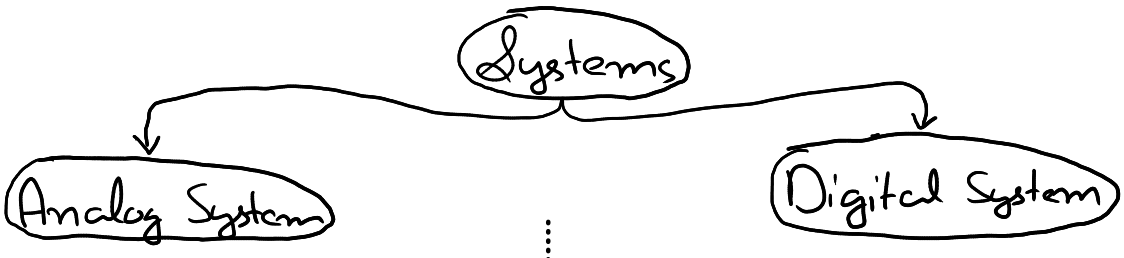
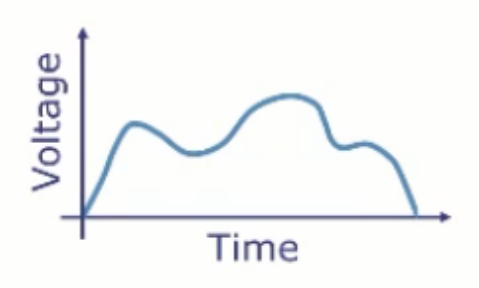
Lecture 1: The digital Abstraction



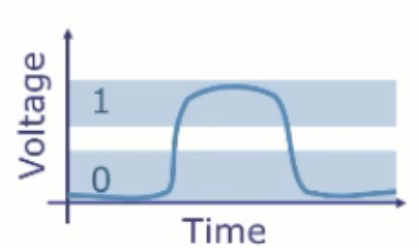
Represent process information using continuous signals (voltage, current, temperature etc...)

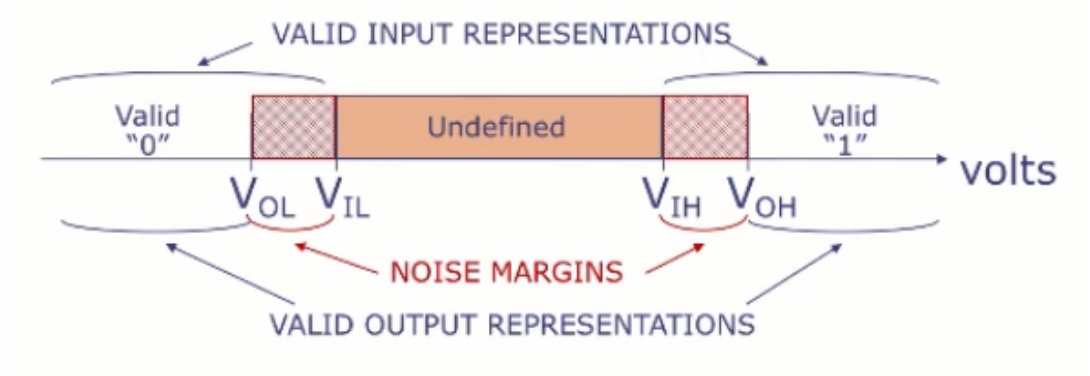


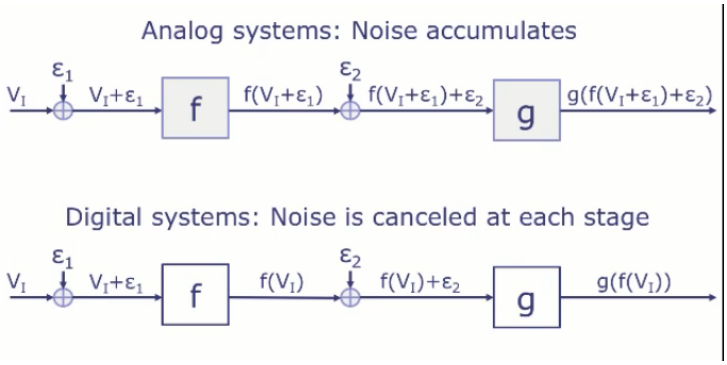
Represent process information using discreate symbols

↳ Tipically binary symbols (bits {0,1})

- Encoded using ranges of values (eg. voltage)
- Digital system tolerate noise

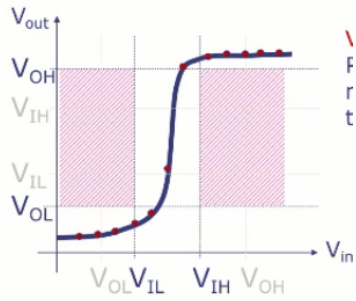






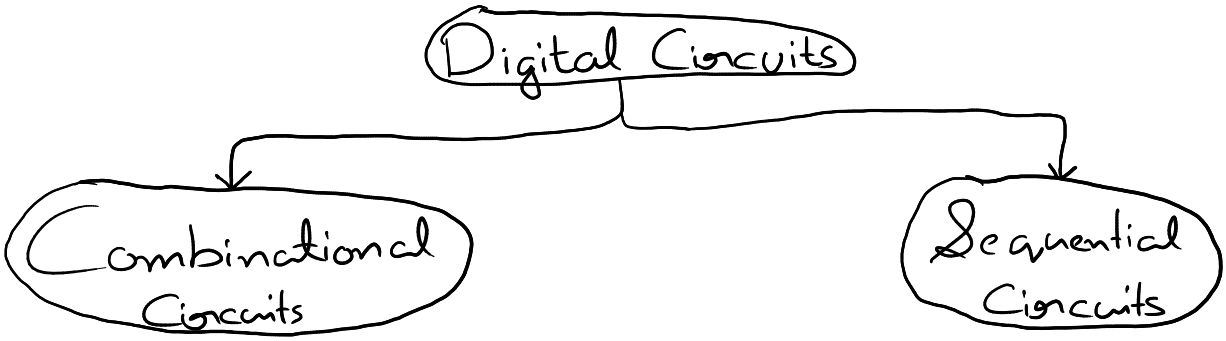
This is why we can build very large digital systems fairly simply.

- Buffer: A simple digital device which copies its input value to its output.



Voltage Transfer Characteristics (VTC)

⇒ Device has gain>1 in the middle and thus Active.



- They donot have memory
- Each output is a function of current input values
- Example: Buffer, Inverter, AND Gate

- Have memory
- Each output depends on current input and memory values

(2)

Lecture 2: Combinational device & Boolean Algebra

⇒ A Combinational device is a circuit element that has:

1. One or more digital input.
2. One or more digital output.
3. A functional specification
   ↳ Details the value of each output for every possible combination of valid input
4. A timing specification consisting of propagation delay (t_{pd})
   ↳ An upper bound on the required time to produce valid, stable output values from an arbitrary set of valid, stable input.

Static Discipline

⇒ A set of inter-connected elements is a combinational device if:

1. Each circuit element is combinational
2. Every input is connected to exactly one output or to a constant (0 or 1)
3. The circuit contains no directed cycles

⇒ Functional specification:

We will use two systematic approaches:

- Truth tables enumerate the output values for all possible combinations of input values
- Boolean expressions are equations containing binary (0/1) variables and three operations: AND (·), OR (+), and NOT (overbar)

Boolean Algebra

- Boolean algebra comprises
  - Two elements, 0 and 1
  - Two binary operators, AND (·) and OR (+)
  - One unary operator, NOT (overbar)

a	b	a·b
0	0	0
0	1	0
1	0	0
1	1	1

a	b	a+b
0	0	0
0	1	1
1	0	1
1	1	1

a	$\bar{a}$
0	1
1	0

- All of Boolean algebra can be derived from the definitions of AND, OR, and NOT

Boolean Algebra Axioms

- Instead of using truth tables to define AND, OR, and NOT, we can derive all of Boolean algebra using a small set of axioms:

identity	a·1=a	a+0=a
null	a·0=0	a+1=1
negation	$\bar{0}$=1	$\bar{1}$=0

- Duality principle: If a Boolean expression is true, then replacing 0 ↔ 1 and AND ↔ OR yields another expression that is true
  - This principle holds for the axioms → Holds for all expressions

Useful Boolean Algebra Properties

- Using the axioms, we can derive several useful properties to manipulate and simplify Boolean expressions:

commutative	a·b = b·a	a+b = b+a
associative	a·(b·c) = (a·b)·c	a+(b+c) = (a+b)+c
distributive	a·(b+c) = a·b+a·c	a+b·c = (a+b)·(a+c)
complements	$a\cdot\bar{a} = 0$	$a+\bar{a} = 1$
absorption	a·(a+b) = a	a+a·b = a
reduction	$a\cdot b + a\cdot\bar{b} = a$	$(a+b)\cdot(a+\bar{b}) = a$
DeMorgan's Law	$\overline{a\cdot b} = \bar{a}+\bar{b}$	$\overline{a+b} = \bar{a}\cdot\bar{b}$

⇒ Given a truth table, it is easy to derive an equivalent Boolean expression:

↳ Write a sum of product terms where each terms cover a single 1 in the truth table.

