Metrics

- $\text{Power}_{\text{dynamic}}$ = 1/2 × Capacitive load × Voltage^2 × Frequency Watts
- $\text{Power}_{\text{static}}$ = $\text{Current}_{\text{static}}$ × Voltage Watts
- $\text{Energy}_{\text{dynamic}}$ = Capacitive load × Voltage^2 Joules

Reliability equations

$$MTTF = \frac{\text{total hours of operation}}{\text{total number units fail}} = \frac{\text{number of units} * \text{how many hours until fail}}{\text{how many units fail}}$$

↳ Mean Time to Failure

$$FIT = \frac{10^9}{MTTF}$$ → Failures In Time (per billion hours)

$$FIT_{system} = \sum_{i=1} FIT_i$$

$$MTTR = \frac{\text{total maintenance time}}{\text{total number of repairs}}$$ → Mean Time To Repair

$$MTBF = MTTF + MTTR$$ → Mean Time Before Failure

$$\text{Availability} = \frac{MTTF}{MTTF + MTTR}$$

SpecRation and performance

Design X is n times faster than design Y

$$n = \frac{\text{execution time Y}}{\text{execution time X}} = \frac{\frac{1}{\text{performance Y}}}{\frac{1}{\text{performance X}}} = \frac{\text{Performance X}}{\text{Performance Y}}$$

Weighted arithmetic mean $\sum_{i=1}^{n} \text{Weight}_i \times \text{Time}_i$

SpecRation: Measuring against a reference computer

$$\text{SpecRation}_A = \frac{\text{Execution Time reference}}{\text{Execution Time A}} = \frac{\text{Performance A}}{\text{Performance reference}}$$

Geometric Mean: $\sqrt[n]{\prod_{i=1} \text{SpecRation}(a)}$

Pros and Cons Geometric Means

Pros: 1) Ανεξαρτησία από το program frequencies

2) Δεν επηρεάζεται από τα διαφορετικά χαρακτηριστικά κατανομής τιμών

Cons: 1) Δεν προβλέπει τον χρόνο εκτέλεσης

2)

Qualitive principles of design

Taking advantages of Parallelism

- Use pipeline to overlap instructions
- Use multiple excution units
- Use multiple cores
- Use multiple processors to increase throughput

Locality

- Program reuse instructions and data
- 90-10 rule: 90% of execution time spent running 10% of instructions
- Programs acces data in nearby addresses (spatial)

Make the common case fast

- Trade-off's in design (e.g. performance vs power/area)
- Provide efficient design for the common case
- amdahl's law

- Amdahl's law

$$\text{speedup} = \frac{\text{execution time old}}{\text{execution time new}} = \frac{1}{(1 - \text{fraction enchanced}) + \frac{\text{fraction enchanted}}{\text{speedup enchanted}}}$$

πχ 2 cores, 40% Parallelism $\Rightarrow$ speedup $= \frac{1}{(1-0,4) + \frac{0,4}{2}}$

2 cores, Program 1 $\rightarrow$ 25% use and 40% parallelism

Program 2 $\rightarrow$ 75% use and 60% Parallelism

$$\Rightarrow \text{speedup} = \frac{1}{0,25 \cdot \left((1-0,4) + \frac{0,4}{2}\right) + 0,75\left((1-0,6) + \frac{0,6}{2}\right)}$$

- Processor Performance

$\rightarrow 1\,GHz = 1\,ns$

CPU time = Instructions × CPI × (cycle time) $\quad 4\,GHz = 0{,}25\,ns$

$\rightarrow 1\,MHz = 1000\,ns$

Calculate CPI:

Op	Freq	CPI	F * CPI	%Time
ALU	50%	1	0,5	33% → $\frac{0,5}{1,5}$
Load	20%	2	0,4	27% → $\frac{0,4}{1,5}$
Store	10%	2	0,2	13% →
Branch	20%	2	0,4	27% →
			1,5	

Pipeline doesn't help latency of single task, it helps throughput of entire workload

Limits of Pipelining:

- Structural hazards
- Data Hazards: RAW, WAW, WAR
- Control hazards: Branches and Jumps

Speedup of Pipeline

① $\text{speedup} = \frac{\text{Aver. Instruction time unpipelined}}{\text{Aver. Instr. time pipelined}} = \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycles unpip}}{\text{Clock cycles pip}}$

② CPI pipelined = Ideal CPI + Pipeline stall clock cycles per instruction

$= 1 + \text{Pipeline stall clock cycles per instruction}$

① + ② $\Rightarrow \text{speedup} = \frac{1}{1 + \text{Pipeline stalls per instruction}} \times \frac{\text{clock cycles unpipelined}}{\text{clock cycles pipelined}}$

$= \frac{1}{1 + \text{Pipelin stalls per instruction}} \times \text{pipeline depht}$

→ How to avoid stalls,

- Forwarding to avoid data hazards

  BUT: You can't avoid stall after load instruction if next instruction use the same register as input

```
lw  r1, 0(r2)
add r4, r1, r6      stall
```

- Software scheduling to avoid load hazards

Branch Stalls Impact

What to do with the 3 clocks stall in Branches;

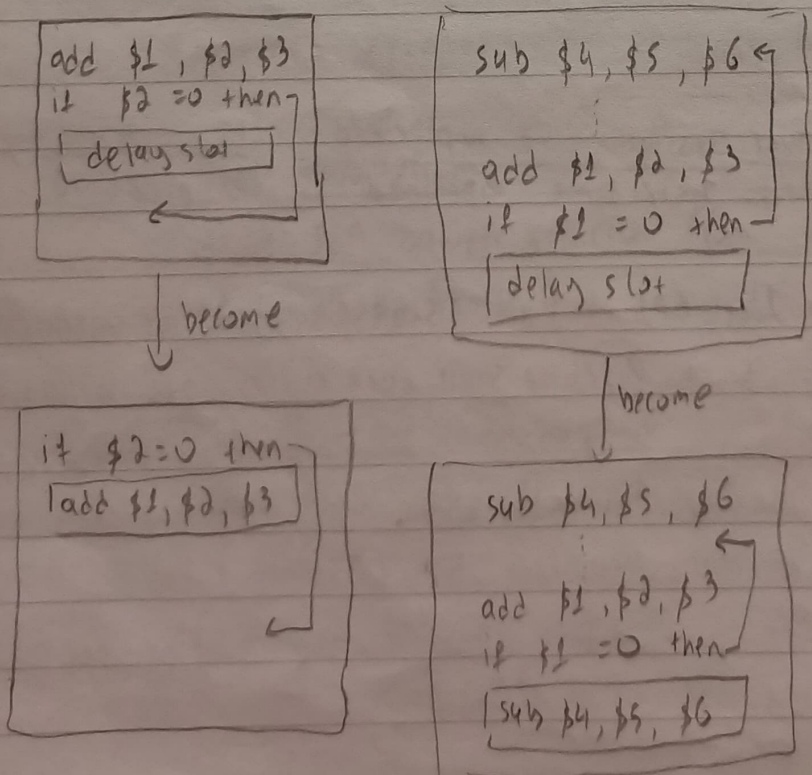
1) Determine branch taken or not taken sooner
2) Compute taken branch addr. earlier.

MIPS Solution: Test if register = 0 or ≠ 0

How: 1) Move zero test to Instruction Fetch / Register Fetch stage
2) Adder to calculate new PC in Instr. Fetc / Reg. Fetch. stage

(SOSH) Always have 1 clock cycle penalty vs 3 were before

⇒ 1 delay slot after every branch



Exception and Interrupt

Problem: Exception appears - happens between 2 instructions (i and i+1)
→ The effect of all istr. up to and including i is totaling complete
a) No effect of any istruction after i can take place
The interupt (exception) handler either abort program or restart at instruction i+1

HW have to remember where is the interupt and stop the instructio at commit point.

