Numbersystems

$2^1=2$ $2^2=4$ $2^3=8$ $2^4=16$ $2^5=32$

$2^6=64$ $2^7=128$ $2^8=256$ $2^9=512$ $2^{10}=1024$

$2^{11}=2048$ $2^{12}=4096$ $2^{13}=8192$ $2^{14}=16'384$

$2^{15}=32'768$ $2^{16}=65'536$ $2^{20}=1'048'576$

Dez.	Hex.	Bin.	Dez.	Hex.	Bin.	Dez.	Hex.	Bin.
0	0	0000	5	5	0101	10	A	1010
1	1	0001	6	6	0110	11	B	1011
2	2	0010	7	7	0111	12	C	1100
3	3	0011	8	8	1000	13	D	1101
4	4	0100	9	9	1001	14	E	1110
						15	F	1111

SI-prefixes

Kilo	10^3	2^{10}	peta	10^{15}	2^{50}
mega	10^6	2^{20}	exa	10^{18}	2^{60}
giga	10^9	2^{30}	zetta	10^{21}	2^{70}
terra	2^{12}	2^{40}	yotta	10^{24}	2^{80}

Big Endian (x86): counting Bytes right to left ←

Little Endian: counting Bytes left to right →

unsigned: No sign-flag, all positive $[0, 2^n-1]$

signed: with sign-flag, pos. & neg. $[-2^{n-1}+1, 2^{n-1}-1]$

↳ -0 and +0 exists! No addition!

2's complement: with sign-flag $[-2^{n-1}, 2^{n-1}-1]$

↳ 0 exists once, Addition work

① Invert each bit ② add 1 2's compl

Subtraction: Take 2's compl of 2° numb and than add

Example: Data Pretetching A, A+1, A+2, A+7, A+8, A+9, A+14, ...

a) Stride PF observing last 3 requests: Coverage = 0, PF will learn const. stride of +1, But Pfd blocks useless, none of the requests will be covered.
b) Accuracy 0%. None of prefetched cache blocks will be accurate.
c) New PF, prefetches next N cache blocks N=2, f.ex. the PF requests block 22 and 23 after seeing the request to 21. Every 2 out of 3 requests will be covered, and every 3 out of 6 requests will be correct. Cov = 2/3 Acc = 1/2
d) bandwidth overhead 5/3 For every group of 3 requests, PF will fetch 2 additional blocks.
e) Min. of N to achieve 100% cov. N=5, Examp. ⇒ bandwith is 7/3

Boolean Logic

Product of Sum:
1 All FALSE lines,
2 Maxterms ANDed,
3 Inputs from table inverted

A	B	Y	
0	0	1	
0	1	0	$A+\bar{B}$
1	0	1	
1	1	0	$\bar{A}+\bar{B}$

$Y=(A+\bar{B})(\bar{A}+\bar{B})$

Sum of Products
1 All TRUE lines
2 Minterms ORed
4 NOT inverted, directly taken

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

$Y=\bar{A}B+A\bar{B}$

Identity: $A\,1=A$ $A+1=1$

Null Element: $A\cdot 0=0$ $A+0=A$

Idempotency: $A\cdot A=A$ $A+A=A$

Involution: $\bar{\bar{A}}=A$

Complements: $A\cdot\bar{A}=0$ $A+\bar{A}=1$

Commutativity $AB=B\cdot A$ ....

Assoc.: $A(BC)=(AB)C$

$A+(B+C)=(A+B)+C$

Distr.: $A(B+C)=AB+BC$

$A+(BC)=(A+B)(A+C)$

De Morgan: $\overline{ABC}=\bar{A}+\bar{B}+\bar{C}$

$\overline{(A+B+C)}=\bar{A}\bar{B}\bar{C}$

Covering: $A(A+B)=A$ $A+(AB)=A$ $AB+A\bar{B}=A$

$(A+B)(A+\bar{B})=A$ $(A+\bar{B})\cdot B=AB$ $A\bar{B}+B=AB$

X: Don't care (value 1 or 0) / Illegal value (value 1 and 0)

Z: Floating value (not 1 nor 0)

