 0
0010 11	SLTIU	rs rt imm	$rs < SignImm ? $rt = 1 : $rt = 0
0011 00	ANDI	rs rt imm	$rt = $rs & ZeroImm
0011 01	ORI	rs rt imm	$rt = $rs | ZeroImm
0011 10	XORI	rs rt imm	$rt = $rs ⊕ ZeroImm
0011 11	LUI	rs rt imm	$rt = { imm, { 16 { 1'b0 } } }

MIPS instruction set - sorted by opcode			
Opcode	Mnemonic	Operands	Function
0000 01	BLTZ ($rt = 0)	rs rt imm	if ($rs $<$ 0) PC = BTA
	BGEZ ($rt = 1)		if ($rs $\geq$ 0) PC = BTA
0000 10	J	addr	PC = addr
0000 11	JAL	addr	$ra = PC+4; PC = addr
0001 00	BEQ	rs rt imm	if ($rs == $rt) PC = BTA
0001 01	BNE	rs rt imm	if ($rs != $rt) PC = BTA
0001 10	BLEZ	rs rt imm	if ($rs $\leq$ 0) PC = BTA
0001 11	BGTZ	rs rt imm	if ($rs $>$ 0) PC = BTA

  - **MIPS Branch Delay Slot**
    - When a branch instruction occurs in assembler, pipelined processors doesn't know what instruction to fetch next
    - Options:
      - (1) Stall the pipeline until the decision is made
      - (2) Predict the branch outcome and branch target. If the prediction is wrong, flush the pipeline to remove the wrong instructions and load in the correct ones
        - Processors using one of these two is hardware interlocked
      - (3) Execute the instruction after the branch regardless of whether it is taken or not. By the time that instruction is loaded, the branch result is known. This is the branch delay slot. The branch therefore does not take effect until after the following instruction
        - Software interlocked
- **Application's View of Memory**
  - An application views memory as a linear sequence of bytes
  - The memory is referenced by addressed
    - Number of bytes from the start of memory
    - Addresses for words must be word aligned (bits 0 and 1 are zeroed)
    - Words are normally stored in little endian but MIPS may be switched to big endian
- **ARM Processor**
  - **Market:** Embedded / low power systems
  - **Instruction Set:** RISC
  - **Registers:** 16 x 32 bit (one is the PC)
  - **Register File**
    - Used to localise intermediate results which improves performance
    - Only a simple and short operand encoding scheme is required which facilitates fixed length instructions and fast decoding



- Don't need to remember ARM instruction formats and mnemonics for the exam
- **Other ARM instruction sets**
  - **Thumb:** First 16-bit instruction set
  - **Thumb2:** 16-bit and 32-bit instruction set
  - **Jazelle DBX:** Direct execution of some Java bytecodes
  - **ARM v8:** 64-bit instruction set – used first time by Apple

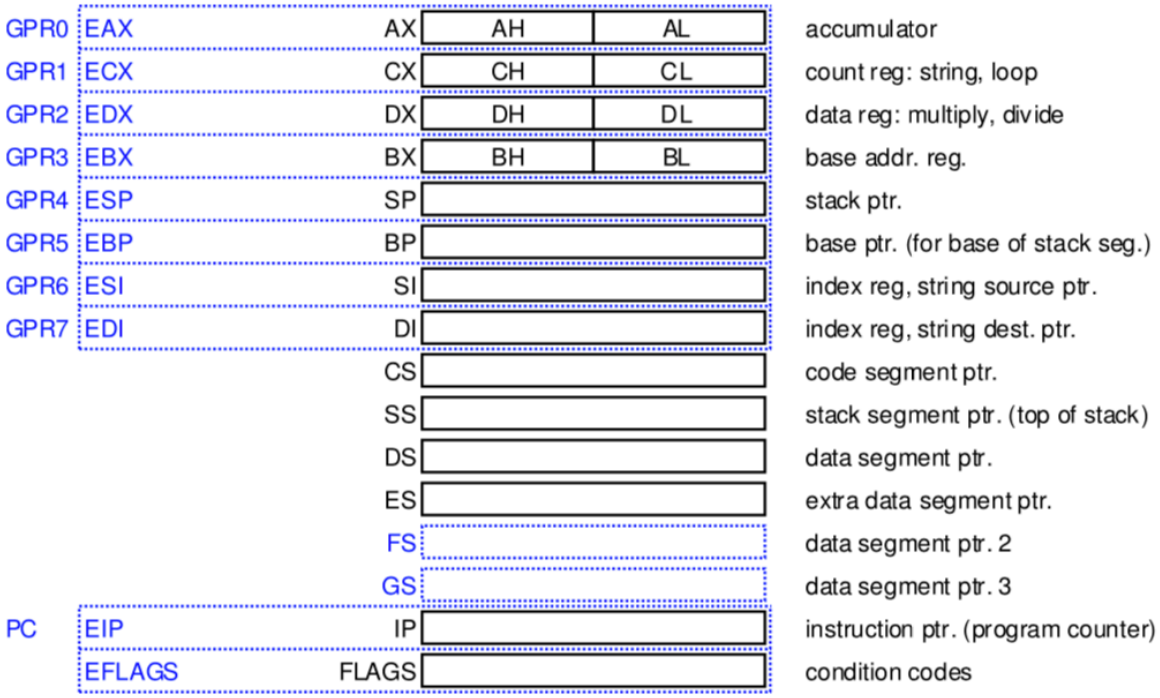
CISC, the Intel ISA, and Java VM

Intel Processors

Name	Year	What did it have?
8086	1978	• 16-bit processor with 16 internal registers and a 16 bit data path • 20-bit address space (1MB)
8088	1978	• Same as 8086 but 8-bit external data path
8087	1980	• Floating-point coprocessor for 8086 based on extended stack architecture
80186 and 80188	1982	• Reduced need for so many external support chips
80286	1982	• Extended addressing to 24-bits • Introduced memory protection model • Has a 'Real Addressing' mode for executing 8086 code
80386	1985	• Extended architecture to 32-bits (32-bit registers and address space)

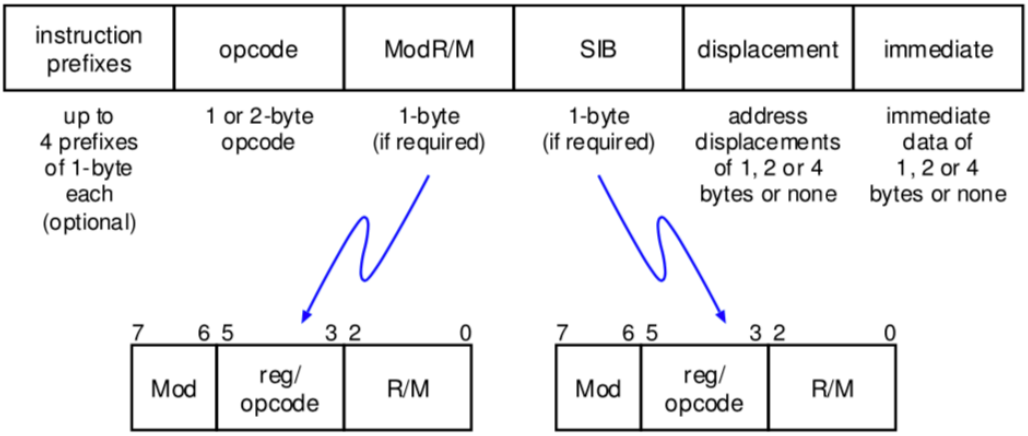
		• Added new addressing modes and additional instructions • Makes register usage more general purpose
80387		
80486	1989	• Floating-point unit and caches on the same chip
80486X		• Floating-point is disabled – crippleware
Pentium (80586)	1993	• Superscalar and larger caches
P6 Family (Pentium Pro, Pentium II, Celeron)	1995	
Pentium 4	2000	
Xeon	2001	
Core, Core 2	2006	
Atom and Core i7, i6, i3	2008	

- **Integer Registers**



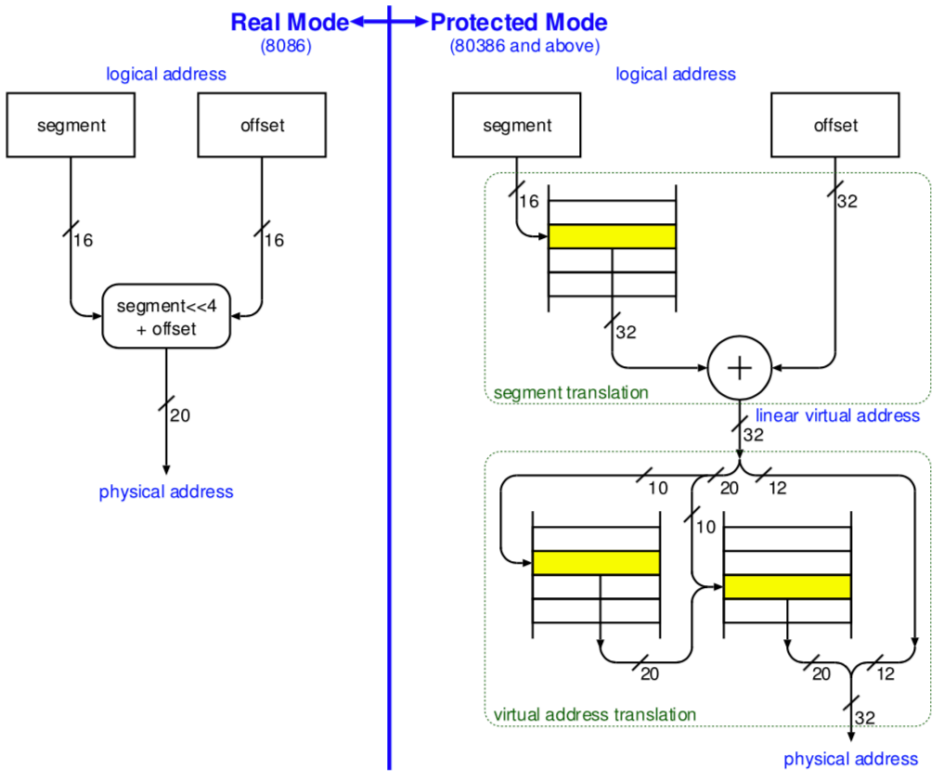
- **Floating Point Registers**
  - 8x80-bit registers implementing a stack
  - Floating-point instructions have one operand as the top of stack and the other operand as any of the 8 floating-point registers
    - Also used to store byte and word vectors for MMX (typically multimedia) operations
- **General Instruction Format**