Summary about hazards,

- Structural: need more HW resources
- Data (RAW, WAW, WAR): need fowarding, compiler scheduling
- Control (Branch, jumps): delayed branch, prediction

Instruction Level Parallelism,

Instruction level Parallelism

- Potential overlap among instructions
- Parallel or out-of-order executions
- Requires extensions on the simple pipeline

Dynamic Instr. sheduling scoreboard

Loop level Parallelism

- Exploit ILP between istruction from different basic blocks (eg. iterations of a loop)

SOS Must Maintain: - Dependence
→ Data flow: the flow of values between istruction that writes (produce) and instruction that read (consume), RAW, WAW, WAR
→ Exception behaviors: changes in execution order should not change the order of exceptions or generate new ones

Scoreboard

Implications

- Solution for WAR:
  - stall registers write-back untill registers have been read
  - read registers only on the Read operand stage
- Solution for WAW:
  - detect this hazard and stall the new istruction (no issue) untill the previus instruction has complete
- There is no register renaming
- Multiple instructions on execution stage
- The scoreboard keeps information about dependencies of the instructions that have been issued
- The scoreboard pipeline replace the Instruction Decode stage with 2 stages, issue and read operands

4 stages of Scoreboard

- Issue: decode istruction & record data dependences & check for structural hazards
  - Instruction issued in order (for hazard checking)
  - Dont issue if structural hazard
  - Dont issue if instruction is output dependent on any previus issue that didn't complete (no WAW hazards)

- Read Operands: wait for data hazards to be resolve, then read registers
  - All real dependencies (RAW hazards) resolved in this stage, since we wait for instructions to write back data
  - No forwarding of data

- Execution: execute the instruction
  - After complete, it notice the scoreboard that it has complete

- Write result: write result on register
  - Stall untill there are no WAR hazards with previus instructions

Scoreboard Control

Issue: structural hazards, WAW
ReadOper: RAW
Execute: —
WriteBack: WAR

Istruction status

	Issue	Read oper	Exec	write Back

Fractional unit status

	Busy	Op	Fi (destination reg)	Fj (source 1)	Fk (source 2)	Qj (function unit produce Fj)	Qk (functional unit produce Fk)	Rj (flag to know if Fj is ready)	Rk (flag to know if Fk is ready)
Integer									
Mult1									
Mult2									
add									

ISOS when we issue an istr. if the Fj or Fk is ready we mark Rj, Rk with YES, but after read oper we mak Rj, Rk with NO

Register result status

	R0	. . .	R13		R30
FU	Integer		Mult1		

ISOS when we issue an istruction we store here that Ri is going to get data.

Dynamic Instruction scheduling
Tomasulo

Tomasulo

Vs Scoreboard

- FU buffers called "reservation station RS" and hold the register/variable values needed for execution
- The source registers are replaced with the actual values or pointers to the RSs that will produce the values. Register renaming
  - avoids WAW, WAR hazards
  - more RSs than real registers (more optimizations)
- Results arrive to the FUs from the RSs, not via register file but over Common Data Bus (CDB), that broadcasts results to all FUs
- Load and Stores use separate FUs with RSs as well
- Intiger Instructions can go past branches, allowing FP ops beyond basic block in FP queue
- Control logic & buffers are distributed with the FUs vs centralized in Scoreboard

Three Stages of Tomasulo Algorithm

- Issue: Get a new instruction from the instruction queue.
  - If there is a vacant reservation station (no structural hazard) then issue instruction and sent operand values or keep track of the FU/RS that will produce the operant (register renaming)

- Execution: Execute on the FU
  - when all operands are ready in RS start execution
  - when not ready => watch the CDB for the result

  SOS: They can be ready multiple instruction for execution. for the FP adders it just select one of them random BUT for the Memory Unit we need 2 steps. First we calculate the base register when register is ready and then we add the offset and we store it in load or store unit. All loads and stores are execute with the right order.

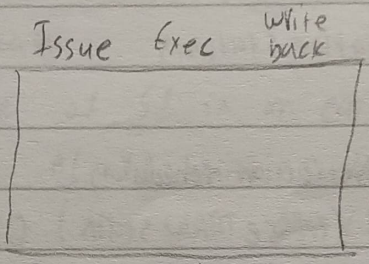
- Write result (commit) 1) Write the result in the CDB and all waiting FUs/RSs get the result 2) write to register file, 3) Mark the reservation station available

Common Data Bus Vs Typical data bus

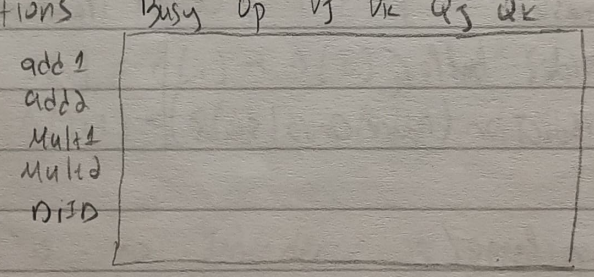
- Typical data bus: data + destination ("go to" bus)
- Common data bus: data + source/reservation station tag ("come from" bus)
  - masters send to all slaves
  - does arbitration and the broadcast
  - 64 bits data + 4 bits of FU source address
  - write result if it matches the expected FU

Tomasulo Data structures

Instruction Status:

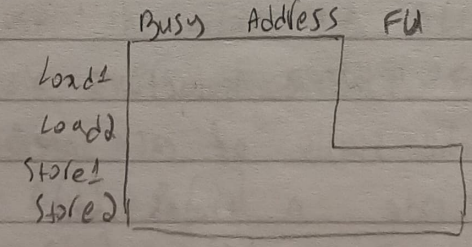


Reservation Stations

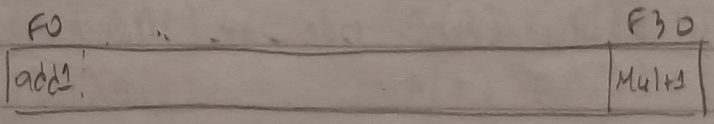


source1 → source2 → from where Vj is waiting → from where Vk is waiting

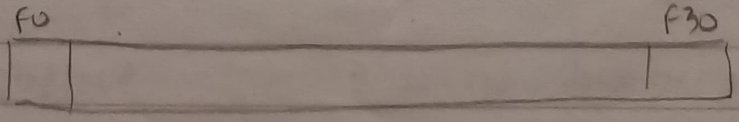
ISSUE if Vj, Vk are ready we don't write anything in Qj, Qk



Register Result Status



We write where the data are going to come



We write the data after write back

Tomasulo Vs Scoreboard

Tomasulo	Scoreboard
• Pipelined FUs (6 load, 3 store, 3 +, 2 ×/÷)	• Multiple FUs (1 load/store, 1 +, 2 ×, 1 ÷)
⇒ window size ≤ 14 Instructions	⇒ window size ≤ 5 Instructions
• No Issue on structural hazard	• Same
• WAR : renaming avoid them	• stall
• WAW : renaming avoid than	• stall
• Broadcast result from FU	• write/read registers
• Control : reservation stations	• Central scoreboard

(Static Instr. scheduling)

Dynamic Instruction scheduling	Static Instruction Scheduling (SW, compiler)
• Scoreboard (reduse RAW stalls)	• Loop unrolling
• Register Renaming (reduse WAW, WAR)	• SW pipelining
- Tomasulo	• Trace scheduling
- Reorder Buffer	
• Branch Prediction (reduce control stalls)	

Make Loop faster

1) Find <stalls> between instructions and shedule the loop code to reduce the stalls of each interation (RAW)

2) Unroll the code n times and change the values of load & store intruction in every other iteration (± Immidiate)

3) Find WAR, WAW and rename registers

4) Rewrite the code to reduce <stalls> (RAW)

When is it safe to unroll?

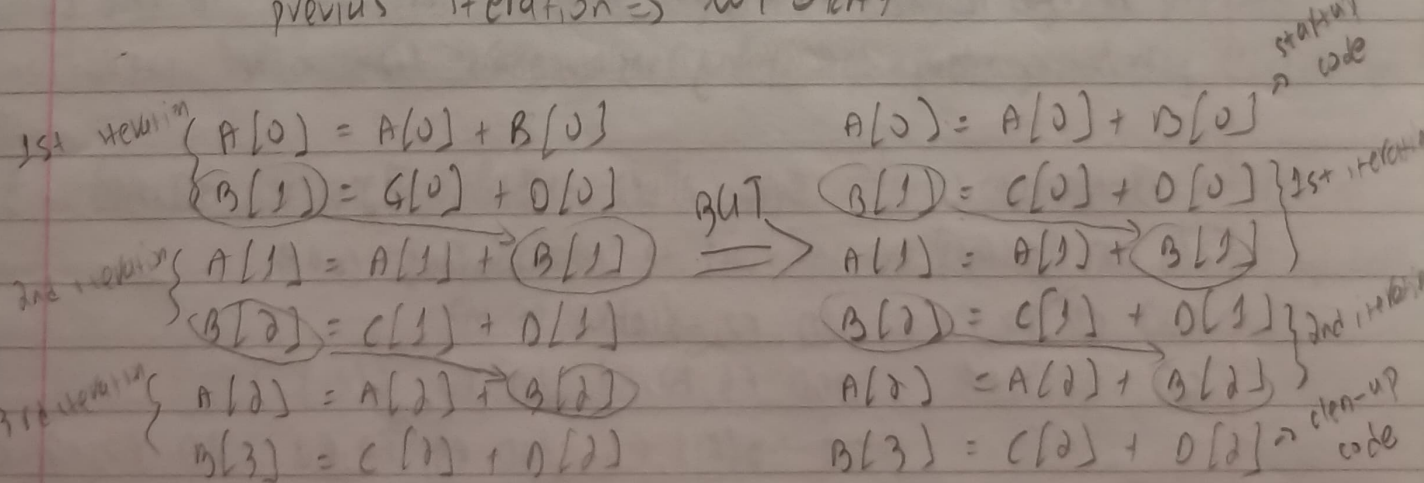
when we don't have dependence across iterations
(loop-carried dependence)

eg. 1)
```
for(i=0; i<100; i++) {
    A[i+1] = A[i] + C[i]    /* S1 */
    B[i+1] = B[i] + A[i+1]  /* S2 */
}
```