Logical completeness: Every formula can be represented by the given set of operations

{AND, OR, NOT} {NAND} {NOR}

Example $F=(\overline{A\,B}+C)+A\cdot C$ convert to NAND use only

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

$=\overline{(A\cdot B)\cdot(\overline{C\cdot C})}=\overline{\overline{(A\cdot B)\cdot(A\cdot B)}\cdot(C\cdot C)}$

Example $F=\bar{A}+(\overline{B\cdot C+\bar{A}\cdot\bar{C}})$ convert to NOR use only

$F=\overline{\overline{(\bar{A}+\bar{A})+(\overline{B\cdot C+\bar{A}\cdot\bar{C}})}+\overline{(\bar{A}+\bar{A})+(\overline{B\cdot C+\bar{A}\cdot\bar{C}})}}$

$B\cdot C=\overline{\bar{B}+\bar{B}+\bar{C}+\bar{C}}$ $\bar{A}\cdot\bar{C}=\overline{\bar{A}+\bar{A}+\overline{\bar{C}+\bar{C}}+\overline{\bar{C}+\bar{C}}}$

Combinational Circuits

Transistors MOS (metaloxide semiconductors)

nMOS: conducts if g=1, pull down well but up poorly

pMOS: conducts if g=0, pull up well but down poorly

Logic Gates

AND

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

NAND

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

OR

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

NOR

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

XNOR

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

XOR

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

NOT

A	Y
0	1
1	0

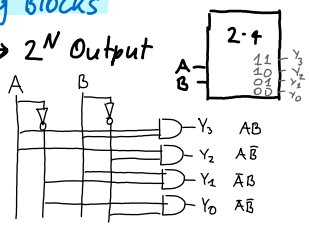
BUFFER

A	Y
0	0
1	1

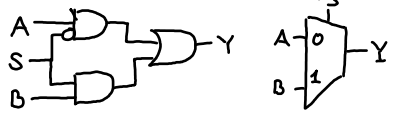
Combinational Building Blocks

Decoder N Input → 2^N Output

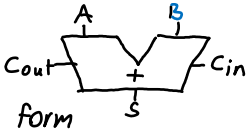
A	B	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



MUX: selects one of n Inputs with $\log_2 n$ select signals



Full Adder:



PLA (Progr. Logic Array)

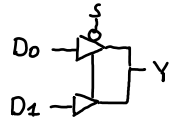
→ directly from Sum of Products form

→ For N-Input logic func, we need 2^n n-Input AND Gates

Tristate Buffer Acts like a switch

E	A	Y
0	0	z
0	1	z
1	0	0
1	1	1

Mux with Tristate Buf



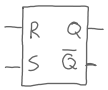
$Y=D_0\bar{S}+D_1 S$

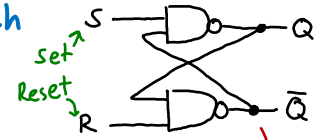
Combinational VS Sequential

- No Memory
- No cyclic paths
- Every node either input or connects to exactly one output
- Has Memory
- Output depends on prior inputs

Sequential Circuits

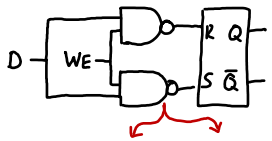
SR-Latch





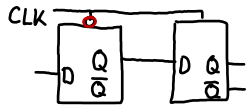
R	S	Q
0	0	!
0	1	1
1	0	0
1	1	Qprev

D-Latch



WE	D	Q
0	0	Qprev
0	1	Qprev
1	0	0
1	1	1

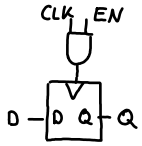
D-Flip-Flop



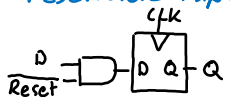
Register ≙ several D Flip Flops

Value from D gets copied to Q at rising clock edge
If clock = 0: Q retains old value

Enabled Flip Flop



Resettable Flip Flop



Reset = true: resets Q to 0 and ignores D

Synchronous Sequential Circuits

- Every element either a register or a comb. circuit
- At least one circuit element is a register
- All registers receive the same clock signal
- Every cyclic path contains at least one reg