- Variable length instructions typified by CISC – this is more complex to decode that RISC but can be more compact
- Registers are specialised but are getting increasingly general purpose

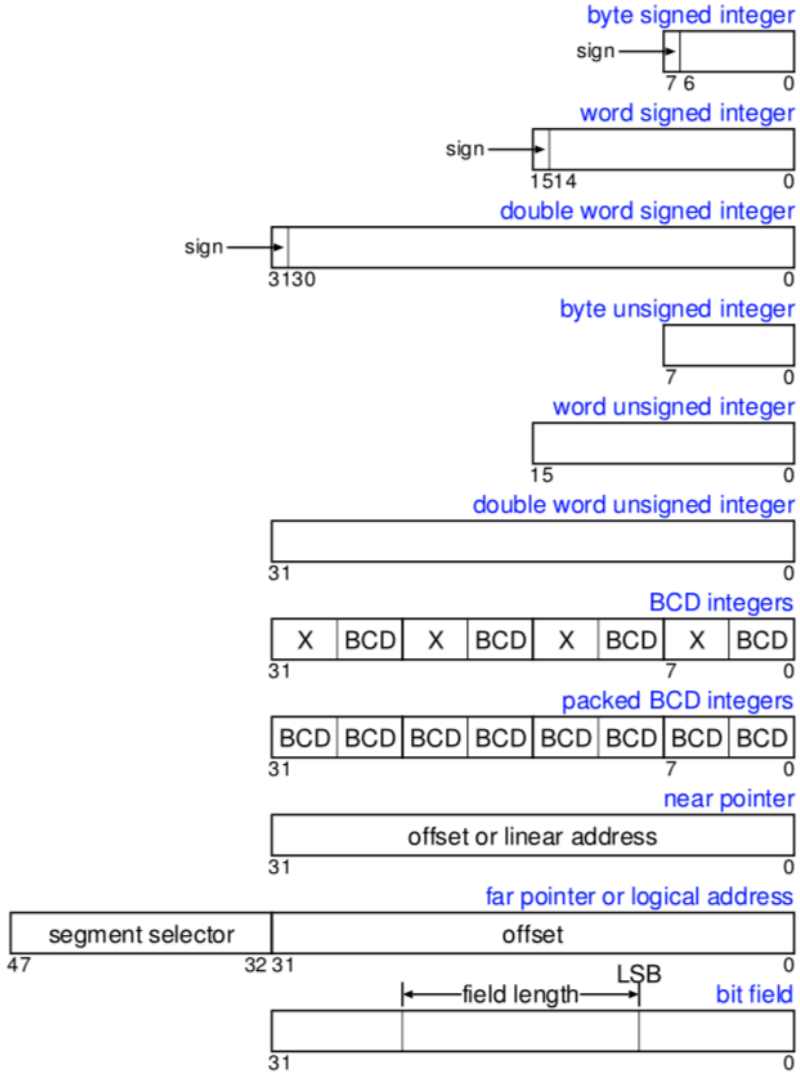


- The prefixes modify the main instruction behaviour
- The opcode specifies the instruction with the possibility of 3 additional bits from the ModR/M field
- ModR/M byte – specifies the addressing mode and up to 2 registers
- SIB – extends ModR/M

- **Addressing Modes**



- **Data Types**



- BCD Integers: stores digits 0…9 in 4-bits (packed) or 8-bits (unpacked)
- Bit fields: A contiguous sequence of bits which can begin at any position of any byte in memory and can contain up to 32-bits.
- Strings: contiguous sequence of bits, bytes words or doublewords.
  - Bit string can begin at any bit position of any byte and can contain up to 2^{32}-1 bytes
  - Byte strings can contain bytes, words, or doublewords and can range from 0 to 4GB
- Floating-point types:
  - Single (32-bit), double (64-bit), extended (80-bit) reals
  - Word (16-bit), short (32-bit), long (64-bit) binary integers
  - 18-digit BCD integer with sign
- MMX
  - Packed 64-bit data types to support multi-media operations

- **Procedure Calling Convention**
  - Parameters passed in general purpose registers
  - Some or all of the parameters can also be passed on the stack
  - CALL pushes the EIP (PC) onto the stack and then jumps to the procedure

  - RET pops the value off the stack into EIP and therefore returns
  - ENTER allocates stack space and sorts out stack frame pointers
  - LEAVE is the reverse of ENTER
- **Software Interrupts**
  - INT_n – raises interrupt or exception n (n=128 for Linux OS call)
  - IRET – return from interrupt
  - INTO – raises overflow exception
  - BOUND – compares a signed value against upper and lower bounds and raises a bound exception if out of range
- **Intel 64**
  - 64-bit flat linear addressing: no segment registers in 64-bit mode (old CS, DS, ES and SS segment registers treated as 0)
  - 64-bit wide registers and instruction pointers – 8 general purpose and 8 additional registers for streaming extensions
  - New instruction-pointer relative addressing mode
- **Virtual Machine Support**
  - Allows different operating systems to run simultaneously on the same machine
- **Other recent additions**
  - (1) Transactional Memory Support
  - (2) Security Extensions
  - (3) Memory Protection Extensions

Java Virtual Machine

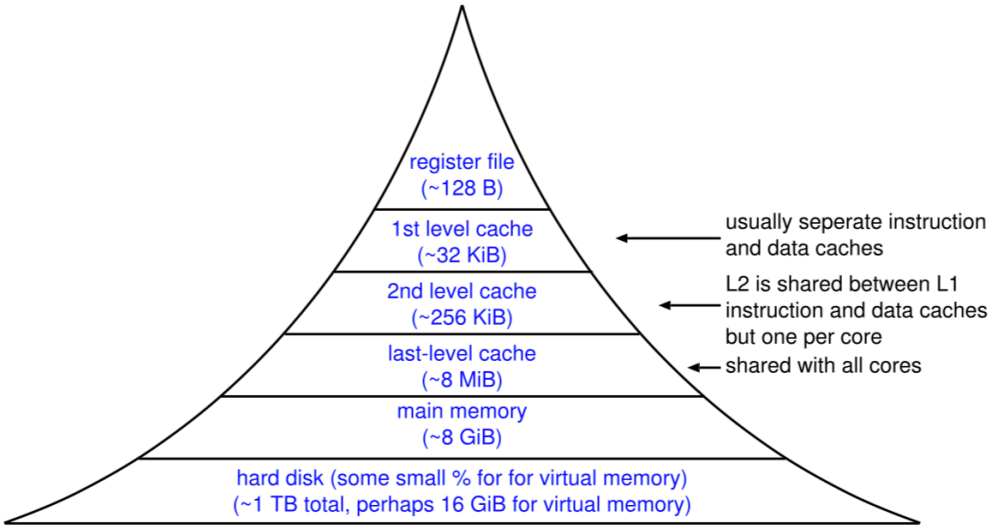
- Virtual processor specification for running Java programs
- Instruction set architecture is stack based with variable length byte code instructions
- Why?
  - Portability
  - Easy to port to new architecture (just rewrite new interpreter)
  - Allows one binary to be shipped for all architecture
- **Primitives**

byte	— 1-byte signed 2's complement integer
short	— 2-byte signed 2's complement integer
int	— 4-byte signed 2's complement integer
long	— 8-byte signed 2's complement integer
float	— 4-byte IEEE 754 single-precision float
double	— 8-byte IEEE 754 double-precision float
char	— 2-byte unsigned Unicode character
object	— 4-byte reference to a Java object
returnAddress	— 4 byte subroutine return address

- **Registers**: Program Counter, Vars (register holding base address of memory where local variables are held), optop (register holding address of top of operand stack), frame (register holding address of base of current frame – holds environment data)
- **Frame**: Used to hold data for a method being executed – contains local variables, operand stack and other run time data.
  - Created dynamically each time a method is invoked and is garbage collected
- **JVM Instructions:** first byte = opcode = 8-bit, subsequent n bytes are operands where n can be determined from the instruction
  - For a list of actual instructions, see lecture notes