- S2 use the value of A[i+1], compute by S1 in same iteration => okay
- S1 use a value of S1 in an earlier iteration, since i iteration produse A[i+1] which is read in i+1 iteration
  The same for S2 => NOT OKAY

2)
```
for(i=0; i<100; i++) {
    A[i] = A[i] + B[i];   /* S1 */
    B[i+1] = C[i] + D[i]; /* S2 */
}
```

- There is no dependence is same iteration => OKAY
- S1 use the value of B[i] which is produce in previus iteration => NOT OKAY



loop-carried dependence

Now the dependencies are in the same iteration so its OKAY to UNROLL

```
for(i=0; i<100; i++) {
    A[i] = A[i] + B[i]      /* S1 */
    B[i+1] = C[i] + D[i]    /* S2 */
}
```

⇓

```
A[0] = A[0] + B[0]          → start-up code
for(i=0; i<99; i++) {
    B[i+1] = C[i] + D[i];   /* S2 */
    A[i+1] = A[i+1] + B[i+1]; /* S1 */
}
B[100] = C[99] + D[99];     → clean-up code
```

Isosi The longer the dependence distance, the more potential to extract parallelism

Software Pipeline

If the iterations of the loop are independent, then we can exploit ILP from different iterations of the loop

So we reduce <stalls> between instruction in the same iteration by executing instructions from other iterations

Iteration 0	Iteration 1	Iteration 2	Iteration 3
LD F0, 0(R1)		→ startup code	
ADD F4, F0, F2	LD F0, -8(R1)		
SD 0(R1), F4	ADD F4, F0, F2	LD F0, -16(R1)	
	SD -8(R1), F4	ADD F4, F0, F2	LD F0, -24(R1)
		SD -16(R1), F4	ADD F4, F0, F2
			SD -24(R1), F4

RAW ⇒ stall in every iteration

↓ finish-up code

Before: Unroll 4 times

```
Loop: LD   F0, 0(R1)        1st iteration
      ADD  F4, F0, F2
      SD   0(R1), F4
      LD   F6, -8(R1)       2nd iteration
      ADD  F8, F6, F2
      SD   -8(R1), F8
      LD   F10, -16(R1)     3rd iteration
      ADD  F12, F10, F2
      SD   -16(R1), F12
      LD   F14, -24(R1)     4rd iteration
      ADD  F16, F14, F2
      SD   -24(R1), F16
      SUBI R1, R1, #32
      BNEZ R1, Loop
```

After: Software Pipelined

```
      SD   0(R1), F4    store M[i]
ADD:  ADD  F4, F0, F2   add M[i-1]
      LD   F0, -16(R1)  Load M[i-2]
      SUBI R1, R1, #8   i = i-1
      BNEZ R1, Loop
```

RAW hazards convert to WAR hazards

Loop unrolling Vs Software pipelined

increase performance of the loop

Loop unrolling: Pros: • Reduce cycles of each iteration takes?

Cons: • Increase the code size
• We need register renaming => register pressure
• The criteria for loop unrolling are very restrictive

Software pipelined: Pros: • Reduce stalls due data dependencies in iterations

Cons: • Overheat at start and the end of loop (start-up code, end-up code)

Static Vs Dynamic Prediction

Static: 1) Stall untill i know if i am taken
2) Always say taken
3) Always say not taken
4) MIPS - always lose 1 cycle to caclulate taken or not

Dynamic: 1) BHT (Branch History Table)
2) BTF (Branch Target Buffer)

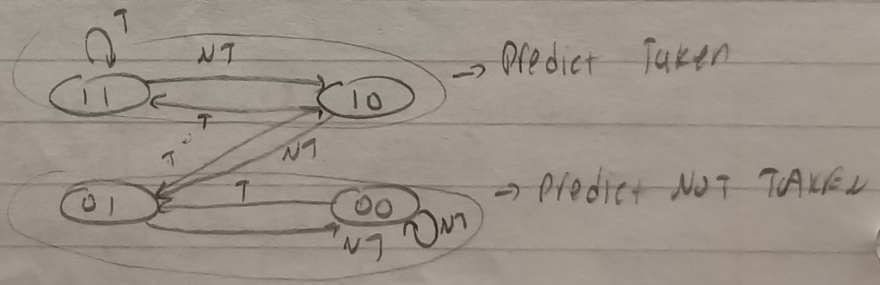
Branch History Table (BHT)

- 1 bit Table

According to the last bits of PC index specifies if branch was taken or not taken last time

limitations: Always have 2 mispredictions:
first time: when it predicts branch not taken because was 0
last time: when it predicts branch taken because was 1

- 2 bit Table:



Add hysteresis in predictions

Correlation predictors - 2 level predictors

Phedictor: (4, 2) , (k, n)

- 4: four bits of history
- 2: 2 bits counter

=> Total bits = $2^{k + \text{addr-size}} * n$

Taxonomy of two-level predictor

X A y

- X: history register
- y: history table

X = G, X = P, X = S

y = g, y = p, y = s

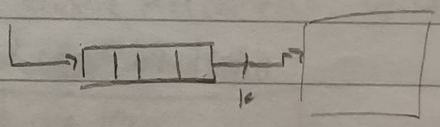
G, g => global

P, p => per-branch

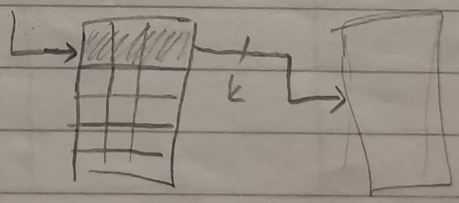
S, s => set-associate per branch

examples: for bra==0 or nla!=0 ?

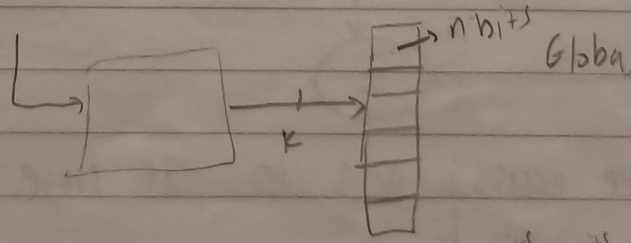
First Level: Global =>



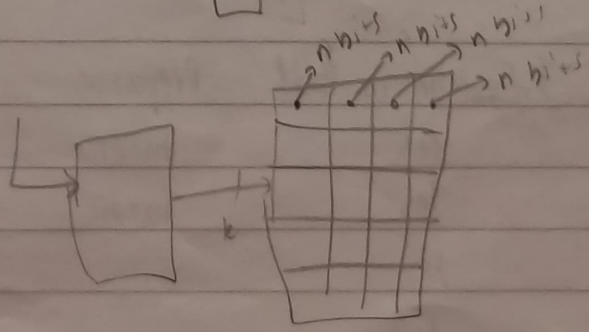
P, S =>



Second Level:

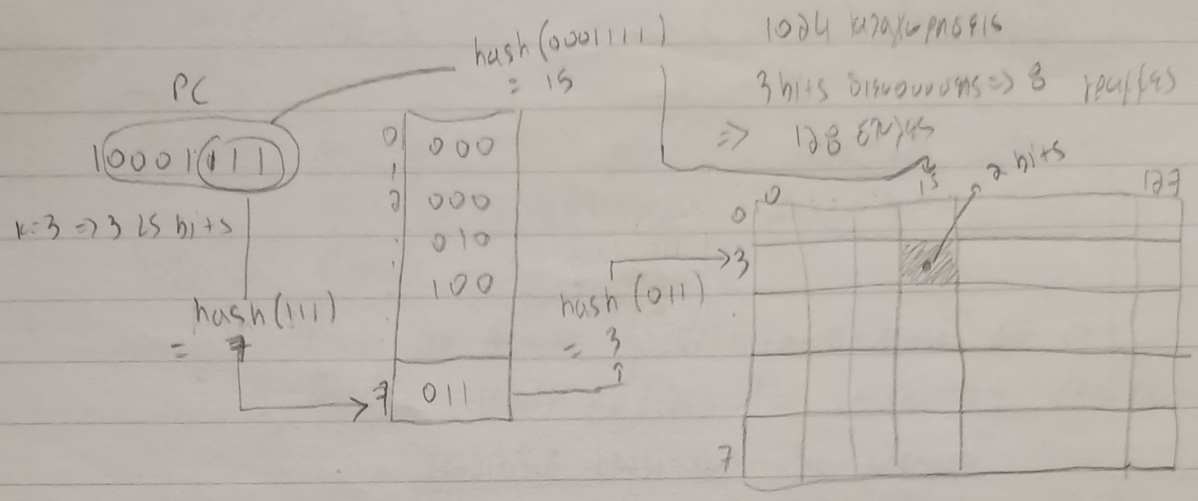


P, S =>



Example: (3, 2), 1024 maximum bits

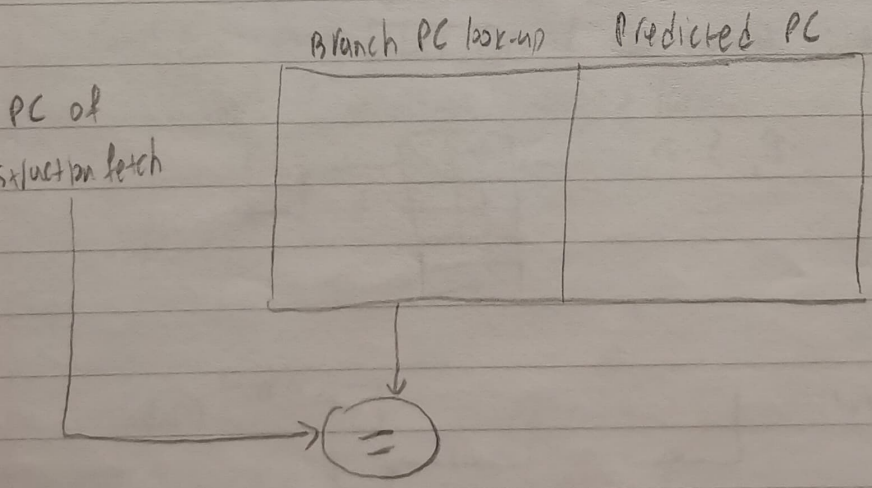
PAp



Branch Target Buffer

- A look-up table with PC address of branches. When the PC matches, it provides the target address of the branch and TAKEN or NOT TAKEN

  Optimization: Stores only the taken branches



We access BTB in IF stage: save a cycle

Instruction in Buff	Prediction	Actual Branch	Penalty Cycles
Yes	taken	taken	0
Yes	taken	not taken	2
No		taken	2
No		not taken	0

Indirect Jumps

switch/case statements, gotos, function returns

Problem: Not good accuracy when procedure called from multiple sites

Solution: Return address Stack

Push a return address in stack at call

Pop a return address from stack at return

Predicated Execution

Convert branches to conditionally executed instructions

Pros: • Reduce branch pressure - rate of misspredictions

• Conditional Move (CMOV)

• May reduce code size

• May be quicker in simple arithmetic operations

Cons: • Takes a clock even if "annulled"

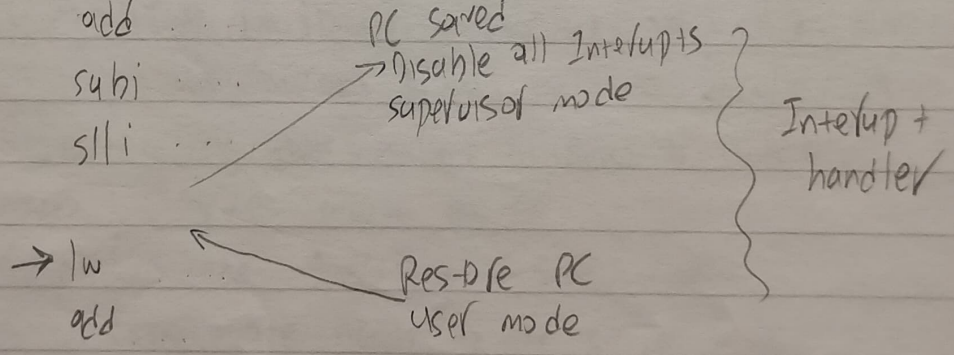
• Stall if condition evaluate late

Types of Interupts / Exceptions

- I/O device request
- Invoking an operating system service from a user program
- Breakpoint
- Page fault
- Integer arithmetic overflow
- Memory Protection violation

Precise Interrupts / Exceptions

if : 1) all intructions before this instruction has complete
2) none of the instruction after has modified
=> Precise interrupt
=> We can restar the execution from the interrupt point and still get the correct results.



Imprecise Interrupts / Exceptions

if the process does not look exactly as if the instructions were executed sequentially
Two possibilities: 1) An instruction may completed that are later in program order
2) An instruction may not yet completed that are earlier in program order

Precise interrupt point requires multiple PCs when they are delayed branches

```
    add  ...
    sub
PC  bne  r1, there  ⟸ Interrupt describe as <PC, PC+4>
PC+4 and  ...
     <other>
```

```
    add
    sub
PC  bne  r1, there
PC+4 and                ⟸ Interrupt describe as <PC+4, there>  taken
     <other>                                 or <PC+4, PC+8>  not taken
```

Why we need precise interrupts

- Several interrupts need to be restartable
  eg. TLB faults. fix translation and then restart faulting load/store
- Restartability does not require preciseness. However, preciseness makes restarts much easier
- Simplifies the Operating System (OS)
  - Less state needs to be saved away
  - Quick to restart

How to guarantee precise exceptions

- We can have exception in different stages of the pipeline
  eg - Arithmetic exceptions occur in execution stage
  - TLB faults can occur in fetch or memory stage

⇒ How to guarantee precise exception ?

- Use the pipeline
  - Each Instruction has an exception status field
  - Keep the PCs for every instruction in the pipeline
  - Check the exception when the instruction reaches the WB stage
- When an instruction reach the WB stage and has an exception
  - Save PC ⇒ EPC, Interrupt vector addr ⇒ PC
  - Convert all fetched instructions to NOPs
- It works because of in-order completion WB

Precise Interrupts and Speculation

- During the issue stage of instructions we operate as if us we are predicting that all previus instructions do not generate instruction
  - Branch prediction, data prediction
  - If we speculate wrong, need to back up and restart execution to the point at which we predicted incorrectly
  - Exactly the same with precise exceptions

⇒ Use ROB

Idea behind ROB

Keep the instructions in a FIFO, with the exact order that they are issued
- Each ROB entry contains PC, dest reg, result, exception status.

- When an instruction completes execution then the results are placed in the allocated entry in the ROB
  - Supplies operands to other instruction between execution complete & commit ⇒ more registers like RS
  - Tag results with ROB number instead of RS

- The instructions change the machine state at <u>commit stage</u> <u>not on the WB</u> ⇒ in-order commit ⇒ values at head of ROB are placed in registers
  ⇒ We can cancel/squash speculatively executed instructions during mispredicted branches or exception

Four stages of Tomasulo with ROB

1) Issue: Get Instruction from Op Queue
   - if there are free reservation stations and ROB slot, issue instruction & send operands & ROB no. for destination (dispatch)
2) Execution: Execute Instr. in the Execution Unit
   - When the values of the 2 source regs are ready then execute instructions, otherwise watch CDB for result. When both in RS execute. Check RAW problem
3) Write Result: End of execution (WB)
   - Write on CDB to all awaiting FUs & ROB. Mark RS available
4) Commit: Update the dest. Reg with the value from the ROB
   - When instr. at head of ROB & result present, update register file with the result (or store to memory) and remove instr. from ROB. Misspredicted branches or exception, flushes ROB

Memory Disambiguation

WAW/WAR hazards:

like hazards in Register file, we must avoid hazards through memory

- WAR/WAW are eliminated because the actual updating in memory occurs in order. When a store is at the head of the ROB, no earlier loads or stores can still be pending

