DDCA Cheatsheet

MSB → 0000 0000 ← LSB

Nibble

Byte

Binary Numbers

$2^0 = 1$ $2^3 = 8$ $2^6 = 64$ $2^9 = 512$

$2^1 = 2$ $2^4 = 16$ $2^7 = 128$ $2^{10} = 1024$ Kilo

$2^2 = 4$ $2^5 = 32$ $2^8 = 256$ $2^{20} = 1'048'576$ Mega

Sign-Magnitude: First Bit as sign-bit

2's complement: $5 \Rightarrow 0101 \xRightarrow{\text{invert.}} 1010 \xRightarrow{+1} 1011 = -5$

Boolean Algebra

$B\cdot 1 = B$ $B\cdot 0 = 0$ $B\cdot B = B$ $\overline{\overline{B}} = B$ $B\cdot\overline{B} = 0$

$B+0 = B$ $B+1 = 1$ $B+B = B$ $B+\overline{B} = 1$

T6 $B\cdot C = C\cdot D$

T7 $(B\cdot C)\cdot D = B\cdot(C\cdot D)$

T8 $(B\cdot C)+(B\cdot D) = B\cdot(C+D)$

T9 $B\cdot(B+C) = B$

T10 $(B\cdot C)+(B\cdot\overline{C}) = B$

T11 $(B\cdot C)+(\overline{B}\cdot D)+(C\cdot D) = B\cdot C+\overline{B}D$

T12 $\overline{B_0\cdot D_1\cdot B_2 ..} = (\overline{B_0}+\overline{B_1}+\overline{B_2}+..)$

T6' $B+C = C+B$ Commutativity

T7' $(B+C)+D = B+(C+D)$ Associativity

T8' $(B+C)\cdot(B+D) = B+(C\cdot D)$ Distributivity

T9' $B+(B\cdot C) = B$ Covering

T10' $(B+C)\cdot(\overline{B}+D)\cdot(C+D) = (B+C)\cdot(\overline{B}+D)$ Consensus

T11' $\overline{B_0+B_1+..} = \overline{B_0}\cdot\overline{B_1}\cdot ..$ De Morgan's Theorem

Product of Sum:

A	B	X	maxterm
0	0	1	
1	0	0	$\overline{A}+B$
0	1	1	
1	1	0	$\overline{A}+\overline{B}$

$X = (\overline{A}+B)\cdot(\overline{A}+\overline{B})$

A	B	X	minterm
0	0	0	
1	0	1	$A\cdot\overline{B}$
0	1	0	
1	1	1	$A\cdot B$

$X = (A\cdot\overline{B})+(A\cdot B)$

Completeness

	Nand	Nor
Not	$\overline{AA}$	$\overline{A+B}$
And	$\overline{\overline{AB}\;\overline{AB}}$	$\overline{\overline{A+A}+\overline{B+B}}$
Nand	$\overline{AB}$	$\overline{\overline{\overline{A+A}+\overline{B+B}}+\overline{\overline{A+A}+\overline{B+B}}}$
Or	$\overline{\overline{AA}\;\overline{BB}}$	$\overline{\overline{A+B}+\overline{A+B}}$
Nor	$\overline{\overline{\overline{AA}\;\overline{BB}}\;\overline{\overline{AA}\;\overline{BB}}}$	$\overline{A+B}$
XOR	$\overline{\overline{A\,\overline{AB}}\;\overline{B\,\overline{AB}}}$	$\overline{\overline{\overline{A+A}+\overline{B+B}}+\overline{A+B}}$
Xnor	$\overline{\overline{\overline{\overline{A\,\overline{AB}}\;\overline{B\,\overline{AB}}}}\;\overline{\overline{A\,\overline{AB}}\;\overline{A\,\overline{AB}}}}$	$\overline{\overline{A+\overline{A+B}}+\overline{B+\overline{A+B}}}$

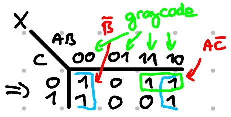
Karnaugh Maps:

1. fewest possible 2. size $2^m x 2^n$ 3. only 1

4. X may be used as 1's 5. as big as possible

A	B	C	X
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

⇒



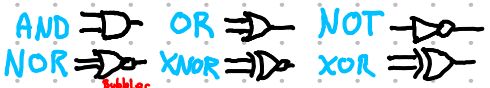
$A\overline{C} \Rightarrow X = \overline{B}+A\overline{C}$

Big vs. Little Endian:

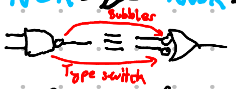
To represent: 0xcafe2b3a

	0	1	2	3
Big Endian:	ca	fe	2b	3a
Little Endian:	3a	2b	fe	ca

Logic Gates

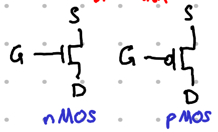


Bubble-Pushing: (De Morgan)



Logic Gate ≡ pMos Network / nMos Network

Transistors



nMOS: conducts when G is high

pMOS: conducts when G is low

Verilog Modules

D Flip-Flop

```
always @(posedge clk) begin
  if (!rst) q<=0;
  else q<=d;
end
```

↳ Synchronous reset: rst not in sensitivity
↳ asynchronous reset:
  · active high: rst
  · active low: !rst

MUX

```
assign y = s[1]?(s[0]?x:y)
               :(s[0]?w:z);
```

↳ could be in always block with

```
always @(*) begin
  case(var)
    2'b01: out= val;
    2'b00: out= val;
    default: out= val;
  endcase
end
```

same for Decoder

D-Latch

```
always @(clk,d) begin
  if (clk) q<=d;
end
```

↳ gets triggered when d changes

Full Adder

```
assign sum = a^b^c_in;
assign cout =(a&b)|(a&cin)|(b&c_in)
```

Correct Code

wires: left of assign w= / no assignment in always-block / connect in/out of modules

regs: can't be connected to output port of module / cannot be used in input port declaration / left of =/<=

I/O: check if names match and if all ports are assigned

General: not multiple assignments to same signal / no recursion / names cannot start with numbers

Verilog Operators

! : logic negation

~ : bitwise negation

& : red. and

| : red. or

~& : red. NAND

~| : red NOR

^ : red XOR

~^ : red XNOR

Combinational vs. Sequential

Comb:
- all left-hand signals are assigned in every possible way
- all inputs are in sensitivity list
- all outputs are assigned
- no memory
- no cyclic paths
- combines inputs to get output

Seq:
- has memory
- depends on prior inputs
- not all outputs are assigned in all cases

Finite State Machines

State changes depending on prev. state + inputs

Moore:
- Output depends on current state.
- more number of states
- Synchronous output & state generation
- output placed on states

Mealy:
- Output depends on current state and input
- less states
- output placed on transitions

Designing a FSM

1. Identify inputs & outputs
2. Sketch State Transition diagram
3. Write state transition & output table
4. write boolean equations for next state

new state, bit 0 depends on old state

$NS[0] = IN_1 \cdot CS[1] + \overline{IN_2}$

inputs

output, bit 0 depends on current state

$O[0] = CS[1] + CS[4]$

Area of FSM

- #FF = #bits for state x 2
- #logic Gates = count nextstate/output logic

State Encodings

Binary Encoding
(00, 01, 10, 11)
$\log_2$(#states) bits needed
reduces #FF to hold states

One-Hot Encoding
(001, 010, 100)
#states bits needed
reduces next state logic

Output Encoding
(100, 110, 111)
reduces output logic

Correctness of FSM

- reset line
- not multiple transitions for same input
- no missing transitions
- no unmarked transitions
- initial state
- no mix of Moore/Mealy

MIPS

R-Type: Register Type, two source/one destination register

I-Type: Immediate Type, one source/one destination register + imm. value

J-Type: Jump Type, operand + address (+Branch)