Memory Hierarchy

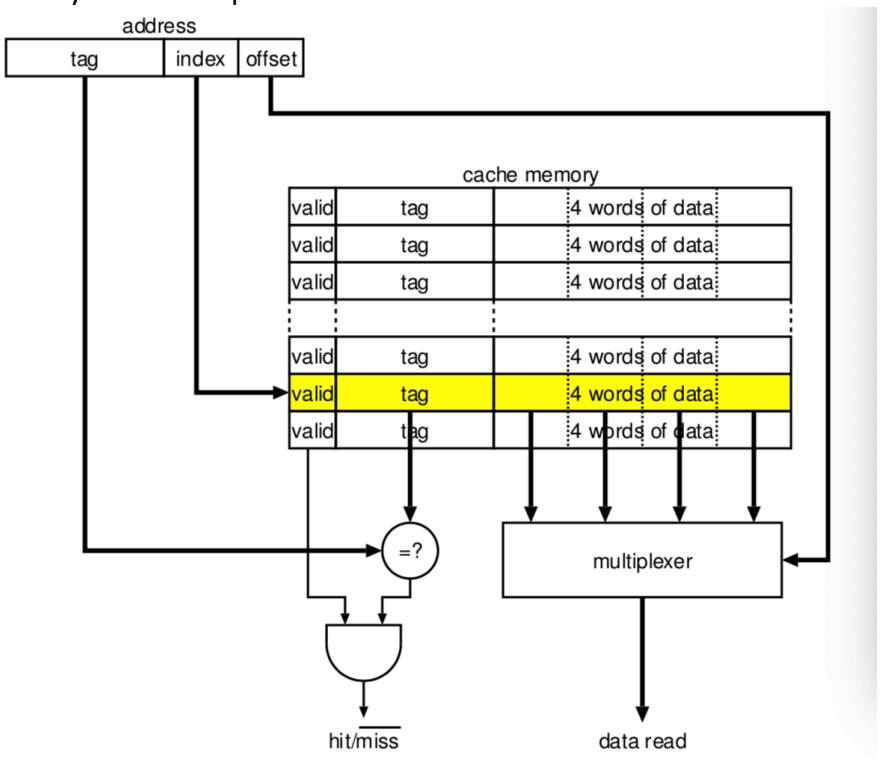
- **Diagram of Caches**



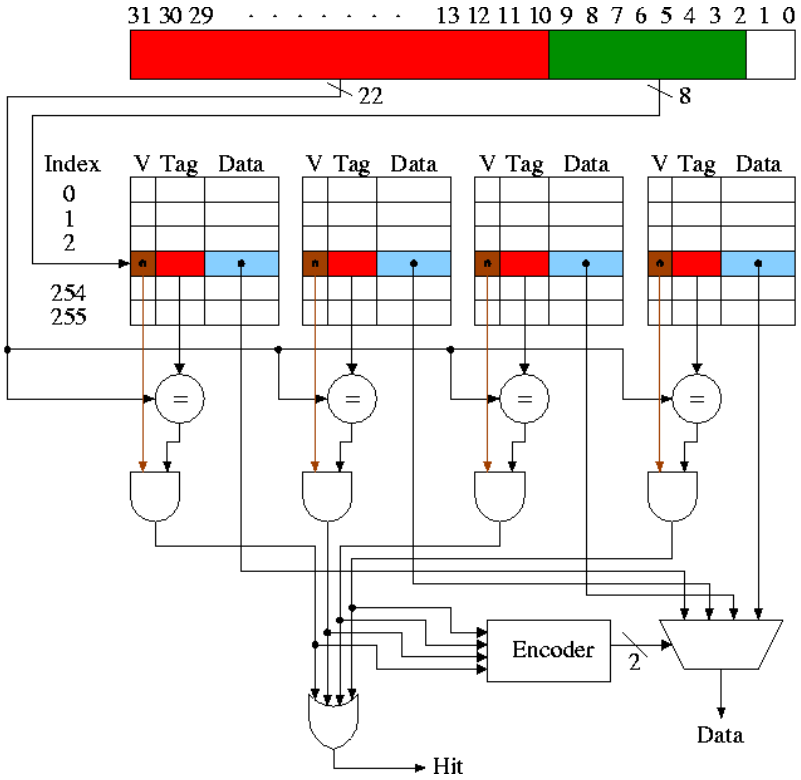
- **Memory Technologies**
  - SRAM
    - Maintains store provided power kept on
    - 6-8 transistors per bit
    - Fast but expensive
  - DRAM
    - Relies on storing charge on the gate of a transistor
    - Charge decays over time so requires refreshing
    - 1 transistor per bit
    - Fairly fast, not too expensive
  - ROM – read only memory
  - PROM – programmable ROM
  - EPROM – erasable PROM (though takes 20 minutes to erase)
  - EEPROM – electronical erasable PROM – faster erase
  - Flash memory – bank erasable
    - Slow write, fairly fast read
  - Magnetic disk
- **Definitions**
  - **Latency:** number of cycles to wait for data to be returned
  - **Bandwidth:** amount of data returned per cycle
  - **Register file:** Small SRAM with < 1 cycle latency and multiple reads and writes per cycle
  - **First Level Cache:** Single or multi-ported SRAM: 1-3 cycle latency
  - **Second Level Cache:** Single ported SRAM – around 3-9 cycles latency
  - **Main Memory:** takes somewhere **between 10 and 100 cycles to get first word but can then receive adjacent words every 2 to 8 cycles**
    - Single write takes 8 to 80 cycles, further consecutive words every 2 to 8 cycles
- **Cache Principles**
  - **(1) Temporal Locality:** If a word is accessed once then it is likely to be accessed again soon

- **(2) Spatial Locality:** If a word is accessed then its neighbours are likely to be accessed soon
  - To make use of spatial locality we store neighbouring words to form a cache line – typically 4 or 8 words aligned in memory
  - Also has the advantage that a group of words can be read from DRAM in a burst
  - And reduces translation logic

- Possible cache designs
  - (1) Naïve Cache Design – fully associative, one-word cache lines
    - Allow any word to be stored anywhere in the cache
    - You need to look up an item by its address – huge overhead in storing this
  - (2) Direct Mapped Cache
    - Use one part of the address to map directly onto a cache line so a word can only live in one place in the cache



    - Breaks if lots of things which are all being used map to the same index
  - (3) Set Associative Cache
    - Direct mapped cache but with a set of cache lines at each location – look-up the data in the direct mapped cache and used the appropriate valid tag to indicate which element to read from
    - So a word can be stored in n places

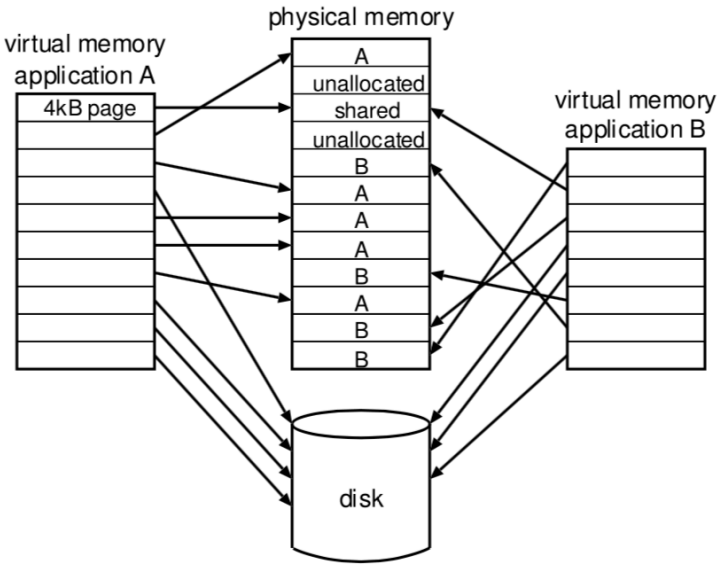


- Victim Buffer – a one-line cache line buffer to store the last line overwritten in the cache
  - Used to augment a direct mapped cache to give it a small amount of associativity
- Victim Cache
  - A small fully associative cache used in conjunction with a direct mapped cache
- Cache Line Replacement Policies
  - Least Recently Used: good but hard to implement
  - Not Last Used: pass a pointer and keep passing it when a cache line is accessed
  - Random: tends to work quite well
- **Writing Strategies**
  - Fetch on write: block is first loaded into the cache and the write is performed
  - Write around: if it isn't in the cache then leave it uncached but write result to the next level in the memory hierarchy
  - **Write through**
    - Data is written to both the cache and the lower level memory
    - So, if a cache line is replaced, it doesn't need to be written back to the memory
    - Write through is common for multiprocessors, so cache coherency
  - **Write back**
    - Data written to cache only
    - Data only written to the lower level memory when its cache line is replaced
    - Dirty bit used to indicate if the cache line has been modified
  - Write buffers
    - Writing to lower level memory takes time however, to avoid the processor stalling, a write buffer is used to store a few writes
- **Virtual and Physically Addressed Caches**
  - Address translation takes time – do not want to introduce extra latency

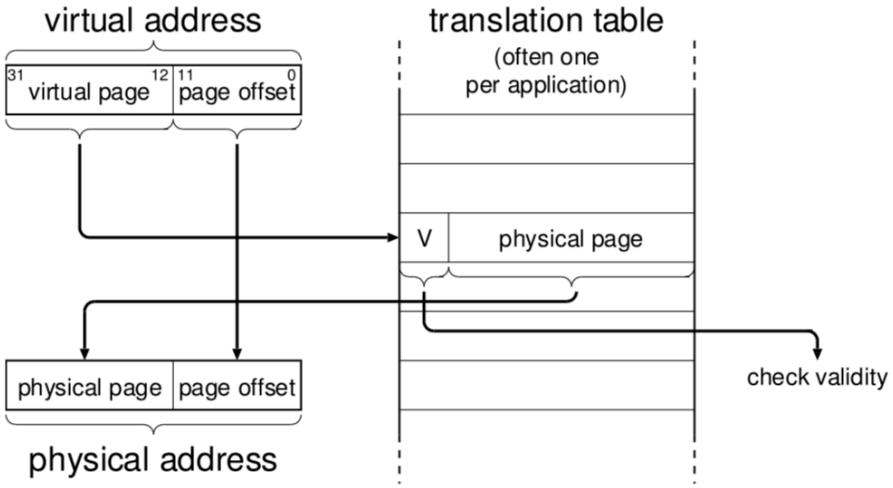
    - Virtual and physical addresses only differ in the upper bits so lower bits of virtual address are sufficient to access the cache without conflict
    - If cache is bigger than a virtual page, then
        - (1) translate virtual to physical and then use physical address to access cache (adds latency)
        - (2) use the virtual address to access the cache, perform address translation concurrently and compare physical tag with tag in cache memory
            - Need to be careful about multiple virtual addresses to one physical address (memory sharing) – cache aliasing

Hardware for OS Support

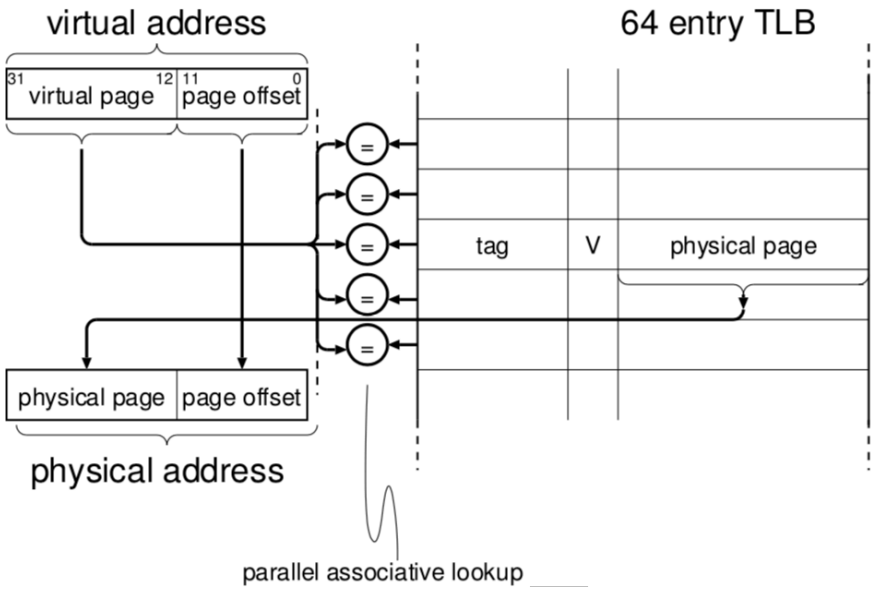
- **Exceptions**
    - Errors cause an exception to occur – terminates current flow of execution and invokes an exception handler
    - Software exceptions are caused by inserting a special instruction (SYSCALL on the MIPS, SWI on the ARM) which can be used to make an operating system call.
    - Interrupts have a similar effect to exceptions except they are caused by an external signal
    - **Possible Exceptions**
        - Alignment Fault – trying to read a 32-bit word from an address which is not on a 4-byte boundary
        - Translation Fault – a TLB miss occurred and the relevant translation information was not present in the translation structure
        - Domain Fault – the TLB entry does not have the same subdomain as the currently executing application
        - Permission Fault – current application does not have the correct read or write permissions to access the page
    - **MIPS**
        - k0 and k1 reserved for kernel use
        - The PC is stored in the exception PC on coprocessor 0
        - Then return to the program
- **Operating Modes:** When interrupt occurs, processor is switched into an alternative mode which has a higher privilege
- **Virtual Addressing**