C	B	A	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

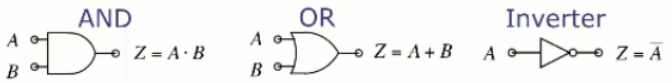
{Example}

$$Y = \bar{C}\bar{B}A + \bar{C}B\bar{A} + CB\bar{A} + CBA$$

⇒ This representation is called the functions normal form.

⇒ A logic diagram represents a Boolean expression as a circuit schematics with logic gates and wires

- Basic logic gates:



⇒ AND, OR, and NOT are universal: They can implement any combinational functions.

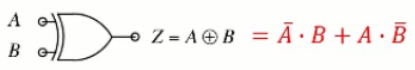
⇒ A minimal sum of products is a sum of products expression that has the smallest number or operators.

⇒ Simple algebric manipulation is sufficient to minimize small expressions (3-4 variables)

⇒ More sophisticated techniques exsts (e.g. K-map).

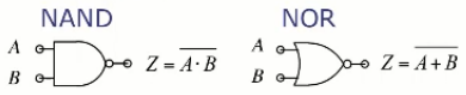
Other Common Gates

- XOR (Exclusive-OR)



A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

- Inverting logic



⇒ NANDs and NORs are universal

⇒ Any logic function can be implemented using only NANDs (or equivalently, NORs)

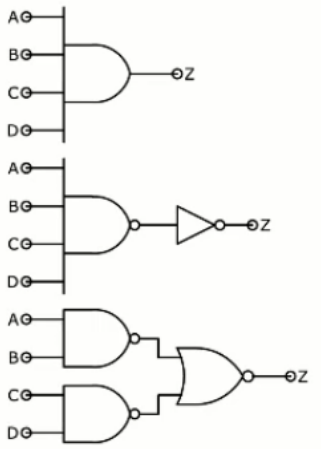
Standard Cell Library

- Library of gates and their physical characteristics
- Example:

Gate	Delay (ps)	Area (μ^2)
Inverter	20	10
Buffer	40	20
AND2	50	25
NAND2	30	15
OR2	55	26
NOR2	35	16
AND4	90	40
NAND4	70	30
OR4	100	42
NOR4	80	32

Observations:
1. In current technology (CMOS), inverting gates are faster and smaller
2. Delay and area grow with number of inputs

Design Tradeoffs: Delay vs Size



AND4:
t_{PD} = 90 ps, size = 40μ^2

NAND4 + INV:
t_{PD} = 90 ps, size = 40μ^2

2*NAND2 + NOR2:
t_{PD} = 1 NAND2 + NOR2 = 65 ps,
size = 2 NAND2 + NOR2 = 46μ^2

⇒ Hardware designers write circuits in a hardware description language, and use a synthesis tool to derive optimized implementations.

↳ In this course we will be using HDL bluespec

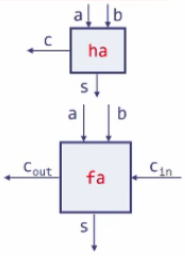
(3)

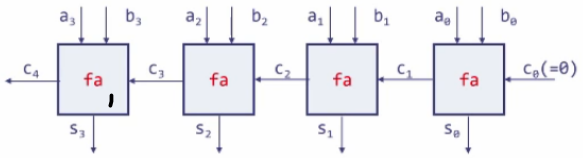
Lecture 3: Describing Combinational Circuit in Bluespec

⇒ Numbers can be written in binary notation and arithmetic can be performed on binary numbers

Combinational Logic for an adder

1. Half adder (HA): adds two 1-bit numbers and produces a sum and a carry bit
2. Full adder (FA): adds two one-bit numbers and a carry, and produces a sum bit and a carry bit
   Can be built using two HAs
3. Cascade FAs to perform binary addition






```
function Bit#(2) ha(Bit#(1) a, Bit#(1) b);
 Bit#(1) s = a ^ b;
 Bit#(1) c = a & b;
   return {c,s};
endfunction
```

- "Bit#(1) a" type declaration says that a is one bit wide

Half adder in Bluespec

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

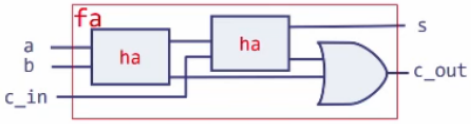
(Truth Table)

S = (~A).B + A.(~B)

C = A.B

```
function ha(a, b);
   s = a ^ b;
   c = a & b;
   return {c,s};
endfunction
```

Full Adder using HAs

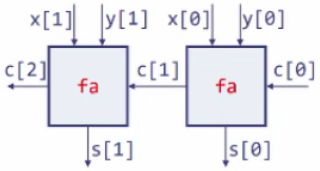



```
function Bit#(2) fa(Bit#(1) a, Bit#(1) b, Bit#(1) c_in);
   Bit#(2) ab  = ha(a, b);
   Bit#(2) abc = ha(ab[0], c_in);
   Bit#(1) c_out   = ab[1] | abc[1];
   return {c_out, abc[0]};
endfunction
```

- ha can be used as a black box as long as we understand its type signature

- Suppose we write t = ha(a,b) then t is a two bit quantity representing s and c values
- We can recover s and c values from t by writing t[0] and t[1], respectively

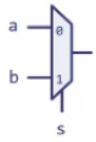
2-bit Ripple-Carry Adder
cascading full adders



Use fa as a black-box

```
function Bit#(3) add2(Bit#(2) x, Bit#(2) y);
   Bit#(2) s = 0;      Bit#(3) c = 0;
   c[0] = 0;
   let cs0 = fa(x[0], y[0], c[0]);
   s[0] = cs0[0];    c[1] = cs0[1];
   let cs1 = fa(x[1], y[1], c[1]);
   s[1] = cs1[0];    c[2] = cs1[1];
return {c[2],s};
endfunction
```

★ Selectors and Multiplexers



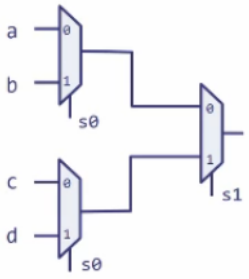
A mux is a simple conditional expression

Bluespec `(s==0)? a : b ;`

(2-way mux)

```
case ({s1,s0})
   0:  a;
   1:  b;
   2:  c;
   3:  d;
endcase
```

(4-way mux)



⇒ n-way mux can be implemented using n-1 two way mux

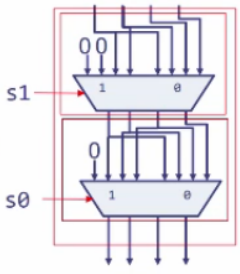
★ Shift Operators

We want to build a shifter which shifts a value x by n where n is between 0 and 31.

- Shift *n* can be broken down into log *n* steps of fixed-length shifts of size 1, 2, 4, ...
  - For example, we can perform Shift 3 (=2+1) by doing shifts of size 2 and 1
  - Shift 5 (=4+1) by doing shifts of sizes 4 and 1
  - Shift 21 (=16+4+1) by doings shifts of sizes 16, 4 and 1

Logical right shift circuit

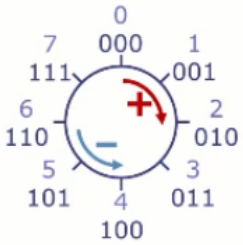
- Define log *n* shifters of sizes 1, 2, 4, ...
- Define log *n* muxes to perform a particular size shift
- Suppose s = {s1,s0} is a two bit number. Shift circuit to shift a number by s can be expressed as two nested conditionals expressions

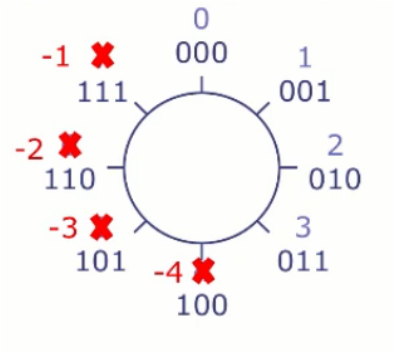


(4) Lecture 4: Binary Arithmetic

Binary Modular Arithmetic

- If we use a fixed number of bits, addition and other operations may produce results outside the range that the output can represent (up to 1 extra bit for addition)
  - This is known as an overflow
- Common approach: Ignore the extra bit





(Encoding negative number)

- This is called 2s compliment encoding.

