UNIVERSITÀ DEGLI STUDI DI TRENTO

Advanced Computing Architectures

Passerone Roberto

Notes curated by:
Sartori Alessandro and Zanella Davide

Contents

Chapter 1

Fundamentals of Computer Design

1.1 Power Consumption Evaluation

In CMOS technology, power consumption is absorbed passively and actively. In the former case, power absorption is a symptom of inevitable current leakages between transistors' junctions, while in the latter case, current is drawn by logic gates when charging and discharging during operation and switching which, as a matter of fact, also causes transient short circuits, i.e. the control signal has finite slope and the N-MOS and P-MOS transistors won't switch perfectly in sync.

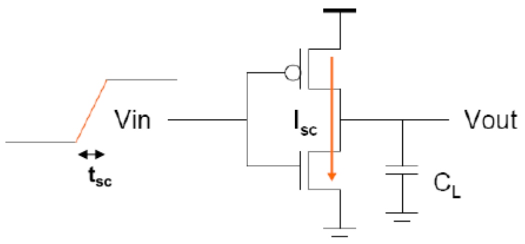


Figure 1.1: Representation of a Logic Gate during a transient short-circuit: both transistors are conducting, creating a direct path to ground. C_L is a parasitic capacitance responsible for most of the active energy consumption.

Active (or Dynamic) power consumption of a transition (E_t) is calculated by multiplying the energy necessary to switch state times the probability of this event (dependent for instance on the architecture). Power, being energy over time, equals energy times clock frequency.

$$E_t = C_L \cdot V_{DD}^2 \cdot P_{0\to1}$$
$$\text{Power} = C_L \cdot V_{DD}^2 \cdot P_{0\to1} \cdot f_{0\to1} = E_t \cdot \text{Frequency}$$

Additionally, one needs to account for the mentioned short-circuits, absorbing a current proportional to the rising time of the signal t_{sc}:

$$\begin{aligned} E_{sc} &= t_{SC} \cdot V_{DD} \cdot I_{peak} \cdot P_{0\to 1} \\ \text{Power} &= t_{sc} \cdot V_{DD} \cdot I_{peak} \cdot P_{0\to 1} \cdot f_{0\to 1} = E_{sc} \cdot f_{0\to 1} \end{aligned}$$

Overall, dissipated power is given by the combination of:

- E_t, mainly dependent on parasitic capacitance, therefore on number of gates and transistor size
- E_{sc}, dependent on transistor technology, size, temperature, and signal slope
- $E_{leakage} = V_{DD} \cdot I_{leakage}$, increasing with higher temperatures and lower threshold voltages
- Frequency, dependent on performance
- $P_{0\to 1}$, dependent on signal statistics

Therefore:

$$\begin{aligned} \text{Power} = {} & C_L \cdot V_{DD}^2 \cdot P_{0\to 1} \cdot f_{0\to 1} \\ & + t_{sc} \cdot V_{DD} \cdot I_{peak} \cdot P_{0\to 1} \cdot f_{0\to 1} \\ & + V_{DD} \cdot I_{leakage} \end{aligned}$$

1.2 Cost Estimation

Costs tend to generally decrease due to a number of factors. Manufacturing yield Y for example, increases even without improvements in the implementation technology, adding up to an increase in production volumes which result in more amortized manufacturing costs. Costs trends also decrease because of competition effects, requiring selling prices to be close to production ones.

Experimentally, the cost C of an integrated circuit is given by:

$$C = \frac{C_{die} + C_{testing} + C_{packaging}}{Y}$$

where C_{die} is the cost of manufacturing a die, $C_{testing}$ that of testing a die after cutting it from a wafer, and $C_{packaging}$ that of packaging and testing the working ones. Unlike these last two costs which solely depend on the complexity of the IC, C_{die} depends on more factors:

$$C_{die} = \frac{C_{wafer}}{N_{die} \cdot Y_{die}}$$

Where N_{die}, in turn, equals the number of dies on a round wafer of diameter D, minus those close to edge:

$$N_{die} = \frac{\text{Area of wafer}}{\text{Area of die}} - \text{Chips near the edge} = \frac{\pi \cdot \frac{1}{4} D^2}{A_{die}} - \frac{\pi \cdot D}{\sqrt{2 \cdot A_{die}}}$$

And Y_{die} is given by:

$$Y_{die} = Y_{wafer} \cdot \left(1 + \frac{N_{defects} \cdot A_{die}}{\alpha}\right)^{-\alpha}$$

with $N_{defects}$ being the number of defects per unit area of the wafer, and α an empirical value related to manufacturing complexity.

In conclusion, it can be inferred that a designer only has control on the strong dependence that A_{die} has on costs, while all other parameters only depend on the manufacturing process.