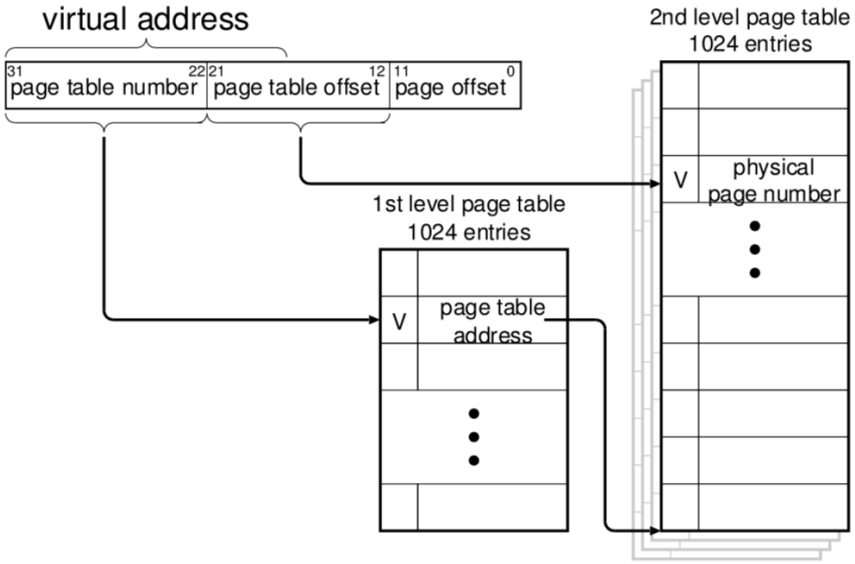
- What an application uses to address its memory
- Must be converted to physical addresses so that they reference a piece of physical memory
- Translation usually performed on pages – upper bits of a virtual address correspond to the virtual page and the lower bits specify the index
- If there is insufficient memory, then some pages may be swapped to disk – if accessing a page from disk, then causes an exception which invokes the operating system



- However, this would require a very large translation table – therefore use a Translation Look-aside Buffer (TLB)



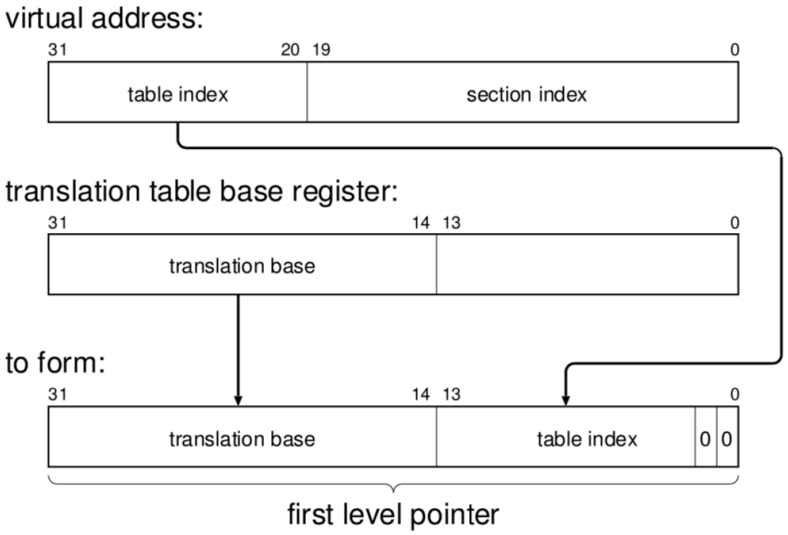
- But associative TLBs don't scale so they can only cache recently performed translations so that they may be reused
- **Multilevel Page Tables**



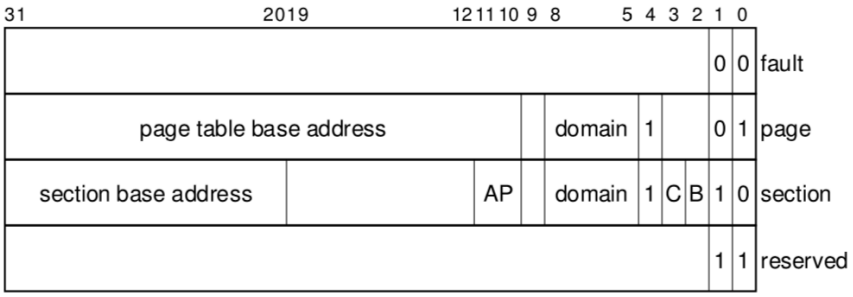
- **Inverted Page Tables**
  - Have a page table with an entry per physical page – each entry contains (valid, processes ID, virtual page number)
  - Compact table
  - Have to search the table to match the virtual address
- Translation table base register adds a translation base and is set by the operating system on a context switch
- **Single or Multiple Virtual Address Spaces**
  - **Multiple:** Each application resides in its own separate virtual address space.
    - Used by UNIX
    - Makes sharing libraries and data a little bit harder
  - **Single:** Only one virtual address space
    - Force linking at load time

- Sharing libraries and data is much simpler

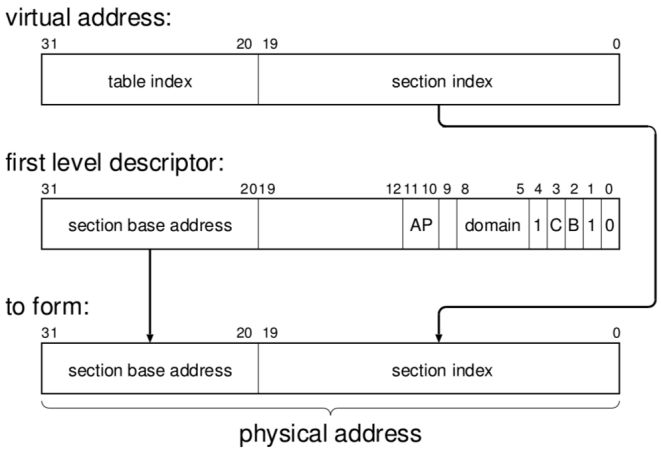
- **Memory-Mapped I/O**
  - I/O devices are usually mapped to part of the address space
  - Memory protection used to ensure that only the device driver has access
  - Device drivers can be in user mode or kernel mode depending on the OS
  - Some processors have special instructions to access I/O within a dedicated I/O address space



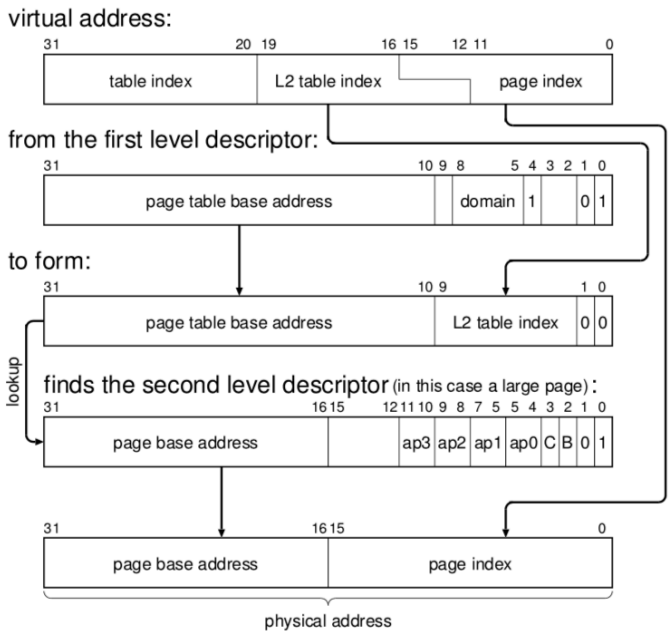
  - The first level pointer is used as an address to lookup the first level descriptor



where domain identifies which of the 16 subdomains this belongs to
C & B control the cache and write-buffer functions (more later)
AP controls the access permissions — what can be read and written in User and Supervisor modes.



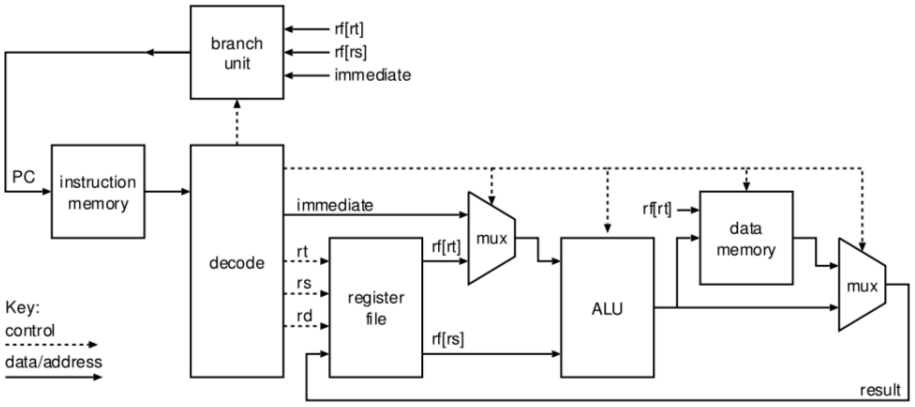
  - You then find the second level pointer by prepending the L2 table index



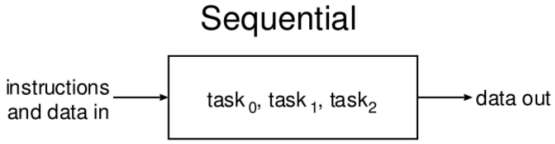
- TLB entry also contains the protection data