⇒ To negate a number, we invert all bits and add one.

6: 0110

1001 (Inverting all bits)

-6: 1010 (add one)

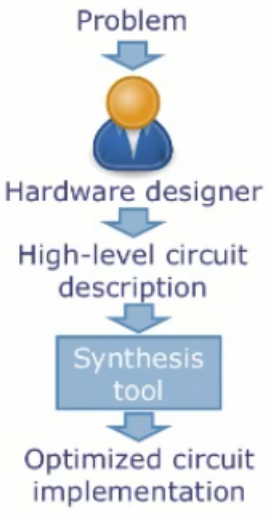
⇒ To compute A-B, we simply use addition and compute A+(-B)

Asymptotic Analysis

- Formally, $g(n)=\Theta(f(n))$ iff there exist $C_2 \geq C_1 > 0$ such that for all but *finitely many* integers $n \geq 0$,

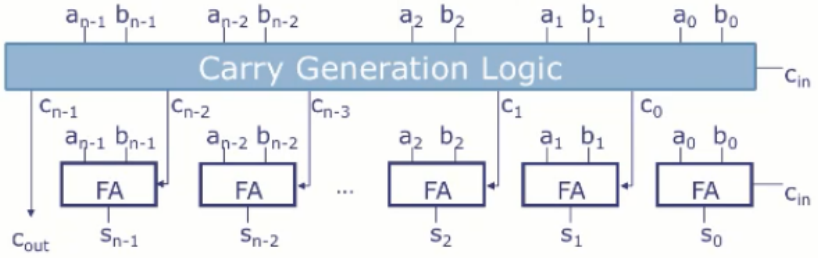
$$\underbrace{C_2 \cdot f(n) \geq g(n) \geq C_1 \cdot f(n)}_{g(n) = O(f(n))}$$

$\Theta(\ldots)$ implies both inequalities;
$O(\ldots)$ implies only the first.



Carry-Lookahead Adders (CLAs)

- CLAs compute all carry bits in $\Theta(\log n)$ delay



- Key idea: Transform chain of carry computations into a tree

This significantly decreases the transmission delay of the circuit.

(5) Lecture 5: Complex Combinational Circuit in Bluespec

⇒ w bit Ripple carry adder

```
function Bit#(TAdd#(w,1)) addN(Bit#(w) x, Bit#(w) y,
                               Bit#(1) c0);
   Bit#(w) s = 0;
   Bit#(TAdd#(w,1)) c;
   c[0] = c0;
   let valw = valueOf(w);
   for (Integer i=0; i<valw; i=i+1) begin
      let cs = fa(x[i], y[i], c[i]);
      c[i+1] = cs[1];
      s[i] = cs[0];
   end
   return {c[valw],s};
endfunction
```

Types (In Bluespec)

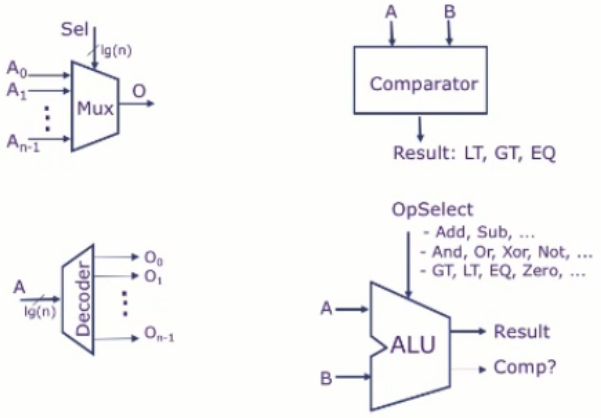
- A type is a *grouping* of values, examples
  - Bit#(16) // 16-bit wide bit-vector (16 is a numeric type)
  - bit [15:0] // synonym for Bit#(16)
  - Bool // 1-bit value representing True and False
  - UInt#(32) // unsigned integers, 32 bits wide
  - Vector#(16,Int#(8)) // Vector of size 16 containing Int#(8)'s

Compiler assigns a bit representation to every typed value that is implementable in hardware

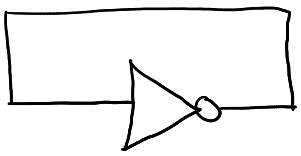
- pack: converts a typed value into bits
- unpack: converts bits into a desired type value

(6) Lecture 6: Sequential Circuits

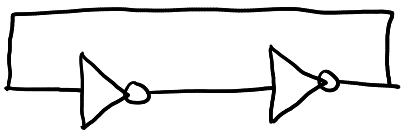
Combinational circuits



★ Simple circuits with feedback

- 

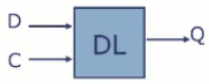
  { • This circuit will oscillate
  • Frequency of oscillation will depend on propagation delay }

- 

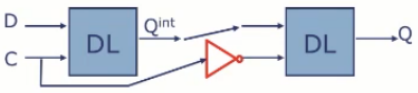
  { This circuit can hold state 0 or 1 }

D Latch: a famous circuit that can hold state

if C=0, the value of D passes to Q
if C=1, the value of Q holds

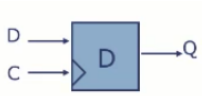


C	D	Q^{t-1}	Q^t
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1



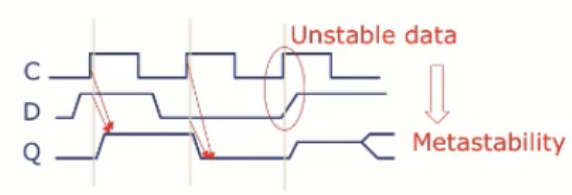
- When C = 0, Q holds its old value, but Q^{int} follows the input D
- When C = 1, Q^{int} holds its old value, but Q follows Qint
- Q doesn't change when C = 0 or C = 1, but it changes its value when C transitions from 0 to 1 (a *rising-edge* of C)

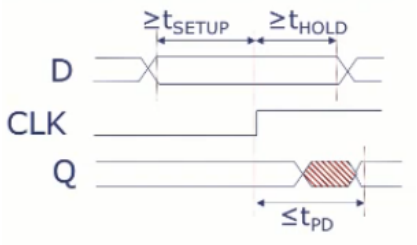
- Called Edge-Triggered D Flip Flop



⇒ Here data are sampled at the rising edge of the clock and must be stable at that time.

↳ If D is changing around the rising edge this leads to whats called Metastability.

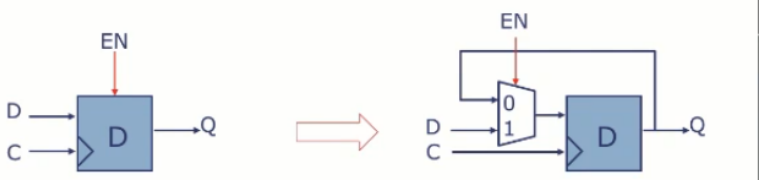


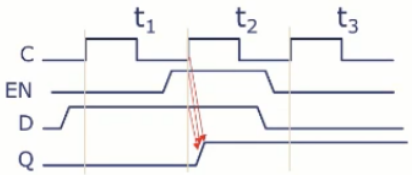


- D should be stable between: $t_{setup} \leq t \leq t_{hold}$
- t_PD is measured from the rising edge of the clock. (propagation delay)