RAW hazards:

```
st   0(R2), R5
ld   R6, 0(R3)
```

What if 0(R2) == 0(R3) => RAW => stall

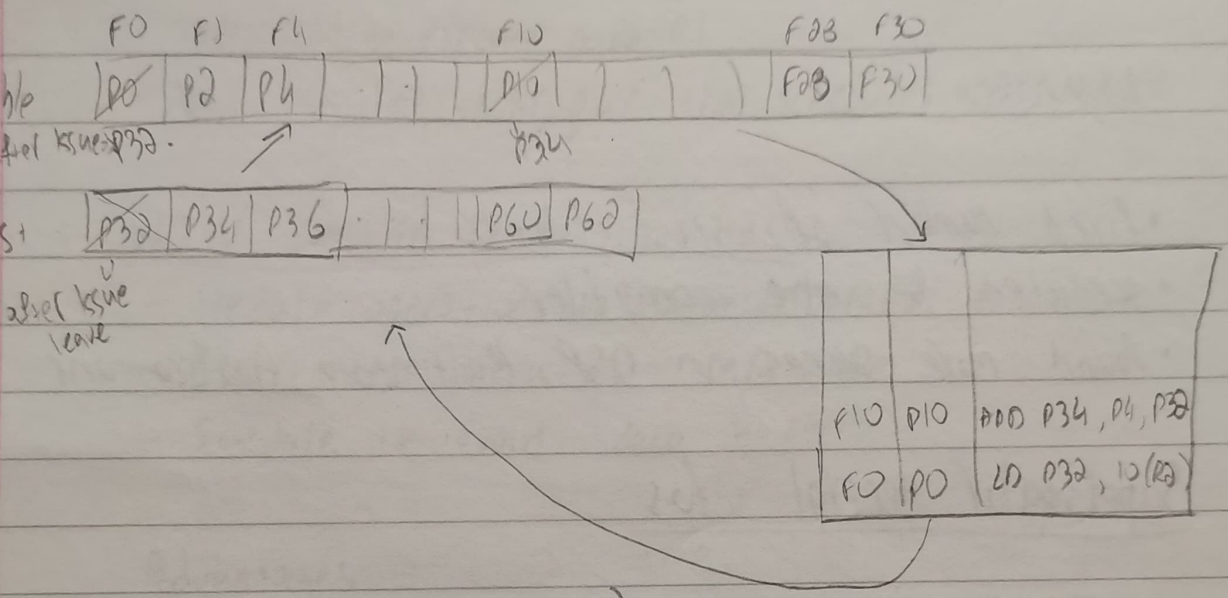
Solution: We might guess at whatever or not they are dependent (dependence speculation) and use ROB to fixup if we are wrong

⇒ HW support for memory disambiguation

- Store buffer: keeps all pending stores in-order
  - Keep track of address and value (when they are available)
  - FIFO ordering: will retire stores in program order
- When issuing a load
  - If any store prior to load is waiting for its address => stall load
  - If load addr matches earlier store addr (associative lookup) then we have memory induced RAW
    - If store value is available => return value
    - If store value is not available => return ROB number of source
  - Otherwise, sent out request to memory
- Stores commit in-order => NO WAR, NO WAW hazards

Explicit register renaming

Upon issue, every instruction that write a register allocates a new physical register from the freelist



e.g. Issue LD F0, 10(R2)

e.g. Issue ADD F10, F4, F0

BUT: what happens with Branches

There is HW feature to save the curr. Map Table and the free list. So we have a checkpoint, if we make wrong speculation to go back in previus state

Superscalar

- Varying multiple instr/cycle, 1-way, 2-way ... 8-way
- Scedule by compiler (statically scheduled) or by HW (dynamically sceduled)

VLIW

- fixed number of instr. (4-16)
- sceduled by the compiler
- found more succes in DSP, Multimedia Applications

Superscalar general infos

eg. Superscalar DLX: 2 instr., 1 Integer, 1 FP
⇒ fetch 64-bits: Integer + FP
⇒ Can only issue FP if there is Integer Instr.
⇒ More ports to FP registers to do FP load & FP op in pair

Advantage: Increase the number of instr. which can be in the pipeline at one time and hence the level of parallelism

Disadvantage: The larger number of instr. in pipeline, increase the potential for data & control depedencies ⇒ stalls

Advantage Vs VLIW: The ability to extract some parallelism from less structure code, and easily cache all forms of data

Challenges: 1) While Inte/FP split is simple for the HW but get CPI 0,5 only if: - 50% FP operation
- No hazards

2) If more Inst. issue at the same time ⇒ grater difficulty of decode & issue

VLIW general Infos

Tradeoff instruction space for simple decoding:

- The long instruction word has room for many operators
- All operations the compiler puts in the long instruction word are independent → execute parallel
- Need compiling technique that schedules across several branches

VLIW compiler Responsibilities:

- Schedule operations to maximize parallel execution
- Guarantees intra-inst. parallelism
- Schedule to avoid data hazards

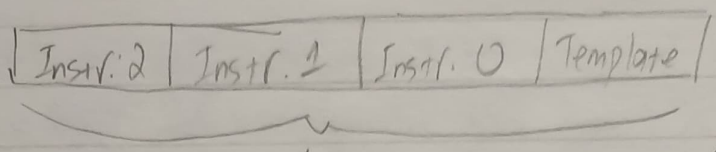
Advantages

- Compiler prepares fixed packets of multiple operations that give the full plan of execution
  - Dependencies are determined by compiler and used to schedule according to function unit latencies
  - Function units are assigned by compiler and correspond to the position within the instr. packet
  - HW doesn't have to "rediscover" dependencies

Disadvantages

- Object-code compability: recompile all code for every machine
- Object-code size:
  - Instr. padding wastes instr. memory
  - loop unrolling/software replicates code
- Scheduling variable latency memory operations
- Knowing branch probabilities: Profiling requires a significant extra step in build process
- Scheduling for statically unpredictable branches

IA-64 Instruction format

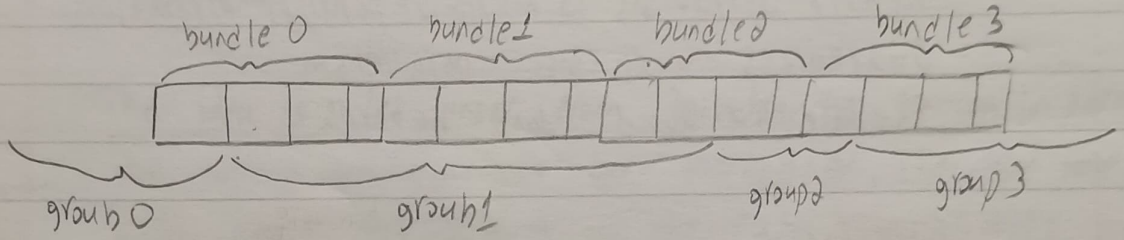


128-bit instruction bundle (41 * 3 + 5)

3 instr * 41 bit each

Template bits

- Template bits: describe the types of instructions and the grouping of these instructions with other in adjacent bundles



- Each group contains instructions that can execute in parallel and the boundary of a group (stop) is indicated by the template

IA-64 Predicated Execution

- Problem: Mispredicted branches limit ILP
- Solution: Eliminate hard to predict branches with predicated execution.

```
b0  Instr 1
    Instr 2
    br a==b, b2 if
```

```
b1  Instr 3      else
    Instr 4
    jmp b3
```

```
b2  Instr 5      then
    Instr 6
```

```
b3  Instr 7
    Instr 8
```

4 basic blocks

Predicated

```
Instr 1
Instr 2
p1 = (a!=b), p2 = (a==b)
(p1) Instr 3  || (p2) Instr 5
(p1) Instr 4  || (p2) Instr 6
Instr 7
Instr 8
```

1 basic block

Branch Predication

- Is an agressive compilation technique to generate code with higher degree of instruction level parallelism
- It lets operations from both branches of a conditional branch to be executed in parallel, to increase amount of parallel operation
- In this way branches are eliminated and replaced by conditional execution

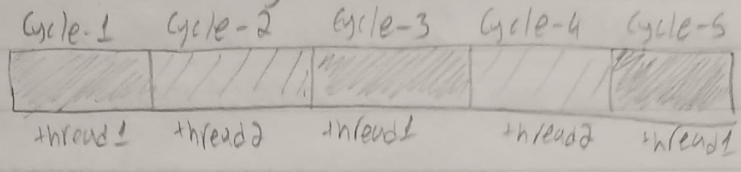
The idea is: let instr. from both branches go on in parallel, before the branch condition has been evaluated. The HW takes care that only those corresponding to the right branch will be finally commited

Multithreading Cost

- Each thread requires its own user state. Many CPU resources are split or shared
  - PC
  - Physical registers
  - Prefetch & Instr. buffers
  - Reorder buffer
  - Load/Store buffer
  - Issue buffers
- Also needs its own system state
  - virtual memory page table base register
  - exception handling registers

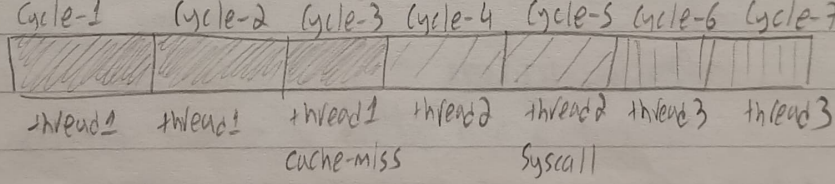
HW multithreading alternatives

- Fine-Grain Multithreading
  - Switches processor context every thread cycle
  - Context belong to same address space



Switch every thread cycle:
- Need fast HW switch between contexts
  - Multiple PCs are register files
- Implemented with round-robin scheduling, skipping stalled threads
- Hide both short and long stalls
- Delays all threads, even if they have no stalls

- Coarse-Grain Multithreading
  - Switches processor context upon long-latency event
  - Context may belong to different address space



Switches upon long-latency events
- Can afford slower context switch than fain-grain multithreading
- Threads are not slowed down
  - Other threads run when one have stall
- Pipeline start-up cost upon thread switching
  - Processor issues instructions from one thread

- Simultaneous Multithreading (SMT)
  - Techiques presented so far have all been "vertical" multithreading where each pipeline stage works on one thread at a time
  - SMT use fine-grain control but allow instr. from multiple threads to enter execution on same clock time.