1.3 Performance Evaluation

Measuring performance is a far from trivial task. Although Moore's Law states that clock frequencies double every 15 to 18 months, architectural enhancements need to be considered as well, therefore leading to the lack of a univocal comparison criteria. An ideal metric would would be:

- **Linear**: if two values differ by a certain factor, performances will also differ by the same factor.
- **Reliable**: a system X outperforms another system Y if the metric indicates so.
- **Repeatable**: measurements results are consistent across repetitions, with a given amount of uncertainty.
- **Easy to use**: more people will adopt the metric and the probability of obtaining incorrect results decreases.
- **Consistent**: units and definition are the same across different systems and configurations.
- **Independent**: the definition is univocal and is not affected by external influences.

Example 1. ***Clock Frequency***
This metric is repeatable, easy to measure, consistent and independent, but NOT linear and reliable. For instance, it doesn't take into account the amount of computation carried out within a clock cycle, nor it considers the interaction with the I/O system.

Example 2. ***Millions of Instructions Per Second (MIPS)***
Although repeatable, independent, and easy to measure, this metric is NOT linear, reliable, nor consistent: as for the Clock Frequency, the meaning of "instruction" and its computational load vary from architecture to architecture.

Example 3. ***MOPS and MFLOPS***
These two metrics try to solve the inconsistency of MIPS by considering (respectively) only integer and floating-point operations, therefore defining more strictly the concept of "instruction". However, it still doesn't consider the differences in implementation of the operations, possibly resulting in misleading figures.

Example 4. ***Execution Time***
*Execution Time meets all the properties listed above. It includes both the time actually spent running (CPU time), and the time spent waiting for memory and I/O operations. CPU time, in turn, consists of **User CPU time** and **System CPU time**, where the former counts the time spent when executing the task's instructions, while the latter counts the time spent for OS calls.*

$$CPUtime = N_{CLK} \cdot T_{CLK} = CPI \cdot IC \cdot T_{CLK}$$

Where N_{CLK} is the number of clock cycles, T_CLK the period of a cycle, CPI stands for Clock Per Instruction and describes the average duration of a single instruction, and IC (Instruction Count) the number of instructions of the task.

$$CPI = \frac{\sum_{i=1}^{n} IC_i \cdot CPI_i}{IC}$$

Where n is the total number of instructions of the ISA of a certain processor, and IC_i and CPI_i are respectively the IC and CPI of the $i - th$ instruction.

1.4 Benchmarks

A benchmark is a suite of tasks that aims at mimicking the expected workload of a given computing system, in order to compare the performance of different systems. Benchmarks come in many types:

- **Real Applications** have many problems related to portability and variability of the workload across repetitions due to the number of employed options.
- **Modified (Scripted) Applications** are Real Applications with improved portability thanks to, for instance, the removal or simulation of I/O operations.
- **Kernels** are key sections of real applications extracted to evaluate the performance of individual features.
- **Synthetic Benchmarks** are similar to kernels but are just simulations of instructions frequencies with no practical usefulness.
- **Toy Benchmarks** consist of 10 to 100 lines of code running simple and portable algorithms with known solutions.

Since companies tend to only select those which highlight the performance of their products, benchmarks are given out among suites with specific rules on compilation options and source code modifications. Depending on the target architecture, benchmarks can be of different kinds from Desktop Computing to Servers or Embedded Systems.

In order to evaluate the performance of a suite, two geometric means are used: one for integer benchmarks, and for floating-point ones. A geometric mean is calculated as:

$$G = \sqrt[N]{\prod_{j=1}^{N} \left(\frac{T_{e_j}}{T_{R_j}} \right)}$$

With N being the number of programs, T_{e_j} the execution time of the j-th task, and T_{R_j} the reference execution time of the j-th program. A geometric mean offers the advantage of maintaining consistent relationships regardless of the reference machine, but long and short tasks will have the same weight, encouraging engineers to focus on easy-to improve benchmarks instead of focusing on the slowest applications. Therefore, a number of other criteria exist:

- **Total Execution Time**

$$T = \sum_{j=1}^{N} T_{e_j}$$

- **Average Execution Time**

$$\overline{T}_e = \frac{1}{N} \sum_{j=1}^{N} T_{e_j}$$

- **Weighted Execution Time**

$$T_{e_w} = \sum_{j=1}^{N} w_j T_{e_j}$$

- **Equal Time Weighing**

$$w_j = \frac{1}{T_{e_j} \cdot \sum_{j=1}^{N} \frac{1}{T_{e_j}}}$$

1.5 Design of Computing Systems

Designing a system or architecture is a multi-variable and multi-objective optimization problem, involving *execution time (T)*, *cost (C)*, and *power consumption (P)*.

E basta, mi son rotto le balle di sto capitolo (slide 52).

Chapter 2

Instruction Set Architectures

Chapter 3

Pipelining

The basic idea of Pipelining is to divide the elementary operations related to a single instruction into a sequence of smaller task that can be performed in parallel.

The goal is to balance the length of each pipeline stage. In this way the time taken per instruction (pipelined) when the pipeline is full is:

Time per instruction (unpipelined) / Number of stages

Pipelining reduces the average CPI.

3.1 Ideal pipelining

$$t_s = t_o + T_i/p$$

where
t_o is the overhead latency of each stage;
p is the number of stages;
T_i is the total latency of a single instruction;

The **CPI** is the number of clocks required (on average) to complete one instruction after the previous one has completed.