D Flip-flop with Write Enable

The building block of Sequential Circuits



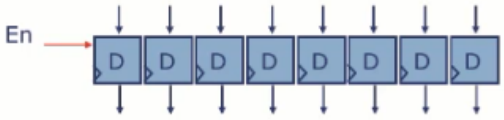


EN	D	Q^t	Q^{t+1}	
0	X	0	0	hold
0	X	1	1	
1	0	X	0	copy input
1	1	X	1	

Data is captured only if EN is on

No need to show the clock explicitly

Registers

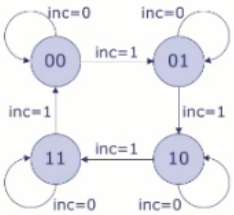


Register: A group of flip-flops with a common clock and enable

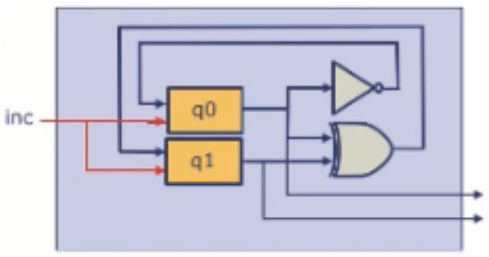
Register file: A group of registers with a common clock, a shared set of input and output ports

Modulo-4 counter

Prev State	NextState	
q1q0	inc = 0	inc = 1
00	00	01
01	01	10
10	10	11
11	11	00



Finite State Machine (FSM) representation



$$q0^{t+1} = \sim inc \cdot q0^{t} + inc \cdot \sim q0^{t}$$
$$= inc \oplus q0^{t}$$
$$q1^{t+1} = \sim inc \cdot q1^{t} + inc \cdot \sim q1^{t} \cdot q0^{t} + inc \cdot q1^{t} \cdot \sim q0^{t}$$
$$= \sim inc \cdot q1^{t} + inc \cdot (q1^{t} \oplus q0^{t})$$

(7)

Lecture 7: Sequential Circuit in BSV

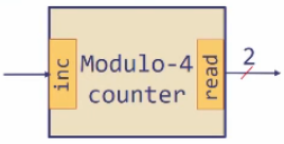
⇒ Complex hardware is viewed as a bunch of modules whose I/O wires are connected in some manner.

⇒ Modular Refinement

↳ Ability to refine a module without understanding the whole design.

⇒ We should never mix implementation with functionality when designing interfaces.

★ Sequential Circuit as a module with interface




```
interface Counter;
    method Action inc;
    method Bit#(2) read;
endinterface
```

- A module has internal state
- The internal state can only be read and manipulated by the interface methods.
- Actions are atomic.
- A module in bluespec is like a class defined in c++

```
module mkCounter(Counter);
   Reg#(Bit#(2)) cnt <- mkReg(0);
   method Action inc;
      cnt <= {cnt[1]^cnt[0],~cnt[0]};
   endmethod
   method Bit#(2) read;
      return cnt;
   endmethod
endmodule
```

★ FIFO

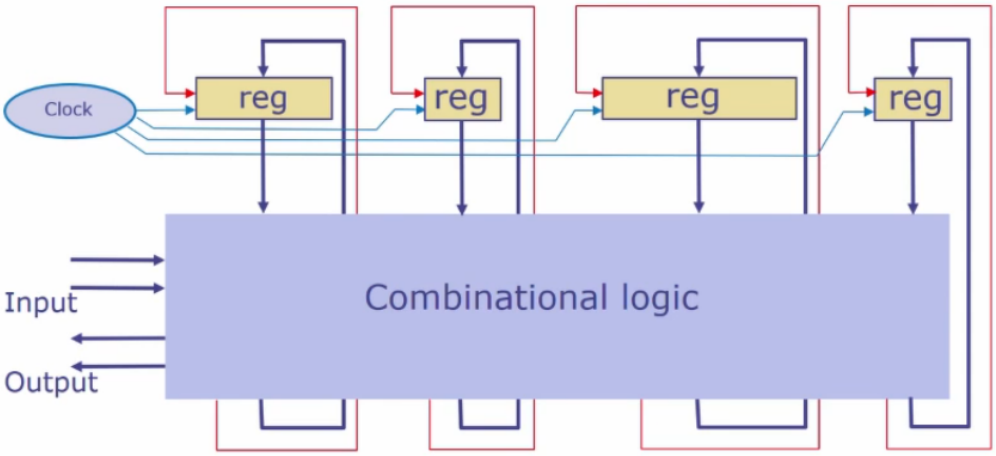
```
interface Fifo#(numeric type size, type t);
  method Bool notFull;
  method Bool notEmpty;
  method Action enq(t x);
  method Action deq;
  method t first;
endinterface
```

8

Hardware Synthesis from Bluespec

⇒ Every module represent a sequential machine in Bluespec.

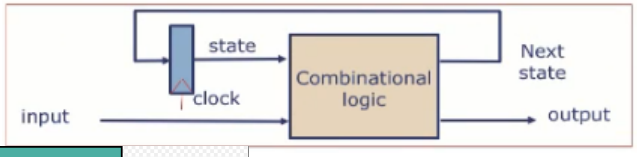
⇒ Rules and Methods only defines combination logic.

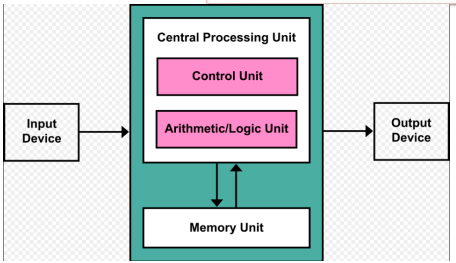


9

Lecture 9: Programmable Machine

⇒ Using Combinational and Sequential Logic, we can build special purpose hardware
: A finite state machine that solves a particular problem.





{Von Neumann architecture of a general purpose computer}

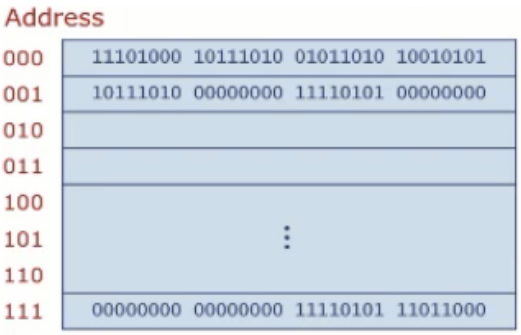
★ Main memory = Random Access Memory (RAM)

⇒ Registers and Register files can only be used to store a small number of data elements.

⇒ To support all other storage needs, we use main memory.

⇒ MainMemory: Array of bits, organized in W words of N bits each.

- → Typically W is a power of 2.
- → Example: W=8, N=32



- → Can read and write to individual words.

★ Storage Conventions: Registers vs Memory

⇒ Typically variables lives in memory.

⇒ Register holds tempory values or values that we need to use repeatedly.

⇒ ALU operations are performed on registers.

⇒ To operate with memory variables

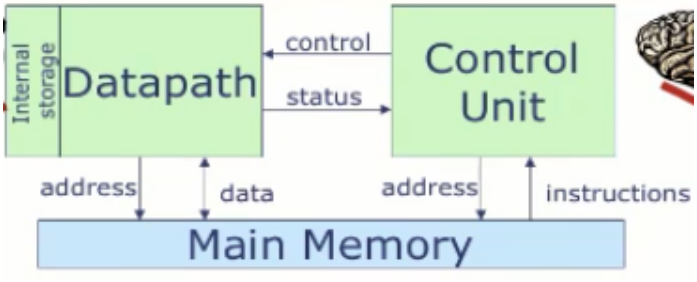
- → Load them into a registor
- → Compute on them
- → Store the results back to memory

★ Von Neumann Computer

⇒ Express program as a sequence of coded instructions.

⇒ Memory holds both data and instructions.

⇒ CPU fetches, interprets, and executes sussessive instructions of the program.



Program Counter (PC)

- → A special purpose register which holds the address of the next instruction.

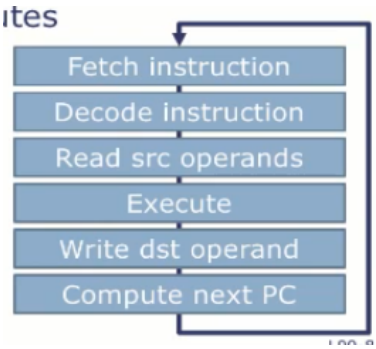
★ Instructions

⇒ Instructions are fundamental unit of work.

⇒ Each instruction specifies:

- → An operation or opcode to be performed.
- → Sorce and destination operands.

⇒ A Von Neumann machine executes instructions sequentially:



⇒ By default, the next PC is {(current PC) + (size of current instruction)} unless the instruction says otherwise.

★ Instruction set Architecture (ISA)

⇒ ISA: Contract between software and hardware

- → Functional defination of operations and storage locations
- → Precise description of how software can invoke and access them.