Simultaneous Multithreading SMT

- Add multiple contexts and fetch engines and allow instr. fetched from different threads to issue simultaneously (superscalar)
- Utilize wide out-of-order superscalar processor issue queue to find instr. to issue from multiple threads
- OOO instr. window already has most of the circuitry required to schedule from multiple threads
- Any single thread can utilize whole machine
- Shared HW mechanism
  - Large set of virtual reg. can hold register sets of independent thread
  - Renaming provides unique register identifiers to different threads
  - Out-of-order completion of instructions from different threads allowed
  - No cross-thread RAW, WAW, WAR hazards
  - Separate re-order buffer per thread

† Page 21 Sigsgn Multithreading
† Page 26-38

SIMD: single instr. mutiple data

Vector Proccesors

- Vector is one dimension array
- A vector processor is one whose instr. operate on vectors rather than scalar (single data) values
- Basic Requirements
  - Need vector registers
  - Need vector length register (VLEN)
  - Need vector stride register (VSTR)
    - distance in memory between two elements in vector

- A vector instr. perform an operation on each element in consecutive cycles
  - Vector FUs are pipelined
  - Each pipeline stage operates on a different data element
- Vector instr. allow deeper pipeline
  - No HW interlock needed within vector
  - No control flow within a vector
  - Known stride allows easy address calculation for all vector elements
    - Enables prefetching of vectors into reg/cache/memory

Vector processor limitations

- Memory (bandwidth) can easily become a bottleneck, especialy if
  - Compute/memory operation balance is not maintained
  - data is not mapped appropriately to memory banks

Vector Registers

- Each vector data register holds N values of M-bits
- Vector control registers: VLEN, VSTR, VMASK
- VLEN is N
- VSTR is the distance in memory between two elements in vector
- VMASK indicates which elements of vector to operate on

  eg. $VMASK[i] = (V_k[i] == 0)$

Loading/Storing Vectors from/to Memory

- Requires loading/storing multiple elements
- Elements seperated from each other by a constant distance (stride)
- Elements can be loaded consecutive cycles if we can start the load of one element per cycle
  - ↳ We achieve this with Memory Banks

Vector Memory System

- Memory is divided into banks that can be access independently. Banks share address and data buses
- Next address = previus address + stride
- if (stride == 1)
  && (consecutive elements interleaved across banks)
  && (number of banks > bank latency)
  => we can sustain 1 element/cycle throughput

Vector Stripmining

- What if data elements > elements in vector register
  Eg. 527 data elements, 64-element VREG
  => 8 iteration where VLEN = 64
  + 1 iteration where VLEN = 15
  => Called vector stripmining

Gather/Scatter Operations

- What if vector data is not stored in a strided fashion in memory (irregular memory access to a vector)
  - Idea: Use indirection to combine/pack elements into VREG
  - Called: gather/scatter operations
- Vector loads/stores use an index vector which is added to the base register to generate the addresses

Eg.

Index Vector	Data Vector (to store)
0	3,14
2	6,50
6	71,20
7	2,5

Stored Vector (in Memory)	
Base + 0	3,14
base + 1	x
base+2	6,50
base+3	x
base+5	x
base+6	71,20
base+7	2,5

Conditional Operations in loop VMASK

- What if some operations should not be executed on a vector

  eg. `for(i=0; i<N; i++)`
  `if (a[i] != 0) then b[i] = a[i] * b[i];`

- Idea: VMASK
  - VMASK is a bit mask determining which data element should not be acted upon

```
VLD   V0 = A
VLD   V1 = B
VMASK = (V0 != 0)
VMUL  V1 = V0 * V1
VST   B = V1
```

- Another example

```
for(i=0; i<64; i++)
    if (a[i] >= b[i])
        c[i] = a[i]
    else
        c[i] = b[i]
```

A	B	VMASK
1	2	0
2	2	1
3	2	1
4	10	0
-5	-4	0
0	-3	1

⇒ Steps to execute this loop:

1) Compare A, B to get VMASK
2) Masked store of A into C
3) Complement VMASK (software/hardware)
4) Masked store of B into C

Some Issues

- Stride & Banking
  - As long as they relative prime to each other and there are enough banks to cover bank acess latency, we can sustain 1 element/cycle

- Storage of Matrics
  - Raw Major & Column Major
  - We need diferent stride to acces a row vs column

⇒ How to optimize
- More banks
- Better data layout to match the access patter
- Better maping of address to bank

Vector Summary

- Good to exploit regular data-level parallelism
  - Same operation performed on many data elements
  - Improve performance, simplify design (no intra-vector dependencies)
- Performance improvement limited by vectorizability of code

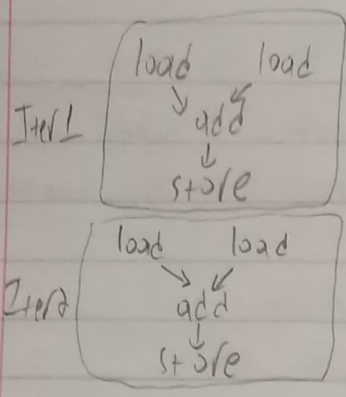
Masked Vector Instr

- Simple Implementation
  execute all N operations, turn off result writeback according to Mask
- Density-Time Implementation
  scan mask vector and only execute elements with non-zero Mask

⇒ Ο δεύτερος τρόπος κάνει περισσότερους ελέγχους, αφού ελέγχει όλο το vector για να δει το Mask αν είναι 1 και μετά κάνει πράξη, αλλά αν τα 1 είναι σχετικά λίγα συμφέρει. Γενικά ο δεύτερος χρησιμοποιείται αφού τα 1 θα προχωρ/χωρά τις περισσότερες φορές δεν περιμένουμε να εκτελέσουν

GPUs

How can you exploit parallelism here




```
for(i=0; i<N; i++){
  C[i] = A[i] + B[i];
}
```

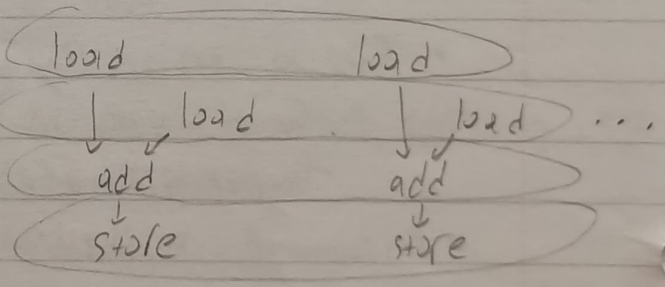
3 ways:
1) Sequential (SISD)
2) Data-parallel (SIMD)
3) Multi-threaded (MIMD/SPMD)

1) Sequential (SISD)

- Pipelined processor
- Out-of-order execution
- Superscalar processor

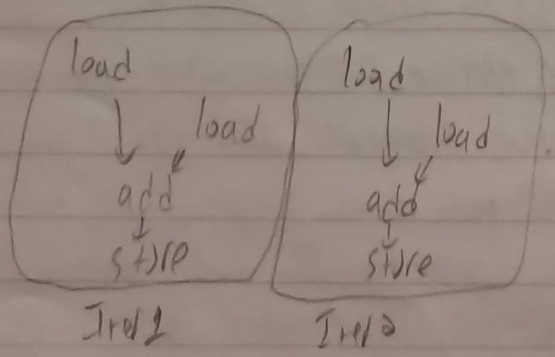
2) Data parallel (SIMD)

- vector processor
- VLD A → V1
  VLD B → V2
  VADD V1+V2 → V3
  VST V3 → C



Programmer or compiler generates a SIMD instr to execute the same instr from all iterations across different data

3) Multi-threaded



Programmer or compiler generated a thread to execute each iteration. Each thread does the same thing (but on different data)

Warp

- A set of threads using the same instruction are dynamically grouped into a warp (eg. execute at the same PC but on different data)

- Fine-Grained Multithreading
  - One instr. per thread in pipeline at a time
  - Interleave warp execution to hide latencies
- Register values of all thread stay in register file
- FGMT enables long latency tolerance (overxn)

Warp-based SIMD vs Traditional SIMD

- Traditional SIMD contains a single thread
  - Sequential instr. execution (lock-step operations in a SIMD instr.)
  - Programming model is SIMD (no extra threads) → SW needs to know VLEN
  - ISA contains vector/SIMD instr.