Finite State Machines (FSM)

Moore:
- output depends only on current state
- output labeled in each state

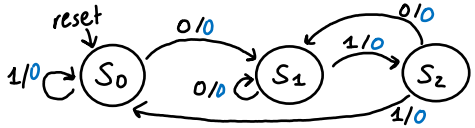
Mealy:
- output depends on current state & input values
- output labeled on the transitions
- A/Y A = value of Input Y = output

⚠ Mealy machines usually requires less states

Area of FSM: #bits for state encoding + # logic gates in next-state logic

Example: How to simplify a FSM?
Does a state has only outgoing edges? Delete
Does two states have the same behaviour? Delete one

Example
Detect 011



State Encoding:

Binary Encoding (00, 01, 10, 11)
$\log_2$(# states) bits needed ⇒ reduces # Flip Flops

One-hot Encoding (001, 010, 100)
states = bits needed ⇒ reduces next-state logic

Output Encoding (010, 000, 110, ..)
⇒ reduces output logic

Correctness of a state diagramm

- Must have reset-line or initial state
- no multiple transitions for same input
- no missing transition
- no unmarked transition
- no mix of Moore / Mealy Labeling

Designing a FSM

1. Identify Inputs & outputs
2. sketch the state transition diagramm
3. Write a state transition table
4. Write an output table 5. Select state Encoding
6. Write Boolean equations for next-state & output logic
7. Sketch circuit

Verilog

~	negation	^	bit-wise XOR
!	logical negation	~^	bit-wise XNOR
&&	log. AND	{...,...}	concatination
||	log. OR	<</>>	shift left / right
&	bit-wise AND	=	blocking at line
|	bit-wise OR	<=	non-blocking at end

Wires:
- always on the left of assign
- cannot be used at left of = / <=
- are used to connect input & output
- can only be assigned once

Regs:
- cannot be used with assign
- in always-blocks, never outside
- cannot be connected to an output port
- cannot be used in input ports

CHECK IF:
→ All Inputs & outputs have to be assigned and names have to match
→ not multiple assignements to same signal
→ names can't start with numbers / No duplicate-names
→ no module recursion
→ enough bits for numbers → Semicolons
→ initialization of variables

Example D-Flip Flop

```
module flop (input clk, input rst, input [3:0] d, output reg [3:0] d);
  always @ (posedge clk, negedge rst)  begin
      if (rst == 1)  q <= 0;        ↖ Don't include for a sync. reset
      else           q <= d;
  end
endmodule
```

Example: Full Adder

```
assign sum = a^b^c_in;
assign c_out = (a&b) | (a&c_in)
             | (b&c_in),
```

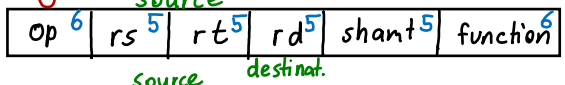
Example Counter

```
always @ *
  begin
    count_next = count;
    count_next += 1;
  end
always @ (posedge clk)
    count <= count_next;
```

MIPS Assembly

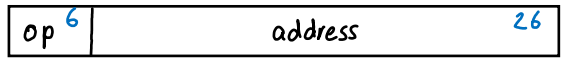
Name	Number	Use
$0	0	const. value 0
$at	1	assembler temporary
$v0-$v1	2-3	return values
$a0-$a3	4-7	arguments for procedure
$t0-$t7	8-15	temporary variables
$s0-$s7	16-23	saved variables
$t8-$t9	24-25	temporary variables
$k0-$k1	26-27	OS temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	procedure return address

R-Type



I-Type

J-Type



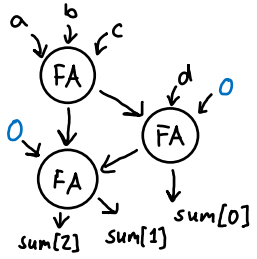
Example Subtraction 64-bit: {$4,$5} - {$6,$7}

```
subu $3, $5, $7        add $2, $6, $2
sltu $2, $5, $7        sub $2, $4, $2
```