⇒ ISA species what hardware provides, not how its implemented

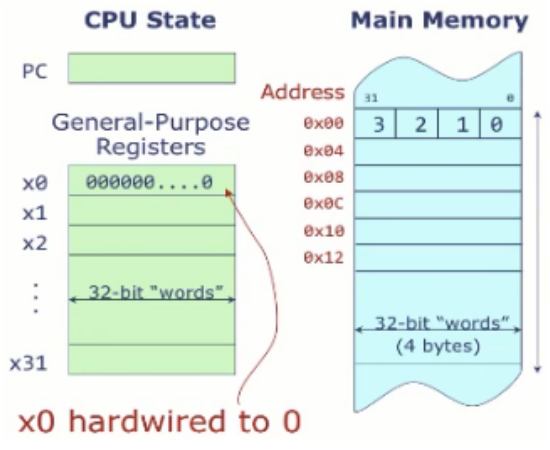
⇒ Enable faster inovation in hardware (no need to change software)

⇒ Downside: Commercially successful ISAs last for decades

- → Today x86 CPUs carry baggage of design decisions from 70s.

★ RISC-V ISA

⇒ We will focus on RV32I Processor, which is the base integer 32 bit variant.



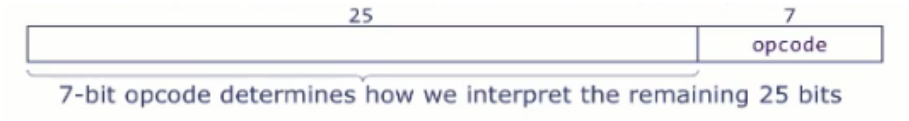
⇒ Each memory word is 32 bit wide, but we use byte memory addresses.

⇒ Since each word contains 4 bytes, address of consecutive words differ by 4.

⇒ Three type of instructions:

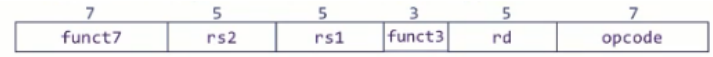
1. Computational: Perform operations on general registers
2. Load and Store: Move data between general registers and main memory
3. Control: Change the program counter

⇒ All instructions have a fixed 32-bit length (4 byte)

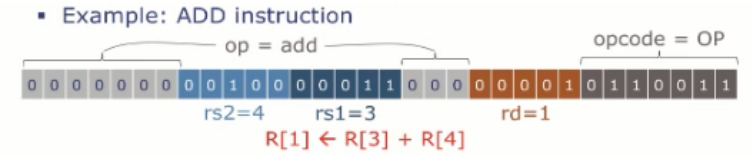


@ Computational Instruction

⇒ Register to register instructions (R-Type)



→ Operation op is specified by (funct7 and funct3)



⇒ We prefer a symbolic representation: add x1, x3, x4

- Similar instructions for other operations:

Arithmetic	Comparisons	Logical	Shifts
ADD, SUB	SLT, SLTU	AND, OR, XOR	SLL, SRL, SRA

⇒ Many programs use small constants frequently.

⇒ Using registors to hold these constants is wastefull.

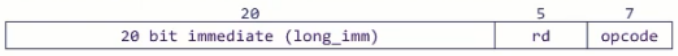
⇒ Solution: Register-immediate instructions (I-type)

→ Example: addi x5, x2, -3

⇒ Immediate operands are encoded into instructions (12 bit instr_imm)

⇒ Sometime we need to use full 32-bit constants

⇒ Solution: Add a format with a 20-bit immediate



- *Example: Write code to load constant 0xCAFE0123 into x3*

```
lui x3, 0xCAFE0     // x3 = 0xCAFE0000
addi x3, x3, 0x123  // x3 = 0xCAFE0123
```

{lui - load upper immediate}

@ Load and Store Instruction

Load Instruction

- Load move data from main memory into a register
- `lw` x2, 4(x3) ---- R[rd] <- Mem[R[rs1] + imm]
  - memory address
- Encode 12 bit immediate offese in instruction

Store Instruction

- Moves data from register into main memory
- Example: `sw x2, 4(x3)`
- Performs Mem[R[rs1] + imm] ← R[rs2]
  - Memory address
- Encode 12 bit immediate offese in instruction

@ Control Instruction

Conditional branch

- First performs comparison to determine if branch is taken or not: R[rs1] *comp* R[rs2]
- If comparison returns True, then branch is taken:
  - Performs pc ← pc + imm
- Else:
  - Performs pc ← pc + 4
- Offset to label is encoded in 12-bit immediate in instruction
- Can change PC only within a +- 4KB range
- Supported comparison operators

			signed		unsigned	
Instruction	**BEQ**	**BNE**	**BLT**	**BGE**	**BLTU**	**BGEU**
comp	==	!=	<	≥	<	≥

JAL: Unconditional Jump and Link

- Example: `jal rd, label`
- Performs R[rd] ← pc + 4; pc ← pc + imm
- Encodes 20 bit immediate in instruction
- Can jump within +- 1MB range of current PC

JALR : Unconditional Jump via register and link

- Example: `jalr rd, 0(rs1)`
- Performs R[rd] ← pc + 4; pc ← (R[rs1] + imm) & ~0x01
  - ~0x01 forces LSB to 0 (half word alignment)
- Encodes 12 bit immediate in instruction
  - imm = signExtend(inst_imm[11:0])
- Can jump to any 32 bit address – supports long jumps

★ Assembly code vs binary

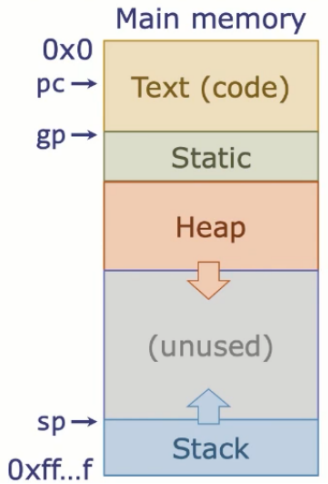
⇒ Its too tedious to write program diractly into binary.

- To simplify writing programs, assemblers provide:
  - Mnemonics for instructions
    - `add x1, x2, x3`
  - Symbols for program locations and data
    - `bneq x1, x2, loop_begin`
    - `lw x1, flag`
  - Pseudoinstructions
    - `mv x1, x2` `// short for addi x1, x2, 0`
    - `j label` `// short for jal x0, label`
    - `beqz x1, dest` `// short for beq x1, x0, dest`

⇒ Asseemblers translate programs into machine code for the processor to execute.

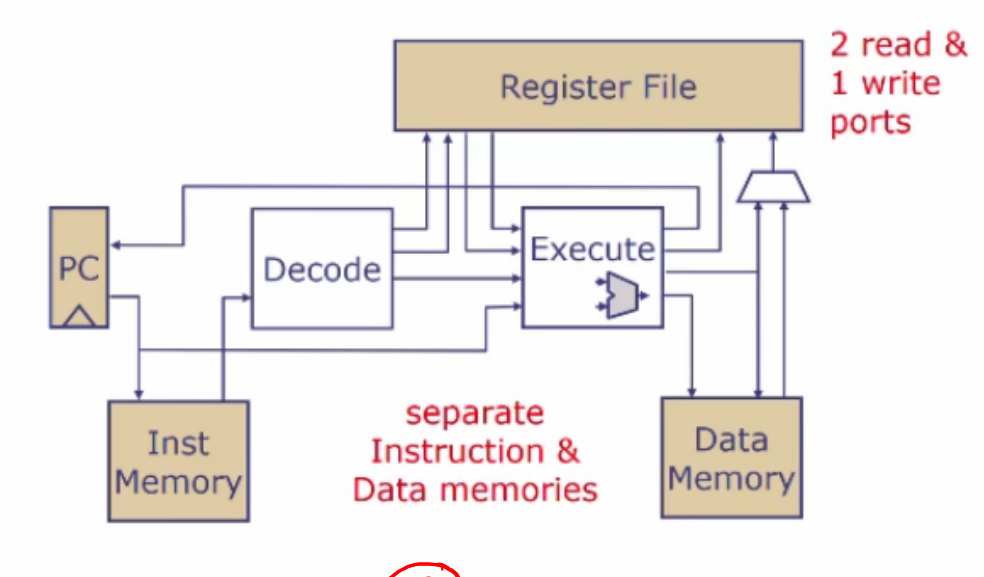
⑩ Lecture 10: Procedures and Stack

RISC-V Memory Layout



- Text, static, and heap regions are placed consecutively, starting from low addresses
- Heap grows towards higher addresses
- Stack starts on highest address, grows towards lower addresses
- sp (stack pointer) points to top of stack
- gp (global pointer) points to start of static region

(11) Implementing RISC-V Interpreter in Hardware



(13) The memory Hierarchy

★ Memory Technology

	Capacity	Latency	Cost/GB
Register	100s of bits	20 ps	$$$$
SRAM	~10 KB-10 MB	1-10 ns	~$1000
DRAM	~10 GB	80 ns	~$10
Flash*	~100 GB	100 us	~$1
Hard disk*	~1 TB	10 ms	~$0.1

* non-volatile (retains contents when powered off)

⇒ We want large, fast and cheap memory.