Pipelining

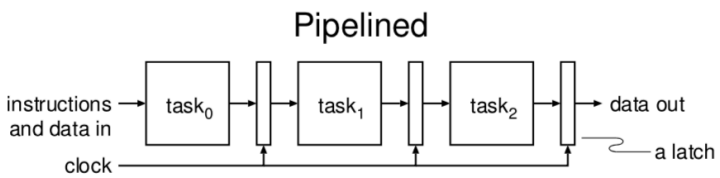
- MIPS Data Movement



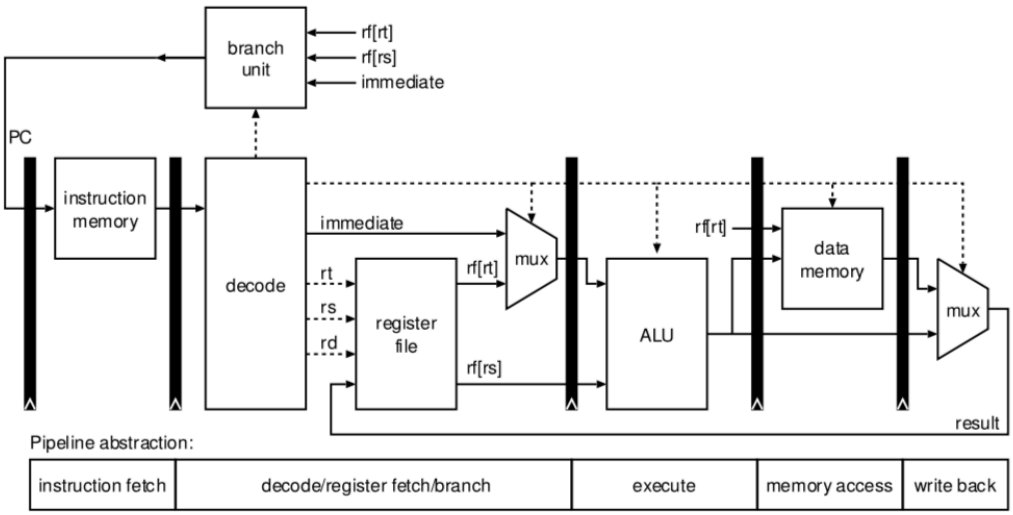
- Sequenced vs Pipelined



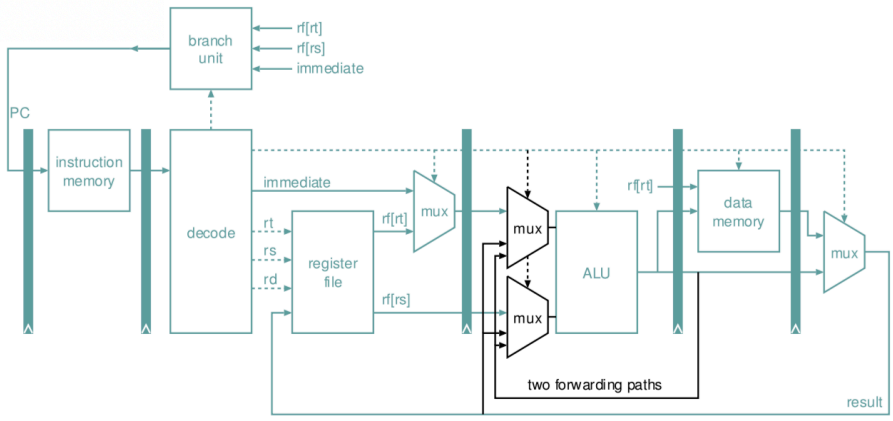
maximum latency = $\ell_s = \sum_{i=0}^{2} maxtime(task_i)$
maximum frequency = $\frac{1}{\ell_s}$



maximum task time = $mt = \mathbf{MAX}_{i=0}^{2} maxtime(task_i)$
maximum latency = $\ell_p = 3 \times (mt + latch\_time)$
maximum frequency = $\frac{1}{mt + latch\_time}$

- MIPS Pipeline
  - 

- **Data Hazards**
  - Sometimes there can be data hazards when it is possible that registers don't contain the most up to date values – things haven't been written back to yet, etc.
  - Conditional branches in particular
  - This can be fixed by:
    - **Stalling –** adding a no-operations stages into the pipeline
    - **Forwarding –** adding a bypass path to get the result to the execute stage faster



    - ARM7 – allow ALU to be used to calculate the branch target in the case of a conditional branch – values available via bypass network
- **Interlocks**
  - Makes the state of two things dependent on one another
  - Hardware Interlocks
    - Preserve a simple sequential programming model but adds complexity to hardware
    - Add flags to show if register is ready
  - Software Interlocks
    - Expose the load delay slot to programmer
    - Lose the simple sequential model
    - Architecturally dependent

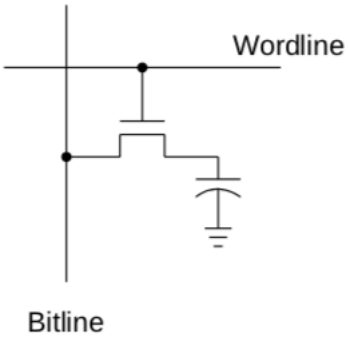
- **MIPS (Microprocessor without interlocked pipeline stages)**
  - Originally all hazards resolved in software

System-on-chip architecture

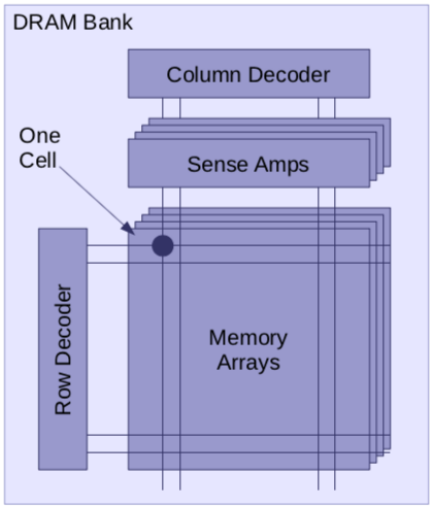
Definitions

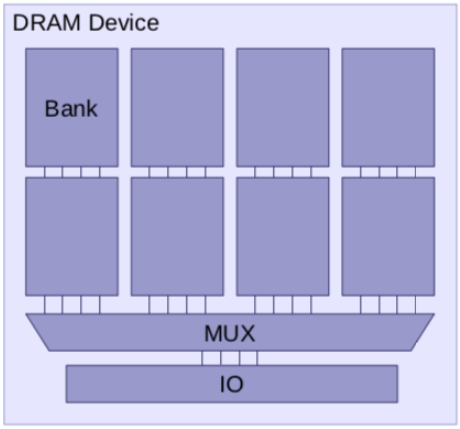
- **SOC:** Ensemble of processors, memories and interconnects tailored to an application domain
  - An SoC is designed for a particular application domain, such as mobile platforms, smart TVs, networking or servers
  - By placing things on the same chip, it increases (1) integration, (2) reduces latency between the different components and (3) can be customised
  - (4) Can also be build from different parts from different vendors
- **Parallelism**
  - In 1972, Michael Flynn proposed a taxonomy of different types of parallel hardware according to the parallelism that was found
  - **(1) SISD**
    - Uniprocessor system
  - **(2) MISD**
    - Multiple streams of instructions operating on a single stream of data
  - **(3) SIMD**
    - Single instruction, multiple data – for example in GPUs and other HPC workloads
  - **(4) MIMD**
    - Multiple streams of information, multiple streams of data
  - **Amdahl's Law**
    - Can be used to estimate speedup for a workload when parallelising part of it – implies we need to write very parallel programs to make best use of the cores
    - Assumes a fixed problem size
    - n is the number of cores
    - B is fraction of workload that remains sequential
    - $Speedup(n) = \frac{1}{B+\frac{1-B}{N}}$
  - **Gustafson's Law**
    - Also shows speedup when cores increase
    - Assumes a fixed execution time
    - Can be applied when a workload can be scaled up with increased compute
    - $Speedup(n) = n = (1-n)B$
- **Memory Placement**
  - On chip
    - Tighter integration is good for latency and bandwidth
    - Less need for a cache
    - May be cheaper
  - On separate chip
    - Space is a premium
    - DRAM process technologies differ from standard

        - Off-chip is more generic
    - Other memory considerations
        - Virtual memory support?
        - Should memory be centralised or distributed?
        - How does programmer see memory – shared memory or message passing
- **Chip Stacking:** Can have package on package – chip put on SoC with a ball grid array. The other alternative is 3D die stacking where the silicon wafers are directly placed and silicon vias connect the dies.
- **DRAM vs SRAM**
    - DRAM is denser, but slower
    - DRAM operation

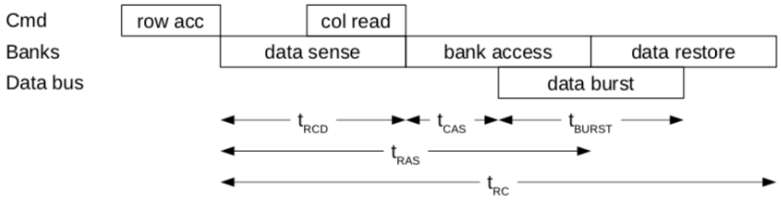


        - A single cell is composed of a capacitor to hold charge and a transistor to read from and write to it – therefore needs a periodic refresh
        - DRAM cells organised into array, and multiple arrays into a bank.
        - Banks grouped into devices, which can be arranged into ranks – therefore some combination of concurrency (ranks and banks) and synchronisation (devices and arrays)





- Memory controller is responsible for turning the memory requests from the processor into the low-level commands that actually do the reading and writing into the DRAM cells
    - Each command takes a certain amount of time – the control issues each command so in sequence to minimise this time



Key DRAM Timings

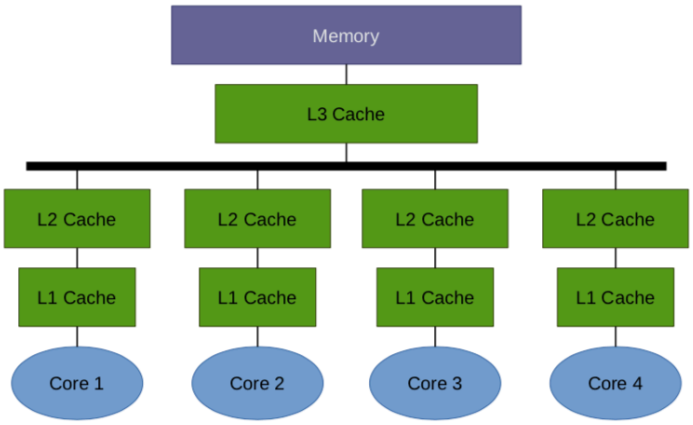
Timing	Description
t_{BURST}	Time that a data burst takes on the data bus
t_{CAS}	Column address / access strobe – time between column access command and the start of data from the DRAM device
t_{CMD}	Time that a command takes to go from the controller to devices
t_{RAS}	Row address / access strobe – time between row access command and data restoration in the DRAM array
t_{RC}	Row cycle – time interval between accesses to different rows in the sam bank, so $t_{RC} = t_{RAS} + t_{RP}$
t_{RCD}	Row to column command delay – time between row access and data ready at the sense amplifiers
t_{RP}	Row precharge – time for an array to be precharged for another row access
t_{WR}	Write recovery – minimum time between a write burst and start of precharge