Dataflow

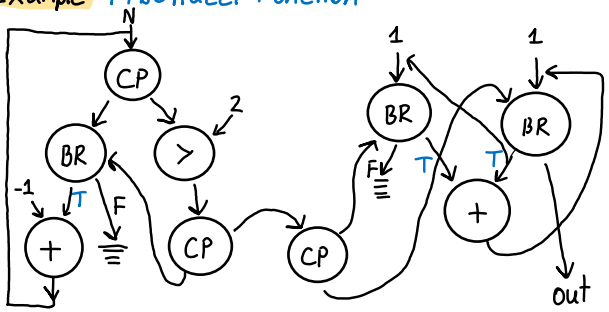
Programm consists of dataflow nodes. A node fires (executes) when all its inputs are ready.

Example Four 1-bit Addition (given Full Adder nodes)



n	required FA
3	1
4	3
5	3
6	4
7	4
8	7

Example Fibonacci Function



Performance Analysis

CPI Cycles per Instruction

MHz = 10^6 cycles/sec ← Clock Freq doesn't matter!

IPC Instruction per Cycle = CPI^{-1}

MIPS Mill. instr. per second = #MHz/CPI = #MHz IPC

Time = #Instructions · CPI / #Hz

Speedup = oldTime / newTime − 1

Higher Frequency ⇒ more Instr. per second (Not IPC!)

Higher IPC ⇒ faster execution of programm (but could need more Instr.)

$$CPI = \sum_{A \in Instr.} (\#cycles\ per\ A) \cdot (proportion\ of\ A)$$

Execution_time = #Instruction · CPI · clock_time

clock_time = 1 / clock_frequency

Example Fibonacci n in $4, result in $2

```
fib:
    addi $sp, $sp, -16   // allocate stack space
    sw   $16, 0($sp)     // save r16 in stack
    add  $16, $4, $0     // r16 = n
    sw   $17, 4($sp)     // save r17 in stack
    add  $17, $0, $0     // r17 = a = 0
    sw   $18, 8($sp)     // save r18 in stack
    addi $18, $0, 1      // r18 = b = 1
    sw   $31, 12($sp)    // save return address
    add  $2, $17, $18    // c = a+b
branch:
    slti $3, $16, 2      // use r3 as temp
    bne  $3, $0, done    // we have finished
    add  $2, $17, $18    // c = a+b
    add  $17, $18, $0    // a = b
    add  $18, $2, $0     // b = c
    addi $16, $16, -1    // n -= 1
    j    branch
done:
    lw   $31, 12($sp)    } restore
    lw   $18, 8($sp)     }
    lw   $17, 4($sp)     }
    lw   $16, 0($sp)     }
    addi $sp, $sp, 16
    jr   $31             // jump to caller
```

```
int fib (int n) {
  a=0; b=1;
  c= a+b;
  while (n>1) {
    c=a+b,
    a=b; b=c;
    n--;
  }
  return c;
```

ISA VS Microarchitecture

ISA Interface between SW & HW. "What the progr sees"

- the machine can't execute subtraction — Instructions
- the machine doesn't have condition codes — Instructions
- A 5-bit imm. can be used in ADD — Instructions
- There are 8 GPR — Memory
- The PC width — Memory
- The mem-mapped location of exception vectors — Exception Handling
- Func. of each bit in a progr branch-predictor config. reg
- The execution order of load & store in a multicore CPU
- The mem.add. modes available for arith. ops
- The ability to flush the TLB from the OS — Exception Handling
- SIMD Instr. support
- The addr. of mem.-mapped devices of the CPU (e.g. keyb.) — Exception Handling
- The #bits required for dest reg of load
- Support of integer division/multiplication
- Mechanism to enter in a system call in the OS
- size of addr. memory