↳ Solution: Hierarchy of memories

⇒ Approach1: Expose Hierarchy

↳ Programmers are expected to use them cleverly.

⇒ Approach2: Hide Hierarchy

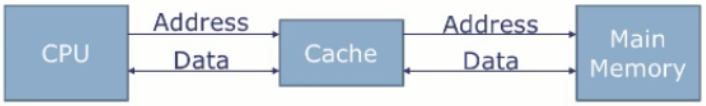
↳ Programming model: Single memory, single address space

⇒ Two predictable properties of memory access:

- Temporal locality
  - If a location has been accessed recently, it is likely to be accesses likely
- Spatial locality
  - If the location has been accessed recently, it is likely that nearby locations will be accessed soon.

Caches

- A small, interim storage component that temporarly retains data from recently accessed location.

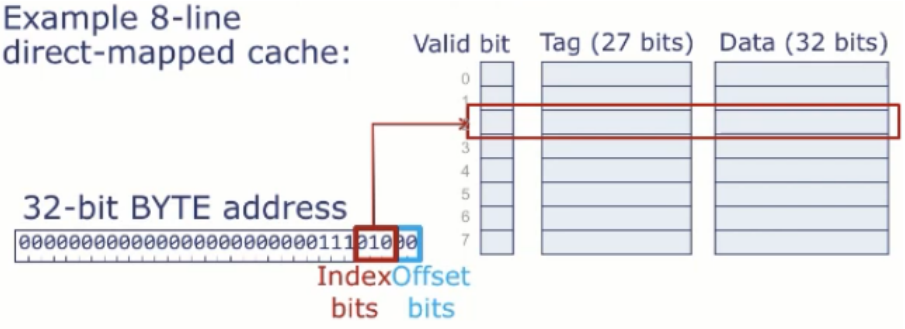


- Processor sends accesses to cache. Two options:
  - Cache hit: Data for this address in cache, returned quickly
  - Cache miss: Data not in cache
    - Fetch data from memory, send it back to processor
    - Retain this data in the cache (replacing some other data)

★ Direct Mapped Caches

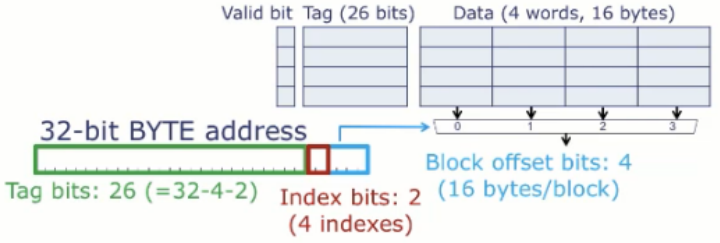
⇒ Each word in memory maps into a single cache line.

- Example 8-line direct-mapped cache:



Block Size

- Takes advantage of spatial locality: stores multiple words per data block.
  - Example: 4-block, 16-word direct-mapped cache

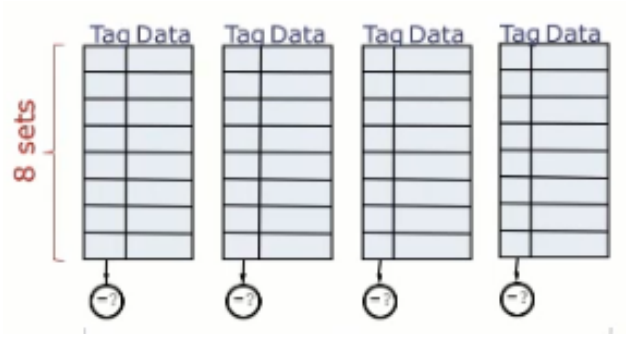


- Block Size tradeoff: Large number of block captures better spatial locality, but due to this number of distinct memory location decreases hence temporal locality is not catpured well.
  - In modern processor block size is typically 16 words.

★ N-Way Set-Associative Caches

⇒ Use multiple way direct-mapped caches in parallel to reduce conflict misses.

⇒ Tags from all ways are checked parallel.



(14)

Caches 2

★ Stores

⇒ On write hit:

- Write-back policy: write only to cache and update the next level memory when the line is evicted.
  - For write-back status needs to be one of:
    - Invalid
    - Clean
    - Dirty

⇒ On write miss:

- Allocate: because of multi word lines we first fetch the line, and then update a word in it.

★ Replacement Policy

⇒ Direct Mapped cache: No Choice

⇒ N-Way Set associative cache:

- N choices for line to replace
- LRU: evict line from least recently used way
  - For a 2-way set associative cache, can implement LRU with 1 bit of state per cache line.

- Blocking cache
  - At most one outstanding miss
  - Cache must wait for memory to respond
  - Cache does not accept processor requests in the meantime
- Non-blocking cache
  - Multiple outstanding misses
  - Cache can continue to process requests while waiting for memory to respond to misses

We will discuss the implementation of blocking caches

15 Introduction to Pipelining

★ Performance measure

⇒ Two metrics of interest when designing a system:

1. Latency: The delay from when an input enters the system until its associated output is produced.

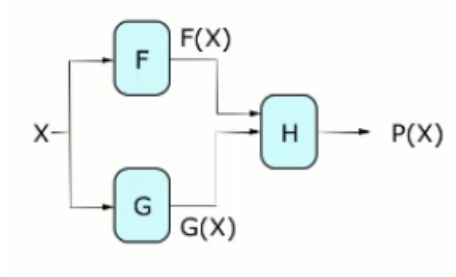
2. Throughput: The rate at which inputs and outputs are processed.

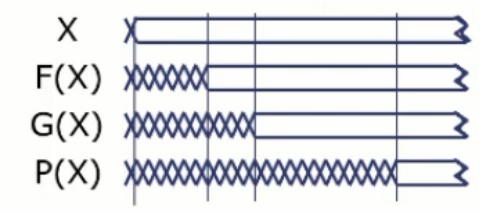
★ Performance of Combinational Logic

⇒ For combinational logic:

latency: t_{PD}

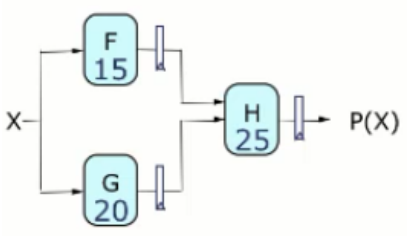
throughput: $\frac{1}{t_{PD}}$





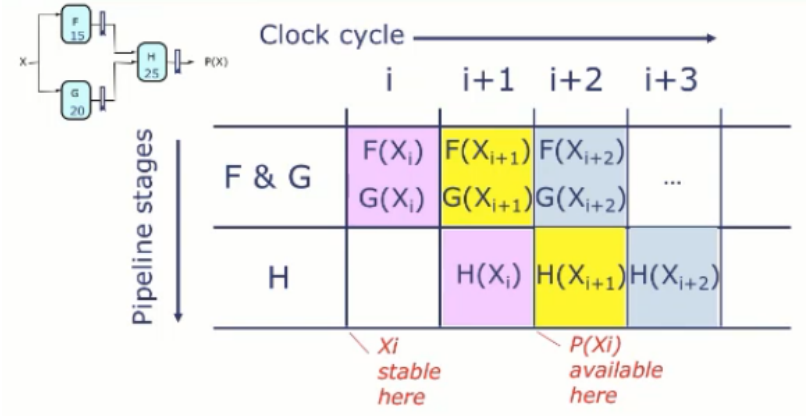
★ Pipelined Circuits

⇒ Use registers to hold H's input stable.



	latency	throughput
unpipelined	45	1/45
2-stage pipeline	50	1/25
	worse	better!