In a pipeline, the throughput of any instruction sequence (i.e., the number of completed instruction per unit of time) is **about p times larger** than in the unpipelined solution. CPI is asymptotically equal to 1. Average Instruction Execution Time $T_e = T_{CLK} * CPI$

3.2 Pipeline in a RISC CPU (MIPS)

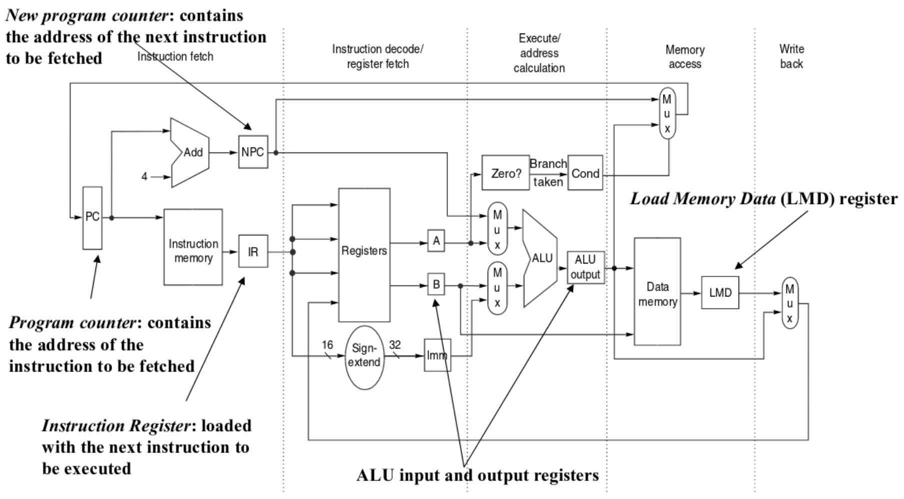
MIPS features

- 32 64-bit general purpose register (GPR), named R0, R1, ..., R31. (R0 = 0)
- 32 floating-point registers (FPR), named F0, F1, ..., F31.
- Data types: 8-bit bytes, 16-bit half words, 32-bit words, 64-bit double words.
- Addressing modes:

  - Immediate: ADD R4, #3
  - Displacement: LD R4, 100(R1)
  - Register indirect: LD R4, 0(R1)
  - Absolute addressing: LD R4, 2536(R0)

- Load/store architecture

3.2.1 Instruction Execution stages



New program counter: contains the address of the next instruction to be fetched
Instruction fetch
Instruction decode/ register fetch
Execute/ address calculation
Memory access
Write back
Add
4
NPC
PC
Instruction memory
IR
Registers
A
B
Sign-extend
16
32
Imm
Zero?
Branch taken
Cond
M u x
ALU
ALU output
Data memory
LMD
Load Memory Data (LMD) register
Program counter: contains the address of the instruction to be fetched
Instruction Register: loaded with the next instruction to be executed
ALU input and output registers

Stages:

- *Instruction fetch stage (IF)*:
  Send the program counter (PC) to memory and fetch the current instruction from memory or I-cache. Update the PC to next sequential value (e.g. by adding 4 if an instruction is 4 byte long)

- *Instruction Decode/ register fetch stage (ID)*:
  decode the instruction and perform simultaneous operand reading. Instruction decode and operand reading can be done in parallel because RISC instructions are fixed-field encoded.
  • In case of register operands read the source registers values from the register file
  • In case of immediate operands, sign-extend the offset field of the instruction

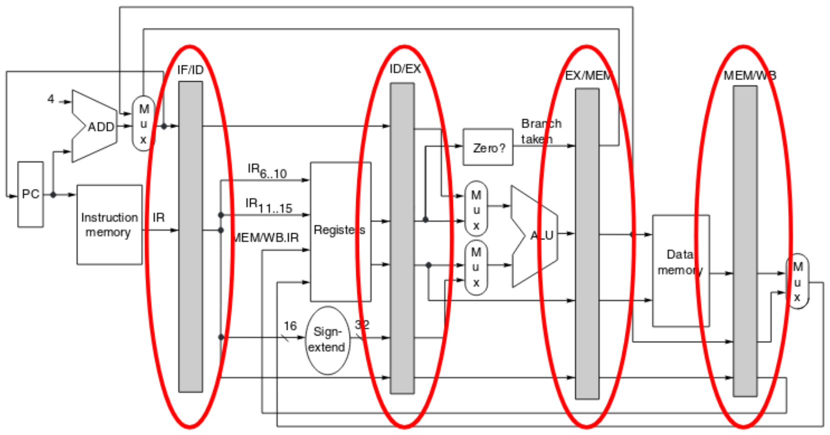
- *Execution/effective address stage (EX)*:
  The ALU operates on the operands prepared in the previous cycle, performing different operations depending on the instruction type.
  • Load/store instructions: The ALU adds the base register and the offset to form the effective address.
  • Register-Register ALU instructions: The ALU performs the operation specified in the ALU opcode on the values read from the register file
  • Register-Immediate ALU instructions:The ALU performs the operation specified by the ALU opcode on the value read from the specified register

and the sign-extended immediate
• Branch instructions: The ALU adds the next instruction value to the extended value of the address (multiplied by 4) to calculate the branch target address (BTA). In conditional branches the value of the condition register is compared with the reference value (i.e., 0) and stored into a special register.