- DRAM commands
    - Row access – move data to sense amps then back
    - Column read – move data from sense amps to bus
    - Column write – move data from bus to sense amps
    - Precharge – reset sense amps and bitlines
    - Prefresh – read out data then restore
- Open-Page vs Close-Page
    - Sense amps act like buffers – store the most recent row that was accessed. To access another row, they must be reset
    - Open-Page

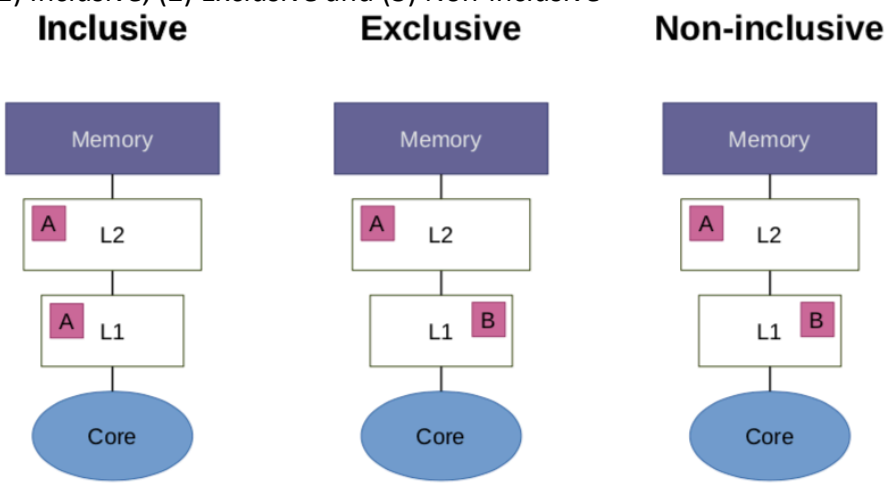
    - Reduced latency when reading from that row again
    - Great for sequential access to different columns
    - Requires explicit reset when accessing another row
- Closed-Page
    - Favours access patterns with little locality

Multicore Processors

- **Shared Memory**
    - By using the same memory, writes by one core are eventually seen by others
    - This doesn't have to happen instantly, it means that reads always get the right value
    - Different cores can use memory to communicate
    - Easy to program
- **Multicore Caching**
    - Add a cache hierarchy to speed up memory accesses
    - **Shared Caches:** Data shared between cores faster than memory
    - **Private Caches:** Provide guaranteed speed for each space and may be faster to access
    - **Multilevel Shared and Private Caches**

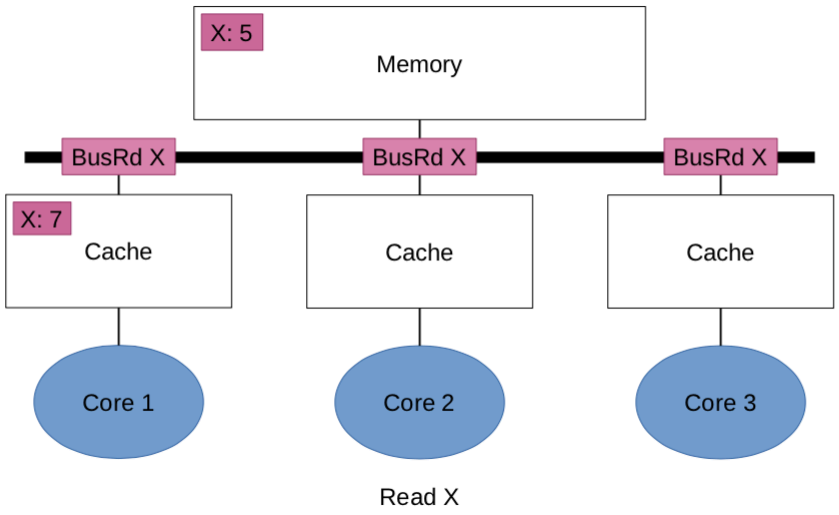


    - **Cache Inclusion**
        - Duplicate data at different cache levels
        - A miss may not have to go all the way to memory, but it is also wasteful
        - (1) Inclusive, (2) Exclusive and (3) Non-inclusive



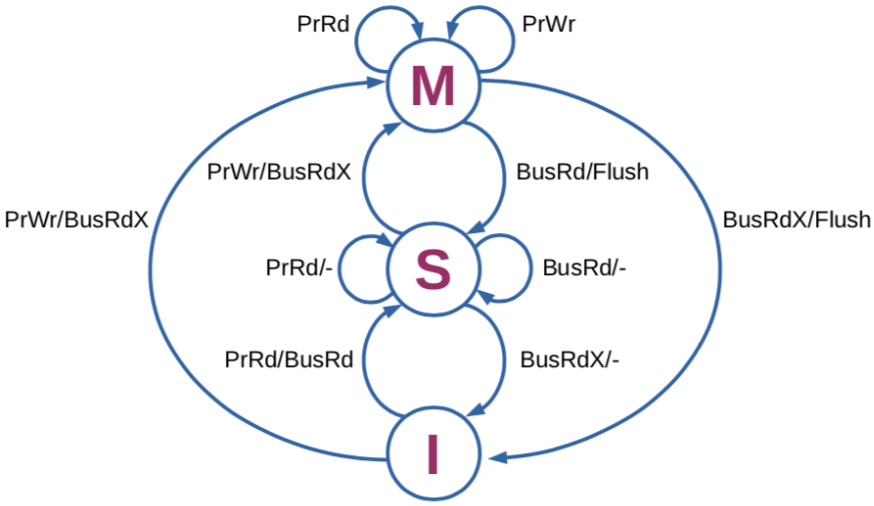
    - **Cache Coherence Protocol**

- Implement hardware to propagate values to keep the caches and the memory coherent
- We assume we have a snoopy bus – that is caches can see (snoop) on other cores' transactions
- There is no cache-to-cache sharing



- MSI Protocol
  - Three states:
    - (1) Modified
      - Single up-to-date copy in this cache
      - Value in the main memory is stale
    - (2) Shared
      - Copy in the cache matches the memory
      - May be present in other caches
    - (3) Invalid
      - No copy in this cache

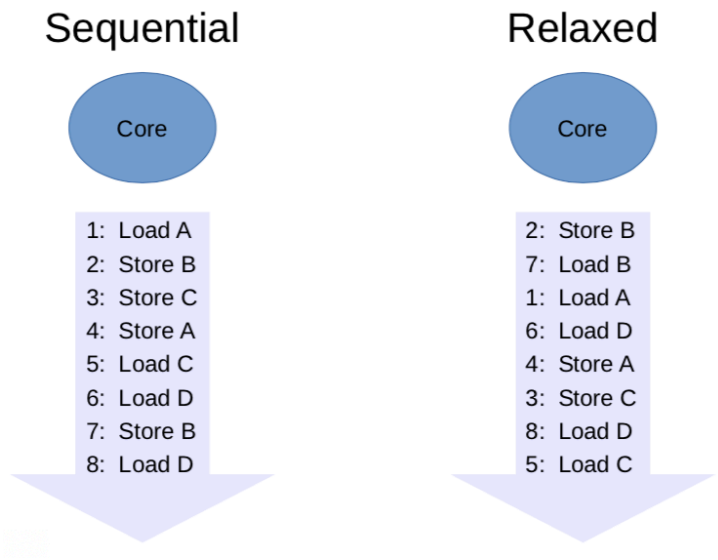
Short Name	Long Name	Description
PrRd	Processor Read	The processor attached to this cache wants to read a value
PrWr	Processor Write	The processor attached to this cache wants to write a value
BusRd	Bus Read	A read request transaction is placed on the bus
BusRdX	Bus Read Exclusive	An exclusive read request transaction is placed on the bus
Flush	Flush Data	Dirty data from the cache is flushed back to main memory



- The state of a line in one cache limits the states the same line in a different cache can have:
  - M => only I
  - S => S and I
  - I => anything

- **MSI Extension**
  - The basic MSI protocol requires us to go to memory to get data – however, if other caches have it, we should get it from that cache.
  - Therefore, open up cache-to-cache sharing
- However, snoopy protocols don't scale well – directory-based schemes can be used instead.

- **Memory Consistency**
  - What ordering do we see between reads and writes from another core?
  - Each location in memory is independent
    - What about ordering of writes to different addresses – or reads and writes to different addresses
  - All writes are eventually seen by all cores
  - Coherence (ensure that cached data written by a core is seen by others) vs Consistency
  - **Memory Consistency Model**
    - Can have some relaxed consistency – some loads and stores can bypass each other
    - However, we want the illusion of sequential consistency – all reads and writes by a single processor are seen in the order they occur.



- Relative ordering between operations to the same address is maintained
- However, loads and stores to different addresses can bypass each other

- However, need sequential consistency for shared data
- **Synchronisation Primitives**
  - Instructions to provide synchronisation – generally not seen by an application programmer
  - (1) Memory Barriers
    - Guarantee ordering of memory operations – within a core
    - All prior memory operations complete before the barrier finished execution
    - Store after a barrier can't overtake a load before it – sometimes called memory fence instruction
  - (2) Read-Modify-Write
    - The most basic class of atomic operation
    - These provide the ability to (1) read a memory location and (2) simultaneously write a new value back
    - Lots of examples:
      - **(1) atomic exchange**
        - Read and write in one uninterruptable instruction (this is particularly difficult in RISC machines)
        - Load Linked / Store Conditional

```
atomic_exch X, r1
```

```
load_linked r0, X
store_conditional r1, X
```

        - Store only succeeds if X hasn't changed – any write to X causes the store to fail
        - The source register contains 1 on success, 0 on fail
        - **Exchange Code**

```
xchg: mov  r3, r0
      ll   r4, 0(r1)
      sc   r3, 0(r1)
      beqz r3, xchg
      mov  r0, r4
```

- (2) test-and-set
- (3) compare-and-swap
- (4) fetch-and-add

```
fadd: ll   r4, 0(r1)
      add  r4, r4, 1
      sc   r4, 0(r1)
      beqz r4, fadd
```

- (3) Spin Locks
  - Lock

```
lock: mov  r3, #1
      ll   r4, 0(r1)
      sc   r3, 0(r1)
      beqz r3, lock
      bneq r4, lock
```