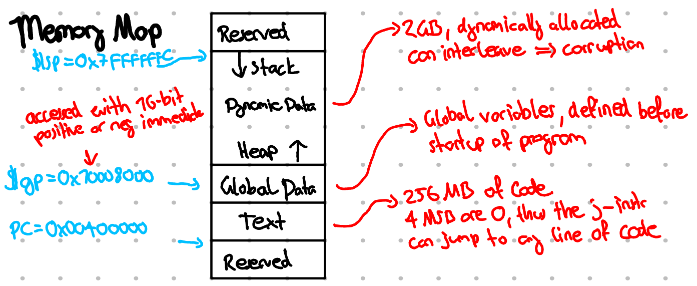
Preserved

saved-register	\$s0 – \$s7
return address	\$ra
stack pointer	\$sp
stack above pointer	

Non-Preserved

temporary registers	\$t0 – \$t9
argument register	\$a0 – \$a3
return value register	\$v0 – \$v1
stack below the stack pointer	

Memory Map



accessed with 16-bit positive or neg. immediate

2GB, dynamically allocated can interleave ⇒ corruption

Global variables, defined before startup of program

256 MB of Code
4 MSB are 0, thus the J-instr can jump to any line of code

ISA

Interface between SW and HW
"what programmer sees"

- Instructions: opcodes, addressing modes, data types, instruction type and format, registers condition codes
- Memory: address space, alignment, addressability, virtual memory management
- Call, interrupt and exception handling
- I/O: memory mapped vs. instructions
- Power & Thermal management
- Multiprocessing/Multithreading support
- Access Control, priority and priviledge
- Memory location of exception vectors
- Function of each bit in a programmable branch predictor register
- Order of loads and stores in multi-core CPU
- Program counter width
- Hardware FP-exeption support
- Vector instruction support
- CPU endianness
- Virtual Page size

µ-Arch

Specifies underlying implementation that actually executes instructions

- Pipelining
- In order vs. Out-of-Order execution
- Memory address scheduling policy
- Speculative execution
- Superscalar processing
- Clock gating
- Caching: levels, size, associativity replacement policies
- Error correction
- Physical Structure
- Instruction latency
- Physical memory page size
- Instruction issue width
- reservation stage capacity
- # pipeline stages
- latency of branch miss prediction
- fetch width of superscalar CPUs
- # non-programmable CPU registers
- register file has one input and two out ports
- number of read ports in physical register file

Performance Evaluation

CPI: cycles per Instruction
IPC: Instructions per cycle
MHz: frequency, 10^6 cycles/s

MIPS: million instructions/sec = MHz/CPI
Time: $\#instr \cdot CPI \cdot \frac{1}{MHz}$
Speedup: oldTime / newTime

higher MHz $\not\Rightarrow$ higher MIPS, IPC could be lower
higher MIPS $\not\Rightarrow$ less time, could need more instructions

Single Cycle Machines

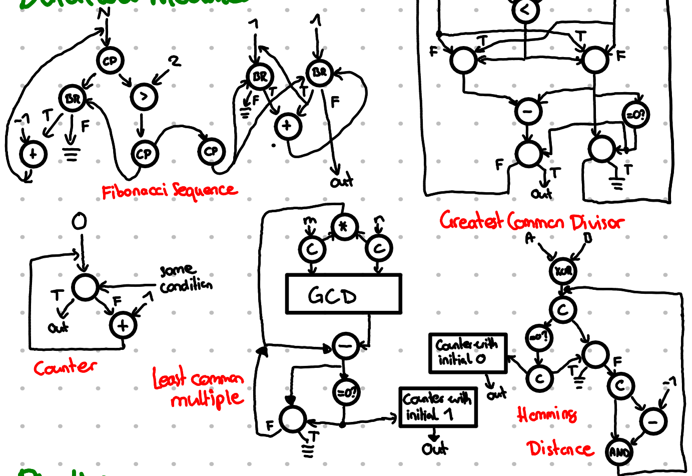
Each instruction takes a single clock cycle and all state updates are made at the end of the cycle. - slowest instruction determines cycle time
+ easy to build