Microarchitecture: specifies the underlying implementation

- ALU doesn't have subtract unit — Pipelining
- It takes n cycles to execute an ADD — Pipelining
- A 2-to-1 MUX feeds one of the ALU inputs — Pipelining
- The reg file has 1 input & 2 output ports — Pipelining
- Latency of branch predictor misprediction — Caching
- The size of physical memory page — Caching
- The number of pipeline stages in the CPU — Error Correction
- The number of cache sets per cache hierarchy — Phys. Struct.
- The Integer multi. Algo. of the ALU — Phys. Struct.
- CPU: size of ROB, fetch width, #non-prog. regs — In order VS OoO Exec
- Branch misprediction penalty — In order VS OoO Exec
- The mem. controller's scheduling algorithm — In order VS OoO Exec

Pipeline

Stages

Fetch: CPU reads Instr. from Instr. memory

Decode: CPU reads source operands from reg. file & decodes the Instr to produce the control signals

Execute: The processor performs the comp. with ALU

Memory: The proc. reads or writes data memory

Writeback: The proc. writes the result to the reg. file

Scoreboarding: Each reg has a valid bit (1 if data is there), stall pipeline if not all sources ready.
Problem: stalls for all dependencies (not only flow)

Data Forwarding: Forward the data of a stage directly to where it is needed (solves RAW)
- ↳ W→D: internal / register file forwarding
- ↳ M→E_a: operand forwarding

Causes of stall: Resource Contentation, Data Dependency, Control Dependency, long multi-cycle ops

Data Dependencies: Flow (RAW) "real", Output (WAW), Anti (WAR) WAW & WAR only a probl. bc regs are limited

Handling Flow Dependencies Detect + stall, Detect + forward, Detect + eliminate on SW level, Detect + move out of way (OoO Execution), Predict, Fine-grained Multithreading

Examples Pipelined Machine

```
MUL R3 ← R1, R2
ADD R5 ← R3, R4
ADD R7 ← R2, R6
ADD R10 ← R8, R9
MUL R11 ← R7, R10
ADD R5 ← R5, R11
```

In-order dispatch without forwarding

```
F D 1 2 3 4 5 6 W
  F D - - - - - - D 1 2 3 4 W
    F - - - - - - - D 1 2 3 4 W
                    F D 1 2 3 4 W
                      F D - - - - D 1 2 3 4 5 6 W
                        F - - - - - D - - - - - - D.... W
```

In-order with forwarding

```
F D 1 2 3 4 5 6 W
  F D - - - - - 1 2 3 4 W
    F - - - - - D 1 2 3 4 W
                  F D 1 2 3 4 W
                    F D - - - 1 2 3 4 5 6 W
```

Out-o-O with Form.

```
F D 1 2 3 4 5 6 W
  F D - - - - - 1 2 3 4 W
    F D 1 2 3 4 W
      F D 1 2 3 4 W
        F D - - - 1 2 3 4 5 6 W
          F D - - - - - - - - - 1 2 3 4 W
```

Out-of-Order Execution

Idea: move dependant Instr. out of the way of independant ones. +Latency tolerance

Reorder Buffer (ROB): complete Instr. OoO but reorder them before making results visible to architect state

Example Prefetching

A:
```
uint8_t a[1000];
sum = 0
for (i=0 ; i < 1000; i += 4)
    sum += a[i]
```

B:
```
uint8_t a[1000];
sum = 0
for (i=1 ; i < 1000; i += 4)
    sum += a[i]
```

a) Accuracy & Coverage

A: a[0], a[4], a[8], ... a[996]
#mem. access = 1000/4 = 250
a[0], a[4] are misses, then the prefetcher learns that stride is 4.
249 cache blocks are prefetched, while 248 are used ⇒ accuracy = $\frac{248}{249}$ coverage = $\frac{248}{250}$

B: a[1], a[5], a[16], a[64], a[256] → 5 acc. to MEM But stride between acc. not constant → Prefetcher cannot prefetch → acc. and cov. is 0

b) Better Prefetcher for A: next block prefetcher, a[4] hit as well ⇒ acc/cov = $\frac{249}{250}$
Better Prefetcher for B: Maybe runahead executions / dual-core exec. improve

c) Runahead for A: No. Could provide good pref., But maybe not all blocks gets prefetched, as runahead exe. happens only when stalled on cache miss
Runahead for B: Yes. could potentially prefetch some cache blocks, especc. as the addr. of cache blocks does not depend on the data from the pending cache miss.