  - Unlock

```
unlock: mov r3, #0
        st  r3, 0(r1)
```

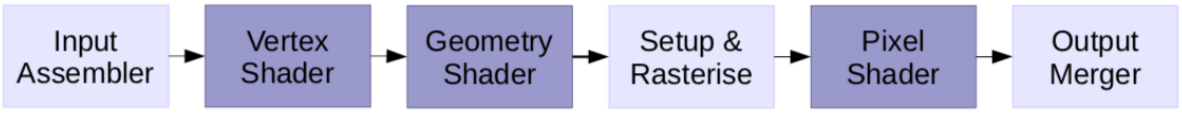
  - The naïve implementation causes lots of bus traffic – also need to avoid writing unless the write is likely to succeed
  - **Adding a barrier**
    - All loads and stores are protected by the lock – nothing performed inside can be seen outside:

```
lock: ll   r4, 0(r1)
      bneq r4, lock
      mov  r3, #1
      sc   r3, 0(r1)
      beqz r3, lock
      membar
```

```
unlock: membar
        st #0, 0(r1)
```

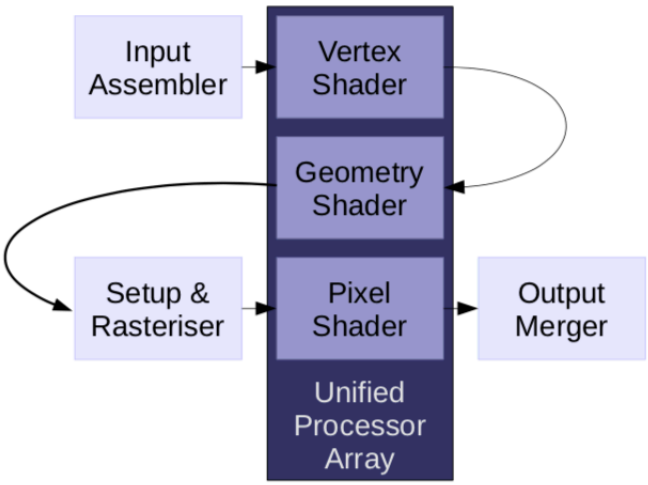
Graphics Processing Units (GPUs)

- **Why GPUs?:** Exploiting SIMT and MIMD
- **Definitions**
  - SIMT is single instruction, multiple thread
    - Each thread has separate state (registers and memory)
  - SIMD is single instruction, multiple data
    - Both take advantage of data-level parallelism
- **Graphics Pipeline**



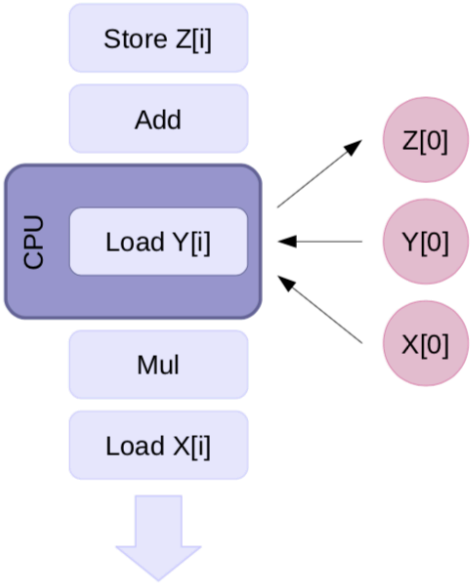
  - Logical graphics pipeline – shaded stages are programmable

- Used to be like that – but now things are more unified:

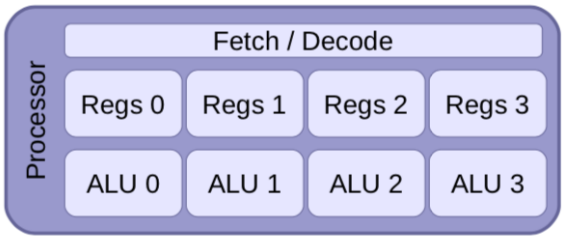


- We map tasks to a unified array of processors and the tasks pass through the array several times
- It still contains fixed function
  - (1) Compression
  - (2) Anti-aliasing
  - (3) Rasterisation
  - (4) Video decoding

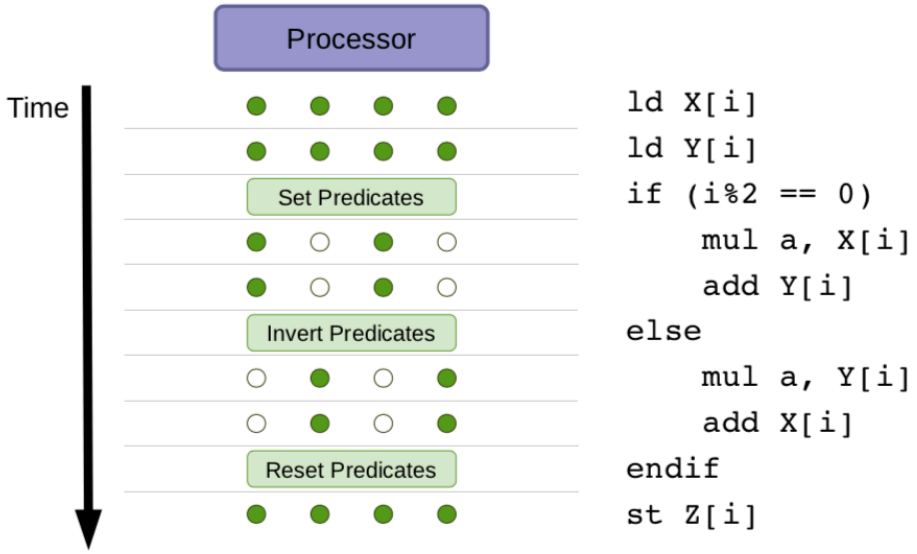
- **Concepts behind GPUs**
  - **Sequential Execution**
    - Instructions execute one-by-one in the CPU – during execution each instruction works on scalar data
    - Only one value is read from or written to memory at a time



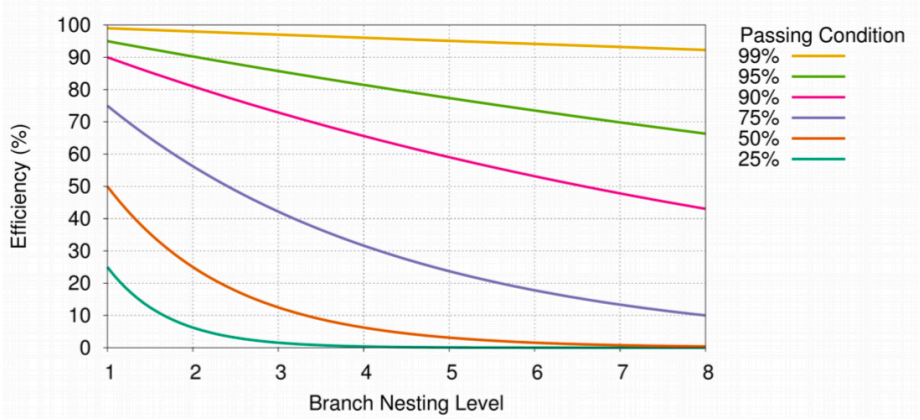
  - Data Level Parallelism – where we are doing the same instruction over again with different data
    - Amortise the cost of instruction fetch and decode
    - **SIMT** – each processor runs many threads (each thread executes the same instructions), each thread has its own registers and memory



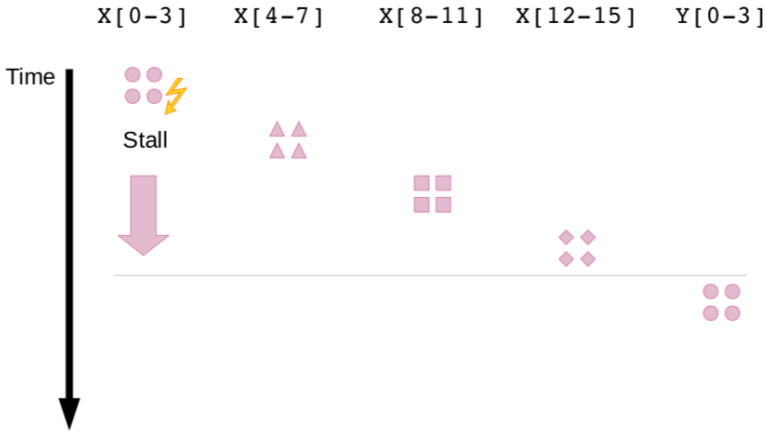
- **Conditional Execution**
  - Add support for conditional execution – **predication or masking** – uses a predicate register to hold conditions
  - Calculated in hardware, but may be special instructions too
  - The threat sits idle if the predicate is false – although this leads to inefficiency



- **Complex Branching**
  - Can also have more complex branching
    - Procedure call and return
    - Nested if-then-else
    - Skip code entirely when no threads execute it
  - Handled by hardware with special instructions
  - Each thread has its own stack
  - Inefficient when some, but not all threads branch – branch divergence



        - **Stalls**
            - All these threads need access to their data – high memory bandwidth requirements and memory accesses can be high latency
            - Stalls are caused by long latency operations
            - Therefore, split the things into warps (smaller groups of things) – therefore can start the next warp when we reach a stall



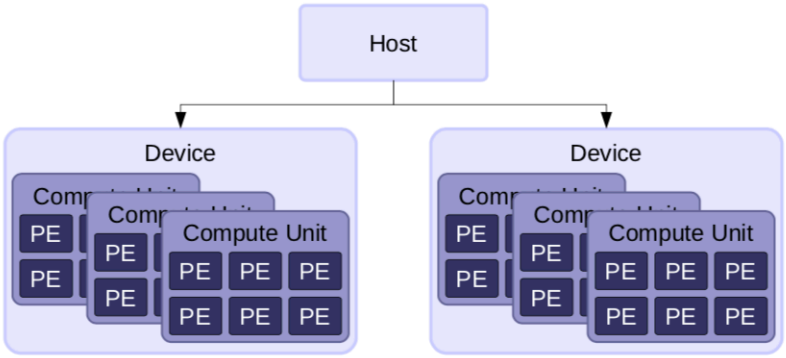
    - Multithreading Support
        - Much more thread state than processing resource
        - Lots of overhead in area from supporting lots of threads
            - Nvidia Fermi architecture supports 48 warps where 1 warp is 32 threads
    - Scheduling
        - Warp scheduler responsible for scheduling warps
        - Selection based on when the operands are ready – for latency hiding (there is no fixed ordering of warps)
        - It chooses ready instructions
        - **Fermi Streaming Multiprocessor**
            - 32 CUDA core per SM
            - 16 Load / Store Units
            - Dual warp scheduler – executed concurrently
- **GPU Memory**
    - Balance the need for isolation vs sharing – dedicated space for each operation but also want to be able to easily share data.
        - Each thread is allocated private local memory and there is shared memory on each multithreaded processor (can be used for communication between threads)
        - There are also constant and texture memories and a **global GPU memory available across the GPU**
    - Also have a cache – though multithreading hides DRAM latency
    - Example: Kepler
        - L1 cache between SM
        - Read-only data cache: high bandwidth and reduces main L1 cache pressure
- **Programming GPUs**