Multi Cycle Machines

Instructions processing is broken into multiple stages/cycles. State changes happen during execution and architectual updates at the end. Instruction processing consists of two components: • Datapath - relay and transform data • Control logic - FSM for signals
+ slowest stage determines cycle time

Dataflow A program consists of dataflow nodes. A node fires (execute) when all inputs are ready

Dataflow Modules



Fibonacci Sequence

Greatest Common Divisor

Counter

Least common multiple

Hamming Distance

Pipelining

The idea is to process proces multiple instructions at once be keeping each stage occupied
In reality there are a few problems, which cause pipeline stalls:

- Data/Control flow dependencies: flow (read after write), output (write after write) and anti (write after read) dependencies. The last two exist due to lack of registers
  beq instructions ⇒ causes flushes
- Resource contention: can be fixed by duplication, increased throughput or detection and stalling
- long latency operations

Handling flow dependencies

e.g. fine-grained multithreading

• stall • eliminate at software level • predict values • do something else
• data forwarding → W → D: internal/register file forwarding
↳ M → E_1: operand forwarding

Pipeline stages

Fetch: Read instruction from memory
Decode: Read operands from register file
Execute: ALU-Operations
Memory: read/write from/to memory
Writeback: write result to register file

Interlocking & Scoreboarding

Detection of data dependencies to ensure correct execution
SW-Interlocking: Compiler inserts nop's
⇒ nops go through all pipeline stages
HW-Interlocking: Stall the pipeline

Out of Order Execution

Move dependent instructions out of the way of independent ones.

In-Order Pipeline with Reorder Buffer

Decode: Access regfile/ROB, allocate entry in ROB, check if it can execute → if yes, dispatch to ALU
Execute: Instruction out of order
Completion: Write result to reorder buffer → else flush pipeline
Retirement/Commit: Check exceptions; if none ⇒ write architectual register file or Mem.

↑ In-order dispatch/execution, OoO completion, in-order retirement.

Tomasulo's Algorithm

Implementing OoO execution. Uses register renaming to eliminate output and anti-dependencies. It further uses reservation stations for individual ops.

1. If reservation station is available
   - instr. + renamed operands inserted into reservation station ← comes from register alias table RAT
   - rename destination register in RAT

   ↳ Else: Stall pipeline

2. While in reservation station:
   - watch common data bus for tag of sources
   - if tag seen ⇒ grab value ⇒ set valid bit
   - if both operands are valid ⇒ instr. ready for dispatch

3. Dispatch instruction to functional unit

4. After instruction finishes
   - put tagged value onto common data bus
   - if register alias table contain tag ⇒ update value and set valid bit
     ↳ write to register file
   - reclaim rename tag
     → no valid copy of tag in the system

VLIW

Compiler finds independent instructions and schedules them into a single VLIW-Instr.
Lock step execution: if one instruction stalls, the whole VLIW stalls

+ simple hardware
+ no dependency checking
+ no instruction distribution

- compiler needs to find N independent Instructions
- lock step causes stalls ↑ complex

Superscalar Execution

Fetch/Decode/... multiple instructions per cycle
+ higher IPC - higher complexity for dependency checking ⇒ more HW

Systolic Arrays

Instead of a single processing element (PE), we have an array of PE and carefully orchestrate the dataflow between them. ⇒ max. comp. on single PE
Difference to Pipelining: Array structure is non-linear and multi-dimensional PE-structure can be multi-directional on different speeds. PE can have memory

Fine Grained Multithreading

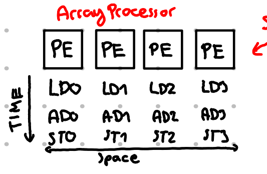
HW has multiple thread contexts (PC + reg) and each cycle the fetch-engine fetches from a different thread.