- *Memory access stage(MEM)*:
  – Load instructions: The memory performs a read using the effective address computed in the previous cycle using A and Imm and saved in ALU output
  – Store instructions: The memory performs a write using the effective address computed in the previous cycle using A and Imm and saved in ALU output, while the value comes from the other register, saved in B
  – Branch instructions: in branches, the PC is replaced with the BTA (saved in ALU output) according to the outcome of the comparison with zero, saved in Cond

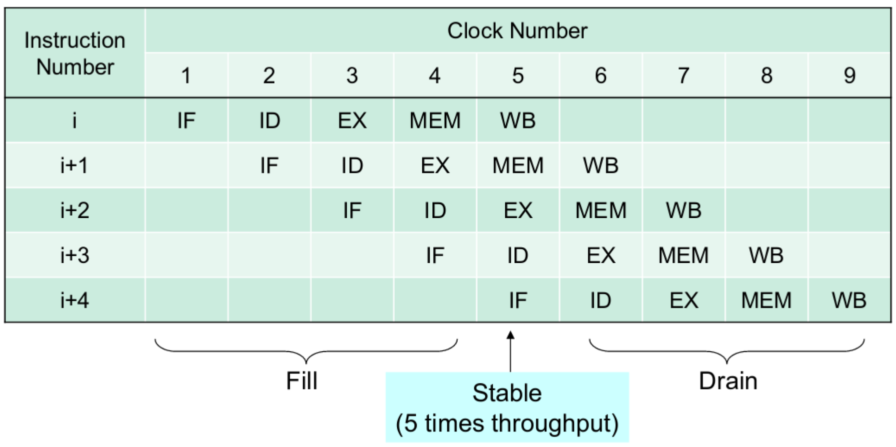
- *Write-back stage(WB)*:
  –Loads and ALU instructions: Write the result into the register file, whether it comes from the memory system (for a load) or from the ALU (for an ALU instruction)

It is important to **perform approximately the same amount of work in each stage**. In fact, the clock cycle of the processor depends on latency of the stage with the worst-case delay.



Instructions are fetched and initiated at every clock cycle filling all the stages. In this way the CPU will be perfectly busy. For this, the following basic modifications have to be applied:

- Insert homogeneous registers or latches between stages (IF/ID, ID/EX, EX/MEM, MEM/WB) thus replacing (i.e. including) temporary registers
- Propagate the control signals generated by the CU as well as the status bits of each instruction within the same pipeline registers
- Anticipate the PC contents update operation to the IF stage (instead of running it in the MEM stage) to avoid fetch latencies



Instruction Number	Clock Number								
	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	WB				
i+1		IF	ID	EX	MEM	WB			
i+2			IF	ID	EX	MEM	WB		
i+3				IF	ID	EX	MEM	WB	
i+4					IF	ID	EX	MEM	WB

Fill
Stable
(5 times throughput)
Drain

Why isn't the speed-up = 5? The unpipelined version skips certain stages when not needed and the pipelined version has a 20% clock overhead.

3.3 Hazards

The presence of several instructions simultaneously active into the pipeline may lead to various type of dependencies which turn into hazards. Hazards reduce the performance from the ideal speedup by forcing the pipeline to **stall**, i.e., to delay subsequent instructions until the source of the hazard is solved. Hazards can be solved using *hardware techniques*, *software techniques* or a combination of both.

3.3.1 Structural hazard

Arise from resource conflicts when the hardware can not support all possible combinations of instructions at the same time.

In a pipelined processor, the overlapped execution of instructions requires pipelining of functional units and duplication of resources. If some combination of instruction requires to use the same resource a structural hazard arise.

To avoid structural hazards (which would lead to incorrect results), the pipeline is stalled -¿ 1 or more “empty”cycles or bubbles are inserted in the pipeline.

Stalls result in a reduction of instruction throughput.

Other solutions:

- Hardware:
  Resource splitting/replication but it's expensive!

- Software:
  Instructions requesting the same resource are located at proper distance from each other. For instance, two conflicting instructions could be delayed by the compiler. It's effective but compiler design could be cumbersome!

Impact of hazards on CPI Say the probability of a stall is 20%: 1 stall every 5 instructions

$$CPI = \frac{\#clocks}{\#instr} = \frac{IC+(\frac{IC}{5}-1)}{IC} = 1 + \frac{1}{5} - \frac{1}{IC}$$

3.3.2 Data hazard

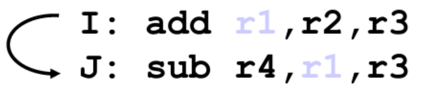
Data hazards are related to **data dependencies**, i.e., sequential constraints imposed by the data flow of the program. Two different instructions are data dependent if they have a common register or memory location; unless the operand is for both a *source operand*.