Tomasulo's Algorithm

Tag comp. per res. stat. = #sources of ops • # func. units = 2·8 = 16
Total tag comp = #units • #st. per unit • #comp. per st. + #reg • #fun. Un
= 16·16·8 + 32·8 = 2304
min tag size = $\log_2$(#units • #station per unit) = log(16·8) = 7
min size of reg. alias table = #regs • (64/32-bit + tag size + 1)
= 32 · (64 + 7 + 1)
min tot. size of tag store = #regs • tagsize + #Regs • #func units • # comp per st. • tagsize

Example:

```
MUL R1, R2 → R3
ADD R3, R4 → R5
ADD R2, R6 → R7
ADD R8, R9 → R10
MUL R7, R10 → R11
ADD R5, R11 → R5
```

```
F D 1 2 3 4 5 6 W
  F D - - - - 1 2 3 4 W
    F D 1 2 3 4 W
      F D 1 2 3 4 W
        F D - - 1 2 3 4 5 6 W
          F D - - - - - - - 1
```

8 func units, 32x64 bit registers,
1 reservation stations, 2 source reg/res st.

Reg	Valid	Tag	Value
R1	1	-	1
R2	1	-	2
R3	0	x	-
R4	1	-	3
R5	0	d(a)	-
R6	1	-	4
R7	0	b	-
R8	1	-	5
R9	1	-	6
R10	0	c	-
R11	0	y	-

Source 1 + Source 2

	V	Tag	Value	V	Tag	Value
a	0	x		1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	

Source 1 • Source 2

	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	0	b		0	c	
z						
t						

Delayed Branching

Delay slot: # Instr. that need to be fetched after a branch. Compiler takes Instr. if they don't influence the branch itself, else delay slot filled with NOP's.

Branch Prediction

Static: Always taken: ~30% - 40% accuracy
Always not taken: ~60% - 70% accuracy
BTFN: Good for loops (fails once)

Dynamic: Last time predictor: Single bit per Branch stored in the Branch Target Buffer (BTB). ⇒ Misses 2x for loops $\frac{n-2}{n} \cdot 100\%$ accuracy.

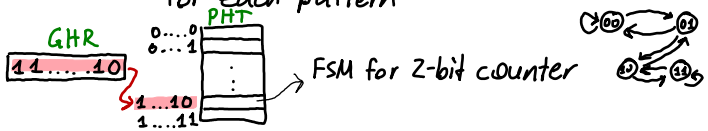
Two-bit counter based prediction: Counter is capped ⇒ 00 - 1 = 00, 11 + 1 = 11. If taken +1, else -1 ⇒ accuracy: ~85% - 90% (00, 01 ⇒ not taken)

Local VS Global: Local: PC based ⇒ no interference bw Br.
Global: single counter globally shared

Global Branch Correlation: The idea is that recently executed branch outcomes are correlated with the outcome of the next branch ⇒ Prediction based on outcome of last branch with same global branch history

First level: Global Branch history register keeps track of the last branch outcome } GHR Index of PHT

Second level: Pattern history table keeps a 2-bit counter for each pattern



VLIW (Very Long Instruction Word)

The compiler finds independant instructions and statically schedules them into a single VLIW instr.

Lock step Execution: If 1 Instr stalls, whole VLIW stalls
+ no dependency checking necessary
— compiler needs to find N independant Instr. per cycle

Example VLIW VS in-order Superscalar:
Why VLIW Faster: supersc. in-order ⇒ bubbles in pipeline. VLIW can reorder
Why Supersc faster: VLIW needs NOP's ⇒ lower I-Cache hit rate & higher fetch bandwidth

Superscalar Execution

Fetch/Decode/... multiple Instr per cycle
+ higher IPC — complex dependency checking ⇒ more hardware

Fine-Grained Multithreading

Hardware has multiple thread contexts (PC + Reg for each thread) and each cycle we fetch from a different thread

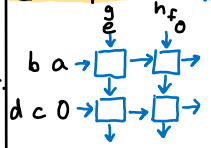
+ no dependencies — extra hardware
+ no branch pred. — reduced single-cycle perform
+ improved throughput, latency, utilization — resource contentation ⇒ dependency checking betw. threa.