- Warp-based SIMD consists of multiple scalar threads executing in a SIMD manner (same instr. executed by all threads)
  - Does not have to be lock step
  - Each thread can be treated individually (placed on diff. warp)
    → Programming model not SIMD
    - SW does not need to know vector length
    - Enables multithreading and flexible dynamic grouping of threads
  - ISA is scalar → SIMD operations can be formed dynamically
  - Essentially, is is SPMD programming model implemented on SIMD hardware

SPMD

multiple instr. streams execute the same program

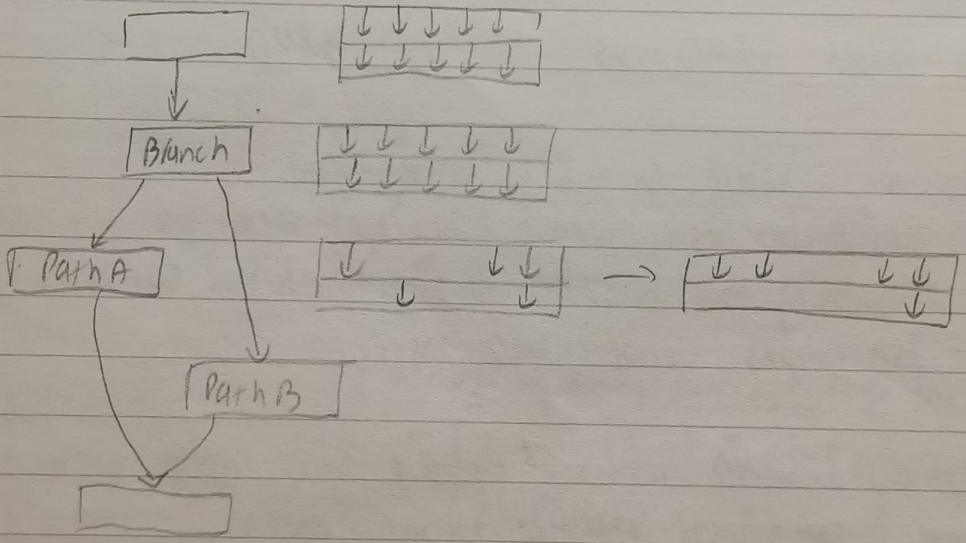
- Each program 1) works on different data 2) can execute a different control flow-path at run-time

Two major advantages of SIMT

1) Can treat each thread separately (can execute each thread independently on any type of scalar pipeline → MIMD processing
2) Can group threads into warps flexibly (can group threads that are supposed to truly execute the same instr. → dynamically obtain and maximize benefits of SIMD processing)

Dynamic warp formation / merging

Idea: Dynamically merge threads executing the same instr. (after branch)



Large warps & Two-level warp scheduling

- Two reasons for GPU resources be underutilized
  - Branch divergence $\xrightarrow{fix}$ Large warps
  - long latency operations $\xrightarrow{fix}$ two-level warp scheduling

Cache Measures

AMAT = Hit time + Miss Rate * Miss Penalty

CPU time = (CPU execution clock cycles + Memory stall cycles) * Clock cycle time

⇒ CPU time = IC * (CPI + $\frac{\text{memory stall cycles}}{\text{Instruction}}$) * Clock cycle time

⇒ CPU time = IC * (CPI + Miss_Rate * $\frac{\text{memory accesses}}{\text{instruction}}$ * Miss_Penalty) * Clock cycle ti

example: - CPI = 1

- Data accesses are only load & store (explain 50% of instr)
- Miss Penalty: 25 cc
- Miss Rate: 2 %

Memory Stall = IC * Miss_Rate * $\frac{\text{memory accesses}}{\text{instr.}}$ * miss-penalty

= IC * 0,02 * (1 + 0,5) * 25

= IC * 0,75

CPU time = (IC * (CPI + Memory_Stall)) * clock cycles

= (IC * (1 + 0,75)) * clock cycles

= IC * 1,75 * clock cycles

Direct Mapped

Pros: - Simple, low complexity, low power consumption
- Fast hit time
- Data available before cache determines hit or miss
  Hit/Miss check done in parallel with data retrieval

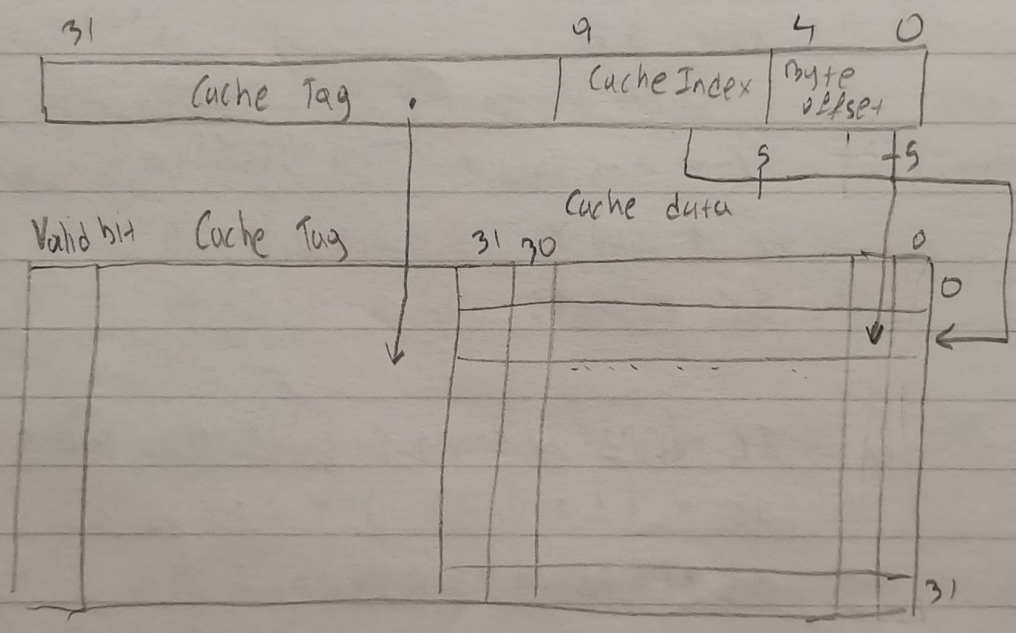
Cons: - Conflicts between blocks mapped to same block in cache

Two-way associative cache

Pros: - Choise of mapping memory block to different cache blocks in a set
- Reduction of conflicts

Cons - Increase complexity - comparators, multiplexor, parallel tag comparison
- Increase power consumption
- Increase hit time, due to comparators and multiplexor
- Data available after cache determines hit or miss

How is block found in the cache



Which Block is replaced on a miss

- Direct-Mapped => only 1 block to choose
- Set-associative cache
  - → Random
  - → LRU
  - → FIFO

what happens on a write

	write-through	write-back
Police	Data word written to cache block, is also written to next lower level memory	write new data word only to 1 cache block Update lower level just before a written block leaves cache, so not lose true value
Debugging	Easier	harder
Can read misses force writes	No	Yes (used to slow some reads, now write-buffer)
Do repeat writes to each lower level	Yes, memory busier	No

write buffers - hold address & data

- An entry may contain multiple words
- If there is an empty entry, the data and address are written to the buffer, and the CPU is done with the write
- if buffer contains other modified blocks, check to see if new address matches one already in buffer - if so combine the new data with that entry
- if buffer full and no address match, cache and CPU wait for an empty entry to appear (meaning some entry has been written to main memory)
- Merging improves memory efficiency, since multi-word writes usually faster than one word at a time

Reduce Miss Penalty

1) Multi-level caches

- Motivation
  - → Bigger caches bridge gap between CPU & DRAM

Performance Analysis:

AMAT = Hit Time L1 + Miss Rate L1 * Miss Penalty L1

Miss Penalty L1 = Hit Time L2 + Miss Rate L2 * Miss Penalty L2

- Local vs Global Miss rate
  - → local miss rate: Number of misses in the cache devided by the number of access to this cache
  - → global miss rate: Number of misses in the cache devide by the number of accesses issued from the CPU

Performance Analysis:

Average Memory Stall per Instruction = Miss per instruction L1 * hit Time L2 + Miss per Instr. L2 * Miss Penalty L2

example:
- Write-back first level cache
- 40 L1 misses per 1000 Instr.
- 20 L2 misses per 1000 Instr.
- L2 cache miss penalty = 100 cycles
- L1 hit time = 1 cycle
- L2 cache hit time = 10 cycles
- 1,5 memory references per instr.