Data dependencies can occur in:

- **Straight-line code**: between subsequent instructions during the execution of a program segment
- **Loops**: between instructions belonging to different iterations of the loop (also called *recurrences* or *inter-iteration dependences*)
  Example: first-order linear recurrences: e.g.: X(i) = A(i) * X(i-1) + B(i)

Three kinds of data dependences:

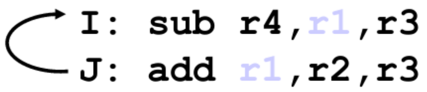
- **Read after Write (RAW)** also called flow or true dependence
  A read uses the value written by a write instruction.



```
I: add r1,r2,r3
J: sub r4,r1,r3
```

Hazard: Instruction J tries to read an operand **before** instruction I has written it.

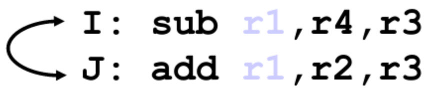
- **Write after Read (WAR)** also called anti dependence
  The destination of an instruction is a source operand of a previous instruction.



```
I: sub r4,r1,r3
J: add r1,r2,r3
```

Hazard: Instruction J writes operand **before** instruction I reads it. This results from reuse of the name “r1”.

- **Write after Write (WAW)** also called output dependence
  Two instructions have the same destination operand.



```
I: sub r1,r4,r3
J: add r1,r2,r3
```

Hazard: Instruction J writes operand **before** Instr I writes it. This also results from the reuse of name “r1”.

Solutions to data hazards

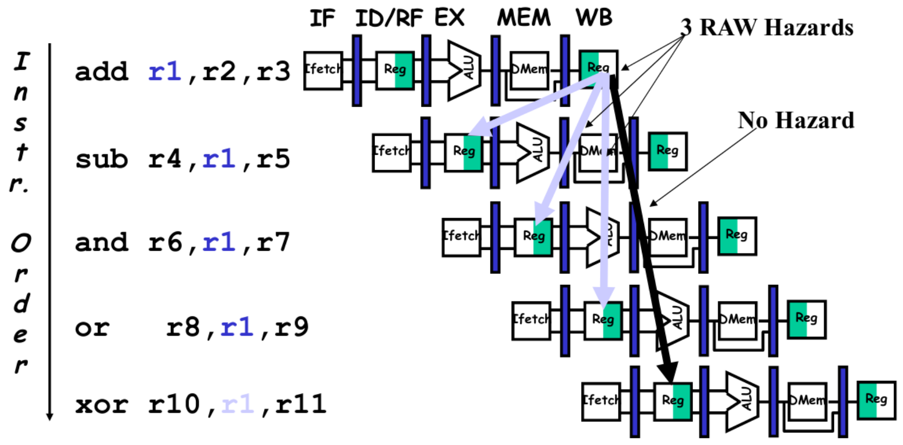
- WAR, WAW are **false dependences**: they can be solved through renaming (during compilation or by means of specialized hardware)

Example

i_k	*mul r4, r1, r2*	→	i_k	*mul r4, r1, r2*
i_{k+1}	*add r1, r5, r3*	→	i_{k+1}	*add r6, r5, r3*

- RAWs are **true dependences**

One RAW hazard example on a 5-stage pipeline:

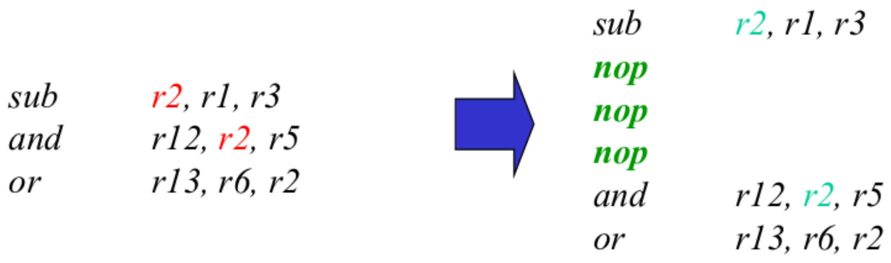


Software solutions

- *Code moving*:
  If possible, the compiler inserts as many non-dependent instructions as necessary (3 instructions in the case considered) between define and first use of a variable.

sub	*r2, r1, r3*	*sub*	*r2, r1, r3*
and	*r12, r2, r5*	***add***	***r24, r20, r20***
or	*r13, r6, r2*	***and***	***r7, r8, r9***
add	***r24, r20, r20***	***lw***	***r16, 100(r18)***
and	***r7, r8, r9***	*and*	*r12, r2, r5*
lw	***r16, 100(r18)***	*or*	*r13, r6, r2*

- *Nop stuffing*:
  The compiler inserts as many nop as are the required cycles between define and first use; So do not perform any active computation.