Systolic Arrays

Instead of a single Processing element (PE) we have an array of PE and carefully orchestrate the dataflow between them ⇒ Maximize comput. on PE's

Difference Pipelining: Array struct. non-linear & multi-dimensional. PE's can have: connections multi-directional with diff. speeds and local memory & execute kernels

Example: Matrix mult. $\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{pmatrix}$



R + R + M·N
Q = N P = M

SIMD (Single Instruction multiple Data)

Array Processor: Same op @ same time ("operates")
Vector Processor: Same op @ same space

Vector Processor

Performs ops on whole array ⇒ ops on each elem. independ. (must be)
Data is stored in vector registers
Vector chaining Data forwarding for vectors (dispatch as soon as individual element is ready.
Stride: distance of element (that are adjacent) in Mem
Strip mining If vector too long ⇒ split in multiple smaller

GPU

GPU's are SIMD engines but using Threads (SIMT)
Warp: Set of threads that execute the same instruction (i.e., at the same PC)
# warps = # threads / (thread per warp)
utilization = (# warps perf. useful work) / (# thread slots)
Dynamic warp merging: Merge threads executing the same instr. after branch divergence. This forms new warps from the warps waiting.

Example

Memory Hierarchy & Caching

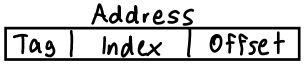
Register
L1-Cache
L2-Cache
L3-Cache
RAM
Storage

Memory Array: stores data, addr select. logic selects row, readout circuitry reads data

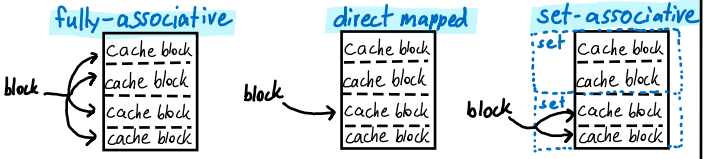
Memory Banking: Multiple mem. units with common data & addr. Bus, helps to resolve long latency. Units can be accessed individually

Locality: temporal = access to same addr. short time
spatial = access to nearby address

Blocks & Addressing Cache:
→ Mem is divided into fix-sized blocks
→ Each block maps to a location in the cache (index bits)
→ For a cache hit the tags need to match



Offset = Byte in Block
Index = Row in tag store
Tag = which block in the set



Associativity:
→ If multiple blocks have the same index → conflict misses
→ n-way associative allows n-Blocks with same index
1-way assoc. → direct mapped no index → fully assoc.

Cache Performance

Cache size C: total data capacity (excl. tags)
Block size b: data associated with addr. tag
Associativity N: # Blocks represented in same index

Blocks $B = c/b$ #sets $S = B/N$

Average Memory Access Time:

$AMAT = (\text{hit-rate} \cdot \text{hit-latency}) + (\text{miss-rate} \cdot \text{miss-lat.})$

Recursive Latency $T_i = t_i + m_i \cdot T_{i+1}$ m_i: miss rate

Replacement Policies: • FIFO (first-in-first-out)
• LRU (last-recently-used) • Random

Classification of Misses:
Compulsory Miss: first reference is always a miss
Capacity Miss: Cache is too small
Conflict Miss: all other misses (more associativity)

Improvement ideas: • reduce miss rate
• reduce miss latency or cost • reduce hit latency or cost

Prefetching

Improve cache performance by preloading data before it is used (speculative) in order to reduce mem. latency.

Runahead execution: allows the proc to pre-process instr. during cache misses instead of stalling

Accuracy = (used prefetches) / (sent prefeches)
Coverage = (prefetched misses) / (all misses)
Timeliness = (on-time prefetches) / (used prefetches)

Virtual Memory (VM)

Virtual address space is divided in pages, while physic. addr. space is divided into frames. Page Table stores mapping: Virtual → Physical together with a valid bit

$\#\text{virtual pages} = \frac{\text{Virt. Addr.}}{\text{Page size}}$ $\#\text{physic pages} = \frac{\text{Phys. Addr.}}{\text{Page size}}$