- Use HLLs to make things portable and abstract away the GPU internals
- OpenGL, DirectX, BrookGPU, CUDA, OpenCL
- CUDA
  - Developed by NVIDIA – C-like language
  - Programmer identifies the code to run on the CPU and that for the GPU
  - **CUDA Terminology**
    - **Kernel:** Program or function, designed to be executed in parallel
    - **CUDA Thread:** A single stream of instructions from a computation kernel
    - **Warp:** A group of threads that are executed together
    - **Thread Block:** A set of threads that execute the same kernel and can cooperate
    - **Grid:** Set of thread blocks that execute the same kernel
  - Calling GPU functions
    - __device__ or __global__ or __host__ distinguishes GPU and CPU functions
    - Variables in GPU functions stored in GPU memory – accessible across all GPU multiprocessors
    - Func<<<dimGrid, dimBlock>>>(params) – must call GPU functions with code dimensions – specifies the number of blocks in a grid and the number of threads in a block
    - Example:

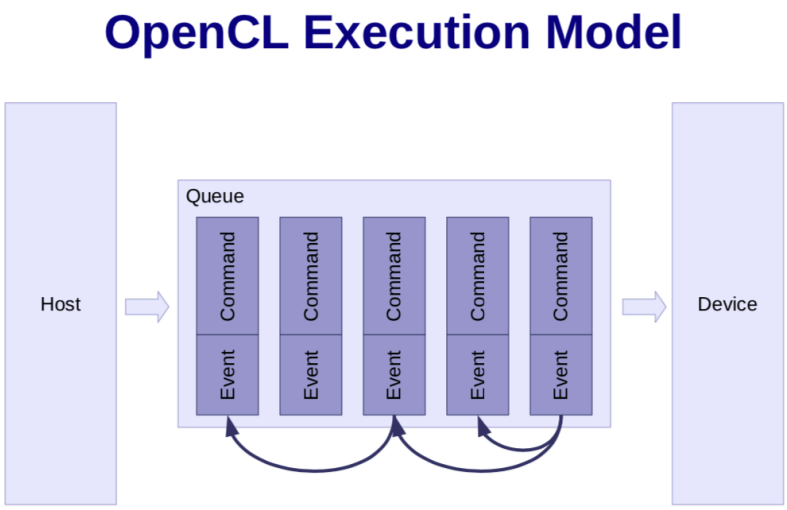
```
void daxpy(int n, double a, double *X, double *Y) {
    for (i=0; i<n; ++i) {
        Y[i] = a*X[i] + Y[i];
    }
}

__host__
Int nBlocks = (n + 255) / 256;
daxpy<<<nBlocks, 256>>>(n, 2.0, X, Y);
```

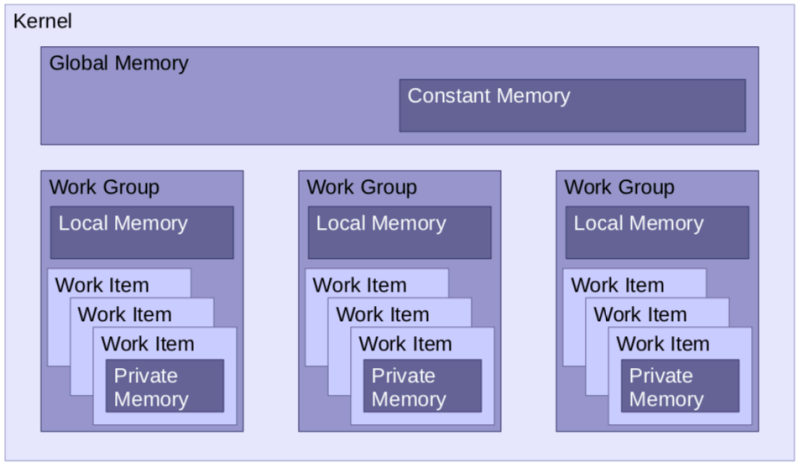
- OpenCL
  - Managed by the Khronos Group
  - Specification defines four models:
    - (1) Platform model – specified one host and multiple devices
      - Devices are then divided into compute units and then divided into processing elements



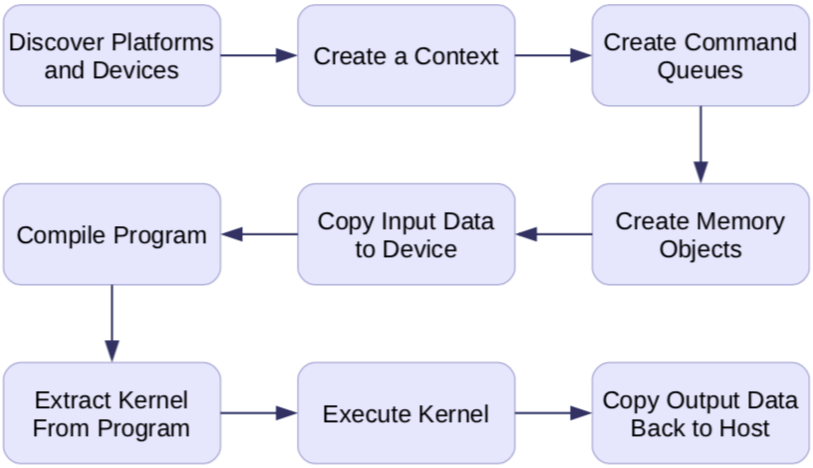
    - Allows a system to have multiple platforms – Intel and AMD
        - This allows portability
    - GPU is a device
- (2) Execution model – defines how the host interacts with devices
    - Describes how execution is actually set up
    - Establish a context
        - Abstract environment for the execution
        - Manage memory objects
        - Manage interaction between host and device
        - Keep track of programs and kernels on each device
    - Command queues (one queue per device) allow the host to communicate with the devices
        - Can be in-order (FIFO) and out-of-order (commands can be rearranged for efficiency)
        - Barriers are used to synchronise queues
    - Event objects specify common dependencies and each command has a wait list (events that this command depends on) and each command has its own event
        - Events contain the state of the command
            - Queued
            - Submitted
            - Ready
            - Running
            - Ended
            - Complete



- (3) Kernel programming model – defines how concurrency is mapped
    - Kernels: What actually run on the device – syntactically like a C function – compiled at runtime
        - Optimised for a specific device
    - Each kernel contains the body of a loop
    - Work-items are created to execute each kernel – each work-item executes one iteration of a loop – corresponds to a thread in CUDA
    - The number of work-items created is expressed as an n-dimensional range (NDRange)
        - 1, 2 or 3 dimensions
        - Index space
    - Work-items within an NDRange are grouped into work-groups – same dimensions and share a memory address space
        - Synchronise on barriers with the group
- (4) Memory model – defines the abstract memory that kernels use
    - Defines what happens to each memory operation
        - What values are read
        - In what order operations occur
    - Memory objects
        - Buffers – equivalent to C arrays
        - Images – abstract storage
        - Pipes – FIFO sequences of data
    - Either host (defined outside OpenCL) or device memory
    - Device memory is accessible to kernels
        - Global memory
        - Constant memory
        - Local memory shared between work-items within a work-group
        - Private memory for each work-item



- OpenCL Overview

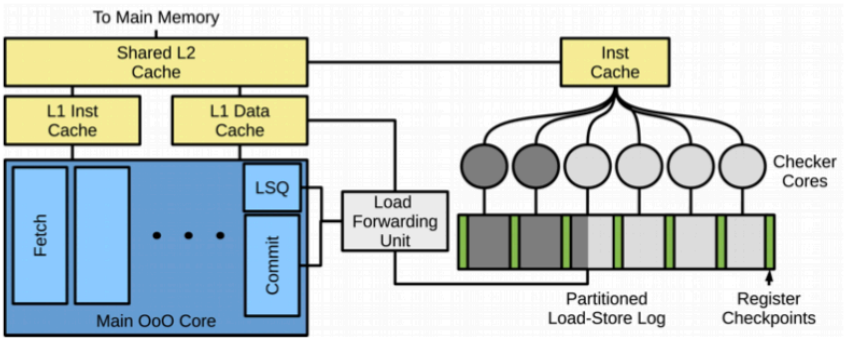


  - CUDA vs OpenCL
    - OpenCL more verbose – platform discovery done at runtime
      - Compilation done at runtime
    - CUDA is only NVIDIA vs OpenCL targets CPUs, GPUs, FPGAs – supporting multiple vendors and being much more flexible

Future Directions

- Challenges
  - (1) Energy efficiency is the new limiter of performance
    - Can get billions of transistors onto a chip but can't power them all on at once… - dark silicon
    - Multicore helps reduce power usage
    - But **Multicore Scaling**
      - Parallel app with 512 cores is only 12x speedup
      - With 50% parallel there is 90% dark silicon at 8nm
      - Therefore, need to fundamentally alter the cores – eliminate overhead
        - **Google's TPUs**
          - ASIC to run neural network interference applications – 15 to 30x faster than CPU or GPU

        - 30x – 80x higher performance / watt
        - Lacks unnecessary features – caches, multithreading and designed for deterministic latency
      - Microsoft said 95% improvement in latency with only 10% more power usage for Bing
    - Limited by Amdahl's Law?
  - **Processing-in-memory**
    - Data movement is otherwise very expensive in terms of energy usage
    - Eg. ReRAM – for massive parallelism
    - Other memory technologies such as Phase-Change Memory (PCM) will have lower power usage (non-volatile) but slower writes
- (2) Reliability
  - Process variation at manufacture
  - **Dual-Core Lockstep**
    - Two separate processors running same code – checking logic to check both give same value
  - **Heterogeneous Error Detection**
    - Better for power and area - only 16% overheard



  - Important to note that reliability is not always necessary
- (3) Performance
  - Computational Sprinting
    - Temporarily exceed power budget and allow some cores to run faster and hotter
- (4) Security
  - Cloud means we may be running on untrusted services
  - Programming languages can reduce faults
  - Lots of recent issues: spectre, CHERI
  - **Trusting Hardware**
    - lowRISC
      - **Open-source** SoC capable of running Linux – cheap and eay to purchase in low quantities
      - Silicon in 45nm and 28nm
      - Using RISC-V ISA
      - Allows some trust in the chip