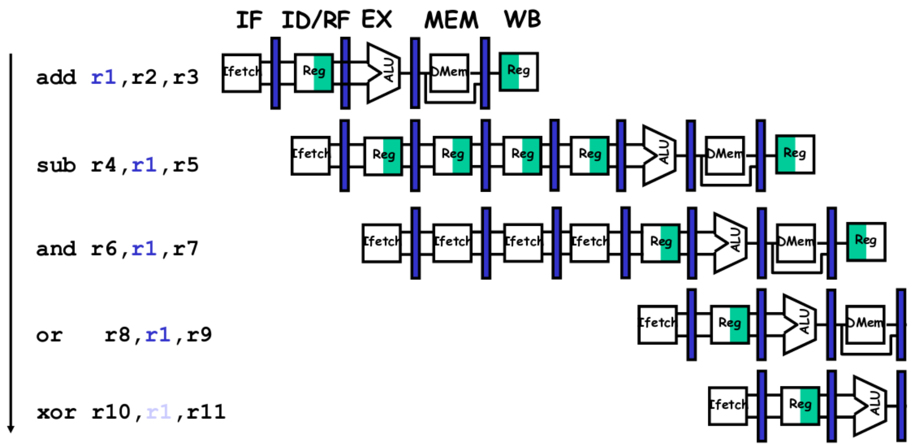
```
sub     r2, r1, r3
and     r12, r2, r5
or      r13, r6, r2
```

```
sub     r2, r1, r3
nop
nop
nop
and     r12, r2, r5
or      r13, r6, r2
```

Hardware solutions The CU of the pipeline:

1. Check (in the ID stage) whether one of the source register r_{s1} and r_{s2} of the considered instruction I_{k+1} coincides with the destination register r_d of some previous incomplete instruction. 6 comparisons are required:
$IF/ID.IR(r_{s1}) == ID/EX.IR(r_d) \| IF/ID.IR(r_{s2}) == ID/EX.IR(r_d)$
$IF/ID.IR(r_{s1}) == EX/MEM.IR(r_d) \| IF/ID.IR(r_{s2}) == EX/MEM.IR(r_d)$
$IF/ID.IR(r_{s1}) == MEM/WB.IR(r_d) \| IF/ID.IR(r_{s2}) == MEM/WB.IR(r_d)$

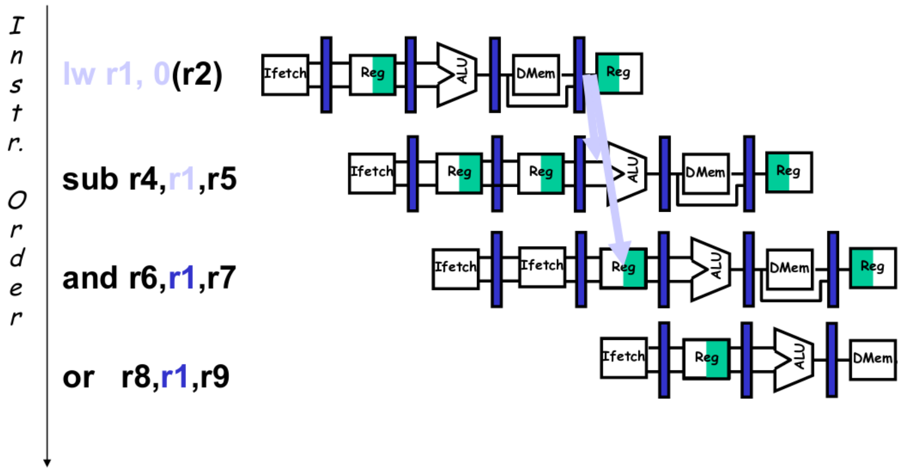
2. If a hazard is detected, the CU stalls the conflicting instruction I_{k+1} in the ID stage until the instruction I_k has completed its flow (pipeline interlock). Same as a nop but with less instruction memory space.



Forwarding We can use forwarding to reduce the stall penalty. Briefly in some cases instead of waiting that the information will be written in the registers we can simply forward them from a inter-stage pipeline register to another one.

Thus we can implement forwarding control by multiple comparisons between EX/MEM.IR(rd) and MEM/WB.IR(rd) with ID/EX.IR(rs) and EX/MEM.IR(rs). Increasing HW complexity due to comparators and MUX enlargement.

Not all RAW hazard can be handled by forwarding. This happens when certain data is required at ALU inputs before it is ready. In this case forwarding reduces stall penalty, but it can not prevent some "bubbles".



3.3.3 Control hazard

Control dependences are sequential constraints due to the control flow of the program. All branch instructions (e.g. jumps, procedure calls and returns, traps, interrupt) create control dependences.

If a branch changes the PC, it is referred to as a **taken branch**; if it falls through, it is not taken, or **untaken**.

Control hazards are related to the risk of fetching into the pipeline the wrong instructions following a conditional branch before knowing if the branch has to be taken or not. For example when we have a beq followed by some other instructions, without any control the following instructions will be fetched before the beq instruction enters in the MEM stage. There is a control hazard.