Pipeline Diagrams



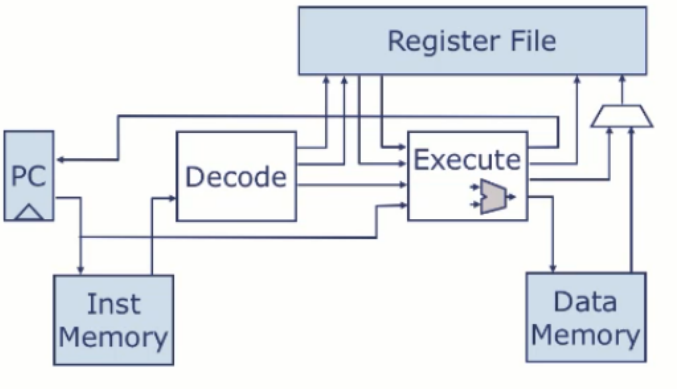
★ Pipeline Conventions

⇒ A well formed k-stage pipeline is acyclic circuit having exactly k registers on every path from an input to an output.

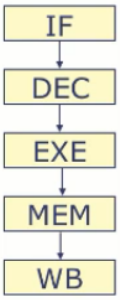
⇒ The clock must have a period t_clk sifficient to cover the longent register to register propagation delay plus setup time.

K-pipeline latency L = K * t_{CLK}
K-pipeline throughput T = 1 / t_{CLK}

★ Pipelined Processor



- We'll study the classic 5-stage pipeline:



Instruction Fetch stage: Maintains PC, fetches instruction and passes it to
Decode & Read Registers stage: Decodes instruction and reads source operands from register file, passes them to
Execute stage: Performs indicated operation in ALU, passes result to
Memory stage: If it's a load, use input as the address, pass read data (or ALU result if not a load) to
Write-Back stage: writes result back into register file.

$t_{CLK} = \max\{t_{IF}, t_{DEC}, t_{EXE}, t_{MEM}, t_{WB}\}$

Pipeline Hazards

- Pipelining tries to overlap the execution of multiple instructions, but an instruction may depend on something produced by an earlier instruction
  - A data value → Data hazard
  - The program counter → Control hazard (branches, jumps, exceptions)

(16)

Processor Pipelining: Control & Data Hazards

Reminder: Data Hazards

- Consider this instruction sequence:

```
addi x11, x10, 2
xor x13, x11, x12
sub x17, x15, x16
xori x19, x18, 0xF
```

	1	2	3	4	5	6
IF	addi	xor	sub	xori		
DEC		addi	xor	sub	xori	
EXE			addi	xor	sub	xori
MEM				addi	xor	sub
WB					addi	xor

- xor reads x11 on cycle 3, but addi does not update it until end of cycle 5 → x11 is stale!

Resolving Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages
- Strategy 2: Bypass (aka Forward). Route data to the earlier pipeline stage as soon as it is calculated
- Strategy 3: Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value

Strategy 1: for data hazards

Stall

	1	2	3	4	5	6	7	8
IF	addi	xor	sub	*sub*	*sub*	*sub*	xori	
DEC		addi	xor	*xor*	*xor*	*xor*	sub	xori
EXE			addi	**NOP**	**NOP**	**NOP**	xor	sub
MEM				addi	**NOP**	**NOP**	**NOP**	xor
WB					addi	**NOP**	**NOP**	**NOP**

Resolving Control Hazards with Speculation

- *What's a good guess for nextPC?* PC+4
- Assume bne is taken in example

```
loop: addi x13, x11, -1
      sub x14, x15, x16
      bne x13, x0, loop
      and x16, x17, x18
      xor x19, x20, x21
      …
```

	1	2	3	4	5	6	7	8	9
IF	addi	sub	bne	and	xor	addi	sub	bne	and
DEC		addi	sub	bne	and	**NOP**	addi	sub	bne
EXE			addi	sub	bne	**NOP**	**NOP**	addi	sub
MEM				addi	sub	bne	**NOP**	**NOP**	addi
WB					addi	sub	bne	**NOP**	**NOP**

(19)

Operating System

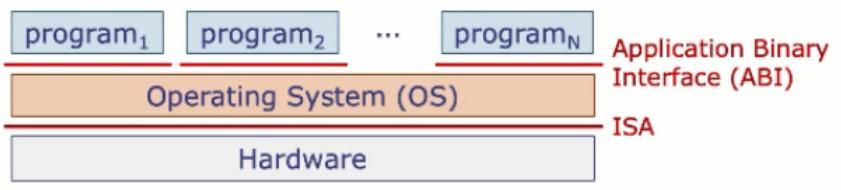
★ Single-User machine

⇒ Hardware executes a single program.

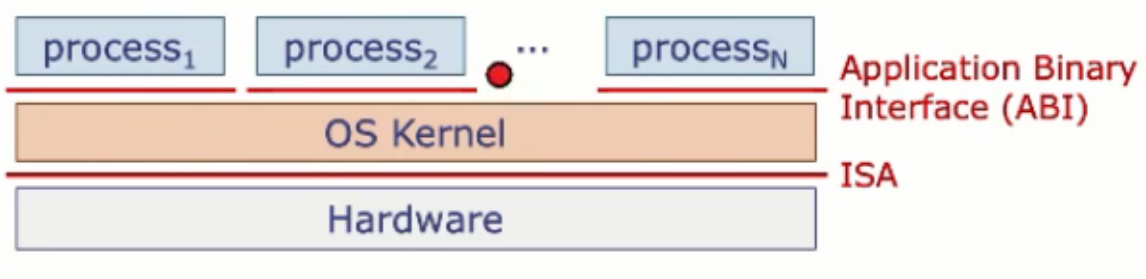
⇒ This hardware has direct and complete access to all hardware resources in the machine.

⇒ The instruction set architeture (ISA) is the interface between software and hardware.

★ Operating System



★ Nomenclature: Process vs Program



Program
↳ Collection of instructions.

Process
↳ An instance of the program that is being executed.

OS Kernel
↳ Process with special privileges.

★ Goles of Operating System

1. Protection and privacy: Process can not access each other's data.
2. Abstraction: OS hides details of underlying hardware.
3. Resource management: OS controls how processes share hardware (i.e. CPU, memory, disk etc)

★ Operating System: Big picture

⇒ The OS Kernel provides a private address space to each process.

⇒ The OS Kernel schedules processes into the CPU.

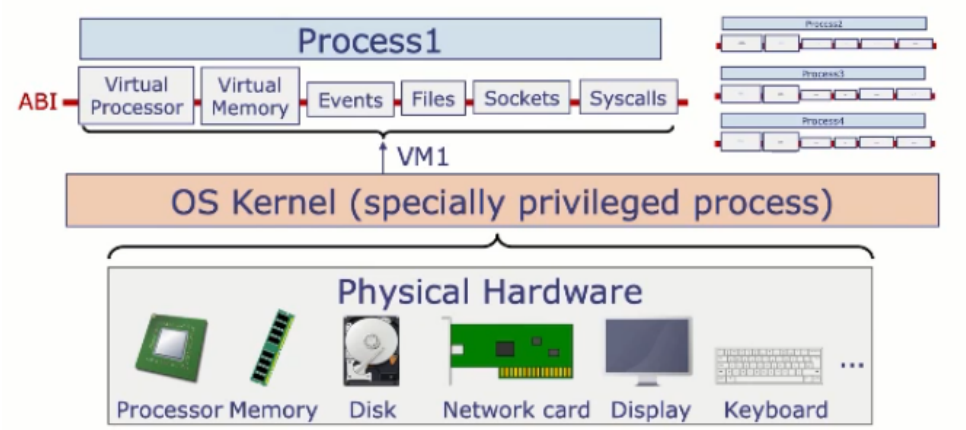
⇒ The OS Kernel lets processes invoke system service via system calls.

★ Virtual machine

⇒ The OS gives virtual machine to each process.

↳ Each process belives it runs on its own machine.

↳ But this machine does not exist in physical hardware.



⇒ Virtual machine can be implemented entirly in software, but at a performance cost.

⇒ We want to support operating system with minimal overhead.

↳ So we need hardware support for virtual machines.

★ ISA Extensions to Support OS

⇒ Two modes of execution: user, supervisor

↳ OS Kernel runs in supervisor mode

↳ All other process runs in user mode

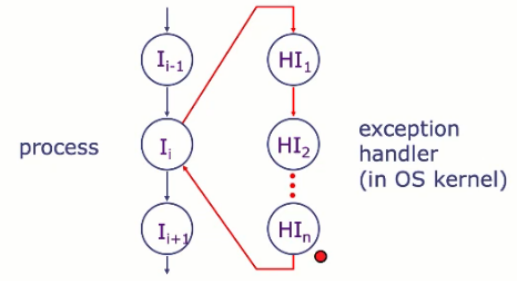
⇒ Privileged instruction and registers that are only available in supervisor mode

⇒ Interrupts and exceptions to safely transition from user to supervisor mode.