+ no dependencies
+ no branch prediction
+ Improved throughput, latency, tolerance, utilization

- extra hardware
- reduced single-thread performance
- resource contention
  ↳ dependency checking between threads

SIMD Single Instruction operates on multiple data

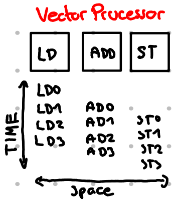
Array Processor



Same op @ same time
Diff ops @ same space
requires N PEs to support vector of length N

diff ops @ same time
Same op @ same space
requires 1 PE per op

Vector Processor



Vector Processing: Perform op on a whole array. Only possible if element are independent

The vector data is stored in N M-bit vector data register.

Vector Control Registers:
- VLEN: length of vector
- VSTR: distance of v.elements in memory
- VMASK: Indicate which elements to operate on

Vector Chaining: Data forwarding from one vector functional unit to another

Vector Stripmining: #data elements > #elements in vector register ⇒ { split into smaller vectors; 8×VLEN 64 + 1×VLEN 15 } (527, 64)

Memory Banking: Memory divided into banks. Access is independent. Can start and complete 1 access per cycle ⇒ can sustain N concurrent accesses if all N are in diff. banks

Min # Banks = max cycle.instr.

- + a lot of work per instruction
- + regular memory access pattern
- + no need for loops
- − works only if parallelism is regular otherwise it is highly in efficient

GPU SIMD engines but programmed using threads (Not SIMD instructions)

SIMT: SIMD instr. (VADD,..) not exposed to programmers ⇒ + treat threads separately

A set of threads executing the same instruction are dynamially grouped into a warp ← done by HW
↑ at same PC

Dynamic Warp Merging: Dynamically merge threads executing the same instr. after branch divergence

#Warps = #threads / #threads per warp

Utilization = # instr. executed / best case when all warps are full instr.

⇔ all branches are taken the same. If >1 instr. takes branch ⇈ all take branch

Branch Prediction

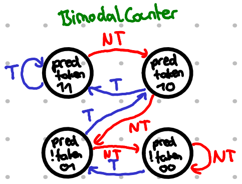
A technique used to predict the next address after a branch. If the prediction is wrong, we need to flush the pipeline (misprediction penalty)

Prediction Schemes

static
- Always (not-) taken
- BTFN (Backward taken, forward not taken)
- Profile Based (likely direction) Hint from compiler

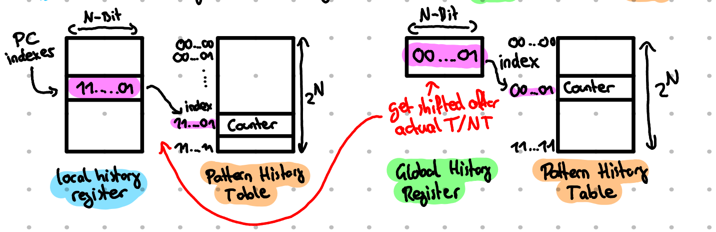
dynamic
- last time predictor: single bit in BTB (Branch target buffer) ↳ loop acc $\frac{N-2}{N}$
- 2-bit counter based prediction
- Advanced algorithms (perceptrons)

Bimodal Counter

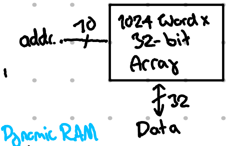


Local correlation: PC indexes local history registers. Entry used to index PHT

Global correlation: global T/NT history of all branches in GHR (used to index PHT)



Memory & Memory Hierarchy



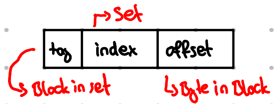
Memory Arrays: stores data, address selection, logic selects row, readout circuitry reads data

Flip-Flop (Latches)	StaticRAM	Dynamic RAM
+ Very fast, parallel access	+ fast	+ cheap
− Very expensive	− serial, expensive	− volatile, slow, serial

Locality: temporal = access to same address in short-time
spatial = access to nearby address