Solutions

1. *Freezeor flush the pipeline*:
   Hold or delete any instruction after the branch until the BTA is known

2. *Predict branch as not taken*:
   As PC+4 is already calculated, the hardware continues to fetch instructions in sequence as if the branch were not executed. If the branch is actually taken, wrongly fetched instructions have to be “flushed” (i.e., turned into nops)

3. *Predict Branch as Taken*:
   As soon as the branch is decoded and the BTA is calculated, we begin fetching instructions from BTA. If the branch was not to be taken, wrongly fetched instructions are “flushed” (like in solution 2).This solution is meaningful only if the BTA is known before branch outcome.

4. *Branch hoisting (delay slots)*:
   The compiler moves 3 independent instructions that have to be executed in any case immediately after the branch instruction into the so-called branch delay slots. When such instructions are complete the branch outcome is known and no stalls are required.

To reduce the control hazard time penalty, we can shift both the decision about taken or not taken and the BTA calculation in the ID stage. This requires an additional adder, but it decreases the possible stalls from 3 to 1. This is not a general rule. In deeply pipelined processors, the control hazard penalties could be very large in terms of wasted clock cycles and can not be solved so easily.

3.3.4 Performance of pipelines with stalls

$Speedup = \frac{CPI_{unpipelined}}{CPI_{pipelined}} * \frac{Tclk_{unpipelined}}{Tclk_{pipelined}}$
$CPI_{pipelined} = IdealCPI + N_{stalls} = 1 + N_{stalls}$

If we ignore the pipeline overhead time we have that: $Tclk_unpelined = Tclk_pipelined$
If all instructions take the same number of cycles: $CPI_{unpipelined} = pipeline - depth$

3.4 Exception

The term exceptions cover all the possible exceptional situations whereby the normal execution order of instructions is changed.

Many differences exist between various CPUs.

Some sources of exception

1. Conditions arising from instruction execution (e.g., overflow, FP anomaly)
2. External interrupts (e.g., I/O device request)
3. System calls (traps)
4. Memory-related exceptions (e.g., cache misses, page fault, memory protection violation, misaligned memory access)
5. Undefined Opcode6.Hardware malfunctions and power failures

Some features

- Synchronous vs. Asynchronous
- User requested vs. coerced
- User maskable vs. unmaskable
- Within vs. between instructions
- Resume vs. terminate

Exception handling

1. Exception is associated with the only instruction presently being executed
2. Exceptions are managed by appropriate routines (e.g., Exception Service Routine or ESR)
3. When an exception is detected, the CU looks into a vector table. Each row of this table contains the memory address of the corresponding ESR
4. The ESR saves the PC of the excepting instruction, runs the instruction to handle the corresponding exception and finally returns to the original program if the exception requires resuming

In pipelined CPUs, the situation is much more involved because concurrent instructions are active in the pipeline. Two main problems:

1. Recognizing the exception (i.e., associating it with the correct instruction)
2. Saving the correct machine state

Some exception examples in a pipeline

- *Exceptions arising from instruction execution (e.g. overflow)*: usually raised during stage EX(i.e., within an instruction). They are also coerced, generally user-maskable and they require a resume.
- *External (I/O) interrupts*: Asynchronous, coerced exceptions requiring a resume. The ISR are usually served when the interrupted instruction is in the WB stage(i.e., between subsequent instructions).
- *Cache miss*: actually, only data cache miss are considered exceptions (instruction cache misses are directly managed by hardware, which stalls the CPU before IF phase). These exceptions (like page faults) are detected in MEM stage, are coerced, synchronous and non-maskable.
- *Non-existing OP code*: may be due to a compiler error, to a memory fault but also, in CPUs belonging to “families”, to instructions invoking functional units not present in the particular device used. This exception is raised in the ID stage;it is synchronous, coerced, non-maskable and leads to program termination.

3.5 Advanced pipelining

3.5.1 Superpipelining

A superpipelined processor has a pipeline where each logical step (IF, ID, EX, MEM, WB) is subdivided into simpler multiple pipeline stages. So smaller T_{CLK}.

Reduction of stage complexity is limited by four factors:

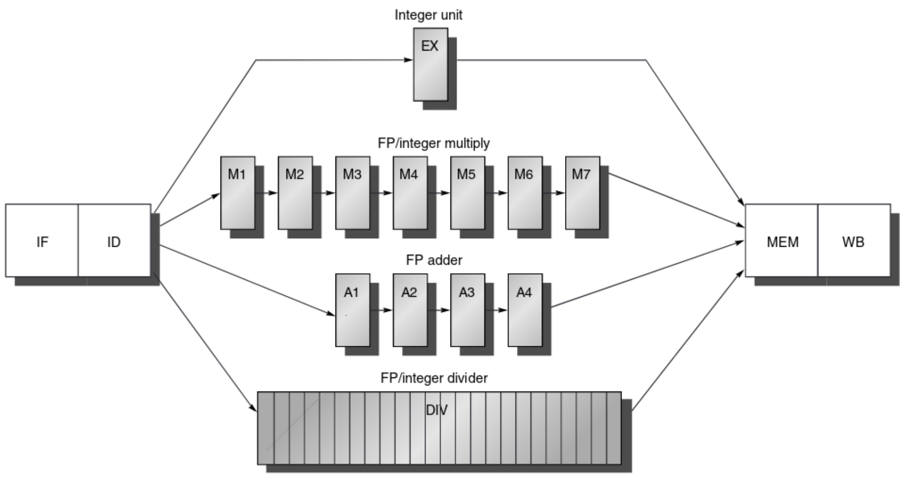
1. *Imbalance problems*:
   Imbalance between stages reduces performance because the clock period can not be shorter than the time needed for the slowest pipeline stage