$\text{Miss Rate L1} = \frac{40}{1000} = 0,04$

$\text{Miss Rate L2} = \frac{20}{40} = 0,5$

AMAT = Hit Time L1 + Miss Rate L1 * (Hit Time L2 + Miss Rate L2 * Miss Penalty L2)

$= 1 + 0,04\ (10 + 0,5 * 100) = 3,4$ cycles

Average memory stall per instr = Miss per instr L1 * hit Time L2 + Miss pen Instr L2 * Miss Penalty L2

$= \frac{40}{1000} \cdot 1,5 \cdot 10 + \frac{20}{1000} \cdot 1,5 \cdot 100 = 3,6$ cycles

Inclusion Property

- Multi-level inlusion => L2 data always present in L2, L2 in L3
  - → Simplifies cache consistency check in multiprocessors
  - → Complicates the use of different block size for L1 and L2
  - second-level cache miss must invalidate all first-level blocks

- Multi-level exclusion => L1 data never present in L2, L2 in L3...
  - → Reasonable choise for systems with small L2 cache relative to L1 cache
  - → Effective expansion of total caching space with a slower cache

2) Critical word first & Early restart

- Critical word first
  - → Cache block size tends to increase to exploit spatial locality
  - → Any given reference needs only one word from a multi-word block
  - → CWF fetches requested word first and sends it to processor
  - → Processor continues execution while rest of the block is fetched
- Early restart
  - → Fetches words in the order stored in the block
  - → As soon as critical word arrives, sends to processor and processor restarts

3) Giving priority to read misses over writes

- Write-through caches:
  - → On read miss check contents of write buffer
  - → If no conflict then let missing read bypass pending write
  - all dekstop & selvels give reads priority over writes
- Write-back caches:
  - → slow path: write dirty block to memory, then fetch a new block from memory to cache
  - → fast path: write dirty block to write buffer, then fetch a new block from memory to cache, then write back dirty block

4) Write Buffer & Victim Cache

- **Write buffer organization**
  - → Processor blocks on write if write buffer full
  - → Processor checks write address with address in write buffer
  - → Processor merges writes to same address if address is present in write buffer

ex. Assume WB with 4 entries, 64-bit word each

WB with no merging

Write address
100
108
116
124

V		V		V		V	
1	Mem[100]	0		0		0	
1	Mem[108]	0		0		0	
1	Mem[116]	0		0		0	
1	Mem[124]	0		0		0	

WB with merging

Write addresses
100
108
116
124

V		V		V		V	
1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
0		0		0		0	
0		0		0		0	
0		0		0		0	

- **Victim cache**

Tiny caches holds evicted cache blocks
- → Small (eg 4 entries) fully associative buffer for evicted blocks
- → Propose to reduce impact of conflicts on direct-mapped caches

Victim cache + direct-mapped cache ≈ associative cache

Όσα διώχνονται από την direct-mapped cache πάνε στην victim cache γιατί μπορεί να χρειαστεί ξανά σύντομα

5) Non-blocking or Lockup free caches

Idea: → Allow for hits while serving a miss (hit-under-miss)
→ Allow not the one outstanding miss (miss-under-miss)

Make sense: → when the processor can handle > 1 pending load/store
superscalar processor
→ when the cache serve > 1 processor or other cache
→ when the lower level allows for multiple pending accesses

Difficulties: → Handling multiple misses at the same time
→ Handing loads to pending misses
→ Handling stores to pending misses

- Miss status handling register
  - Keep track of
    → Outstanding cache misses
    → Pending load & stores that refer to that cache block
  - Fields of MSHR
    → Valid bit
    → Cache block address
    → Issued bit (1 if already request issued to memory)
    → for each pending load or store
      → Valid bit
      → Type (load/store) & Format (byte/halfword...)
      → Block offset
      → Destination reg for load or store buffer entry

1	27	1
Valid	Block address	Issued

1	3	5	5	
Valid	Type	Block offset	Destination	load/store 0
				load/store 1
				⋮

- Non-blocking caches: operations

  - On cache-miss:
    - → Search MSHRs for pending access to same cache block
      - if yes, just allocate new load/store entry
    - → (if no) Allocate free MSHR
      - update block address and first load/store entry
    - → if no MSHR or load/store entry free, stall

  - when one word for cache line become available
    - → check which load/store are waiting for it
      - forward data to LSU
      - mark load/store as invalid
    - → write word in the cache
  - when last word for cache line is available
    - → Mark MSHR is invalid

6) Multi-ported caches

Idea: → Allow for multiple accesses in parallel

Implemented with: → True multiporting

2 ways → Multiple banks

difficulties: → Interaction with parallel accesses (especially stores)

- True multiporting: → Use 2-ported tag storage
  - → Problem: → large area increase
    - → Hit time increase
- Multiple banks: → Partition address space into multiple banks
  - → Benefits: → accesse can go in parallel if no conflicts
  - → challenges: → conflicts
    - → distribution network
    - → bank utilization

Reduce Miss Rate

3 C's model

Characterization of cache misses:

→ Compulsory miss: Miss that happens due to the first access to a block since program began. Cold-start miss

→ Capacity miss: Miss that happens because a block that has been fetched in the cache needed to be replaced due to limited capacity. Block has been fetched, replaced and re-fetched

→ Conflict miss: Miss that happens because address of block maps to some location in the cache with other block in memory. Block has been fetched, replaced, re-fetched and cache has invalid locations that could hold the block if a different address mapping scheme were used

1) Increase the block size

- Spatial locality
  - → Larger block size usually reduce compulsory misses
  - → Increase miss penalty, since processor need to fetch more data
  - → May increase conflict misses, if spatial locality is poor (most words in fetched not used)
  - → May increase capacity misses, if spatial locality is poor (most words in fetched not used)

2) Larger Caches

Pros: → Reduction of capacity misses

Cons: → longer hit time
→ increase area, power and cost

3) Increase Associativity

→ Higher associativity increase hit time
⇒ increase cycle time

4) Prefetching

Idea: fetch data to cache before processor request them
→ can address cold misses
→ can be done by programmer, compiler, hardware

Characteristics of ideal prefetching

→ You only prefetch data that are truly needed
→ You issue prefetch request early enough
→ You don't issue prefetch request too early

- SW prefetching
  Issues: → Take up issue slots
  → Take up system bandwidth
  → Must have non-blocking caches
  → Prefetch distance depends on specific system implementation
  → Not easy to use for pointer base structures

- HW prefetching

Same goals as SW prefetching
Major Questions: → where to place a prefetch engine
→ what to prefetch
→ when to prefetch
→ where to place prefetched data

- Simple sequential prefetching
  - On cache miss, fetch N sequential memory blocks
    - → Exploit spatial locality in both instr & data
    - → Exploit high bandwidth for sequential accesses
  - Called "Spatial Prefetch"
  - Extend to N sequential memory blocks
  - Problem: slow down once every N cache lines
- Steam prefetching

→ Is a continuous version of prefetching
- → stream buffer can fit N cache lines
- → On miss, start fetching N sequential cache lines
- → on hit, move cache line to cache, start fetching $N+1$

→ In other words tries to stay N cache lines ahead

- Stride prefetching

- Idea: Detect & prefetch strided accesses

- Stride detection using a PC-based table
  - → for each PC remember stride
  - → stride detection:
    - → remember the last address used for this PC
    - → compare to currently used address for this PC
  - → track confidence using a two bit saturating counter

5) Compiler Optimizations

- Code transformation to improve:
  - → Spatial locality, through higher utilization of fetched cache blocks
  - → Temporal locality, through reduction of the reuse

distance of cache blocks
→ Examples: loop interchange, loop blocking, loop fusion

- Data layout and data structure transformations to improve
  → spatial locality, through higher utilization of fetched cache blocks
  → Examples: array merging, structure/object class member reordering in memory

Reduce Hit time

- small & simple caches
- virtually addressed caches
- pipelined caches
- trace caches