Caching (Exploit locality) Store adjacent data/recently accessed data

Blocks & Addressing cache:
- Memory is divided into fixed-size blocks
- each block/address maps to a set in the cache
- for a cache hit, the tag needs to match + valid bit



$$\text{tag-store size} = 2^{\text{index}} * (n * (\text{tag} + \text{dirty} + \text{valid}) + \text{RP})$$

associativity eg 2-way
replacement policy bits

address size = tag + index + offset

Associativity
- n-way associative cache: n blocks per set/allows n Blocks with same index
- fully associative cache: 1 set ≡ 0 index bits/Memory address can map to any block
- direct mapped cache: 1 block per set

Cache Performance
- cache size, total data = c
- block size = b
- associativity = n

Missrate = #Misses / #total Mem. accesses
Hitrate = #Hits / #total Mem. accesses

$\# \text{Blocks}: B = \frac{c}{b}$

$\# \text{sets}: S = \frac{B}{n} = \frac{c}{b \cdot n}$

Replacement Policies
- FIFO (least written)
- LRU (Least accessed)
- random

Write Policy
Writeback: dirty-bit/write to lower levels when block is evicted
Writethrough: write to all levels immediately ⇒ simpler, but bandwidth intensive

Types of misses
- compulsory miss: first request to cache is always a miss
- capacity miss: cache too small to hold all concurrently used data
- conflict miss: several addresses map to the same set and evict still needed blocks

Improve Cache Performance

Reduce miss rate:
- more associativity
- better replacement policies
- software approaches
- prefetching

Reduce miss latency/cost:
- multi-level caches
- critical words first
- better replacement policy
- software approaches
- prefetching

Prefetching Preload data to avoid misses in cache. Done by SW/HW

stride prefetcher: prefetches cache block in a pattern with certain stride
↳ if stride = 0: next-block prefetching / n-block prefetcher

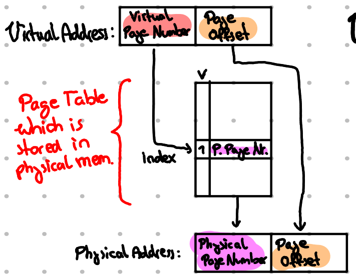
Runahead execution: allows the processor to pre-process instructions during cache misses instead of stalling. Therefore it can detect potential cache misses earlier.

Performance
coverage = #correctly prefetched / # accessed blocks
accuracy = #correctly prefetched / # total prefetched blocks

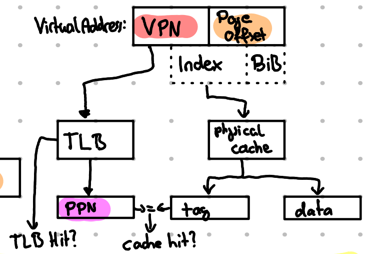
Virtual Memory

Much Larger than physical memory. Virtual address space divided into pages. Physical address space divided into frames. Page Table stores mapping: V → P
If accessed virtual page is not in memory, but on disk ⇒ load in memory (demand paging)

! Physical Memory is a cache for pages stored on disk (fully associative) !



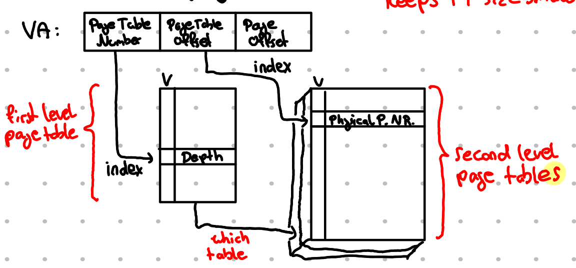
Virtual indexed Physically Tagged



$$\# \text{Virtual Pages} = \frac{\text{Virtual Address Space}}{\text{Page Size}} \qquad \# \text{Physical Pages} = \frac{\text{Physical Address Space}}{\text{Page Size}}$$

Multi-Level page tables:

Keeps PT size small



Memory Protection: different PT for each program

Translation Lookaside Buffer (TLB): cache PT Entries to speed up address translation