⇒ Virtual memory to provide private address space and abstract the storage resources of the machine.

★ Exceptions and Interrupts

⇒ Events that needs to be processed by OS Kernel.



⇒ The term Exceptions and Interrupts are often used interchangeably, with minor distinction.

- Exceptions usually refer to synchronous events, generated by the process itself (e.g., illegal instruction, divide-by-0, illegal memory address, system call)
- Interrupts usually refer to asynchronous events, generated by I/O devices (e.g., timer expired, keystroke, packet received, disk transfer complete)
- When an exception happens, the processor:
  - Stops the current process at instruction I_i, completing all the instructions up to I_{i-1} *(precise exceptions)*
  - Saves the PC of instruction I_i and the reason for the exception in special (privileged) registers
  - Enables supervisor mode, disables interrupts, and transfers control to a pre-specified exception handler PC
- After the OS kernel handles the exception, it returns control to the process at instruction I_i
  - Exception is transparent to the process!
- If the exception is due to an illegal operation by the program that cannot be fixed (e.g., an illegal memory access), the OS aborts the process

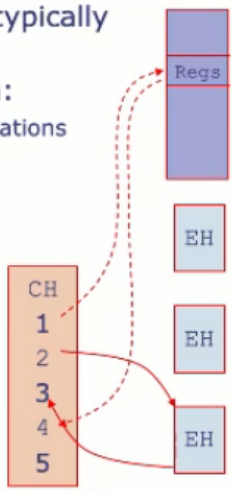
★ CPU Scheduling

⇒ Key enabling technology: Timer interrupts

↳ Kernel sets timer, which raises interrupts after a specified time.

Typical Exception Handler Structure

- A small common handler (CH) written in assembly + many exception handlers (EHs), one for each cause (typically written in normal C code)
- Common handler is invoked on every exception:
  1. Saves registers x1-x31, mepc into known memory locations
  2. Passes mcause, process state to the right EH to handle the specific exception/interrupt
  3. EH returns which process should run next (could be the same or a different one)
  4. CH loads x1-x31, mepc from memory for the right process
  5. CH executes mret, which sets pc to mepc, disables supervisor mode, and re-enables interrupts



20 Virtual Memory

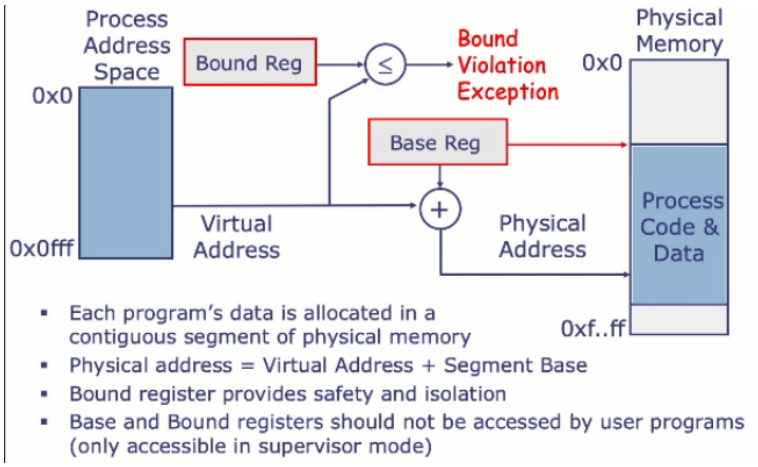
⇒ The price of VM is address translation on each memory reference.

★ Names of Memory Locations

- Machine language address
  - As specified in machine code
- Virtual address
  - ISA specifies translation of machine code address into virtual address of program variable (sometimes called *effective* address)
- Physical address
  - Operating system specifies mapping of virtual address into name for a physical memory location

⇒ Techniques for Address mapping:
- Segmentation
- Paging

★ Segmentation

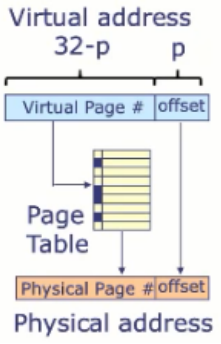


⇒ Problem: Memory Fragmentation

★ Paged memory system

⇒ Divide physical memory in fixed size blocks called pages.

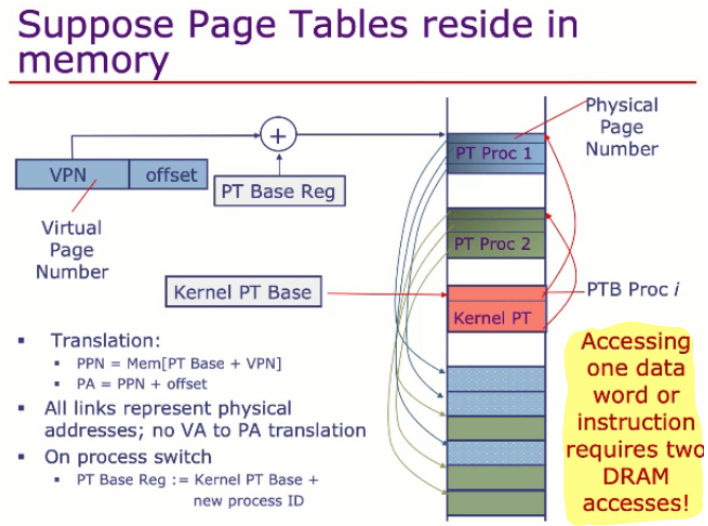
↳ Typical page size: 4KB



⇒ Interpret each virtual address as a pair:
↳ <virtual page number, offset>

⇒ Use page table to translate from virtual to physical page nember.

⇒ Advantage:
1. Paging avoids fragmentation issues
2. Paging allows programs that use more virtual memory then the machines physical memory (demand paging)



★ Demand Paging

⇒ All the pages of the process may not fit in main memory.

↳ Therefore, DRAM is backed up by swap space on disk.

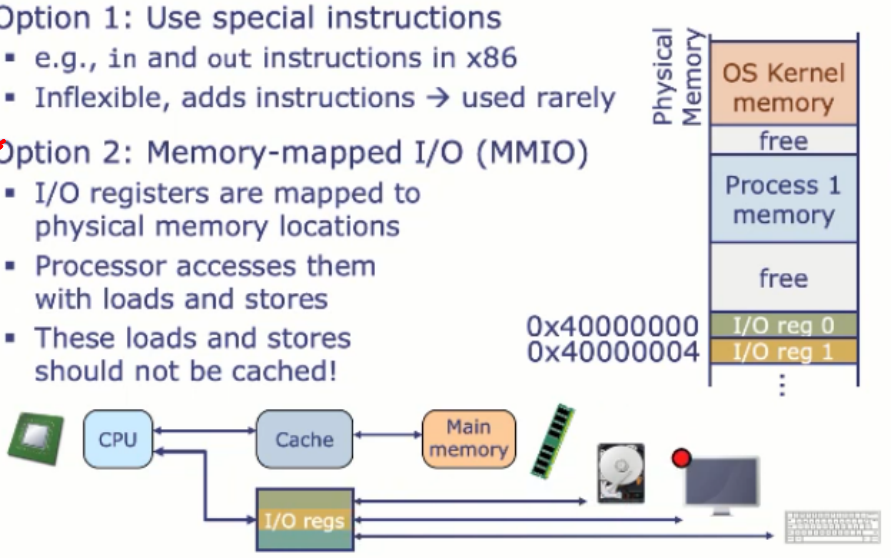
21

Operating System: I/O and System Calls

★ Communicating with I/O

⇒ The system has shared I/O registers that both the processor and the I/O devices can read and write.

- Option 1: Use special instructions
  - e.g., `in` and `out` instructions in x86
  - Inflexible, adds instructions → used rarely
- Option 2: Memory-mapped I/O (MMIO)
  - I/O registers are mapped to physical memory locations
  - Processor accesses them with loads and stores
  - These loads and stores should not be cached!



Coordinating I/O Transfers

- Option 1: Polling (synchronous)
  - Processor periodically reads the register associated with a specific I/O device
- Option 2: Interrupts (asynchronous)
  - Processor initiates a request, then moves to other work
  - When the request is serviced, the I/O device interrupts the processor