2. *Pipeline overhead time*:
   It results from the combination of two contributions:
   - Pipeline register delay: this is due to setup and propagation time through the register. This is constant and related to technology
   - Clock skew: worst-case delay between active clock signal edges in two points of a digital system (e.g., between any two pipeline registers)

   The sum of these two contributions represents lower bound to clock cycle.
3. *Growing data dependencies*:
   This leads to a higher number of bubbles in a deeper pipeline–performance loss due to data hazards increases
4. *Longer control hazards*:
   imply slower branches in case of wrong predictions

Summing up: cycle time is shorter but the number of cycles required by a given program becomes larger.

Certain operations, such as floating point, may require multiple cycles to complete so it's impractical to have them complete in one cycle. The EX stage will be repeated as many times as necessary.



Problems

- Instructions may complete in a different order than they were issued
- WAW hazards are now possible, since instructions no longer reach WB in order
- WAR are still not possible, since registers are still read in ID, and instructions go through ID in order
- Stalls due to RAW will be more frequent, since instructions have longer latency. Instructions stalls for long periods awaiting the results from the floating point operations.
- Structural hazard: Instructions reach MEM and WB simultaneously!

3.5.2 Superscalar

Superscalar architecture is a single processor that can execute several scalar operations in parallel.

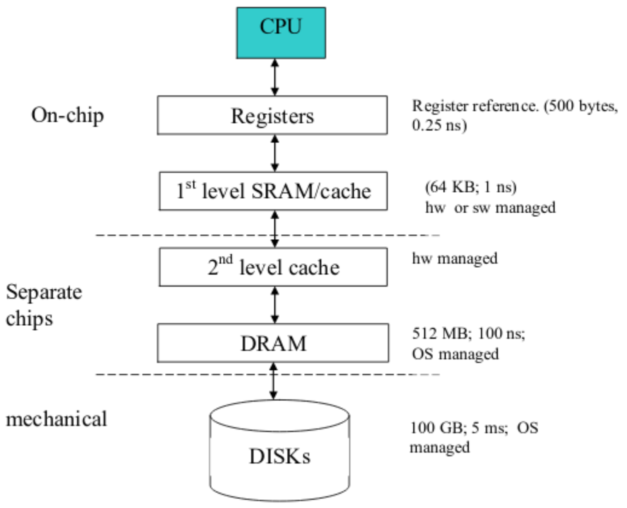
- Using different dedicated pipelines, implemented on the basis of the classes of instructions that are more frequently used on a given architecture.
- Using multiple identical pipelines, i.e. multiple copies of a given physical pipeline.

Chapter 4

Memory organization

Larger memories are slower than smaller memories. Fast memories are much more expensive than slow memories.

So instead of using a single large memory, modern systems use a hierarchy. Characterized by different technology, cost, dimensions and access mechanisms. The programmer sees an unique large memory space. The CPU sees a fast memory at acceptable cost.



Hierarchies work well because memory access follows the principle of locality.

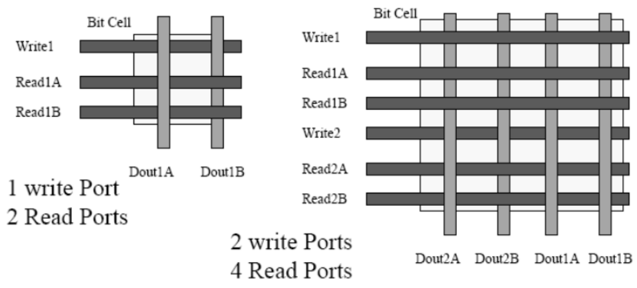
Locality

- **Temporal locality**:
  when a memory entry is referenced, with high probability it will be referenced again within a short time.
- **Spatial locality**:
  whenever a memory entry is referenced, with high probability reference will be made shortly to neighboring items.

Registers Registers are usually "exposed" to the programmer. Alternatively, the compiler allocates variables to registers. Registers are usually placed into a regular register file. A register file consists of multiple read/write ports to allow parallel, simultaneous data accesses. Typically, at least 2 read and 1 write ports are required to access operands of a single instruction.

Unfortunately, the size of the register file grows with the square of the number of ports so an excessive number of ports slows down the processors.

This quadratic increase in chip area is mostly due to the new additional input and output lines:
• Routing problems
• Internal bit cell loading increases -¿ larger power consumption
• Linear increment in delay with the number of ports



It is not profitable to have a register file with more than 15-20 ports: 12 is actually the best tradeoff when number of FU is larger than 4

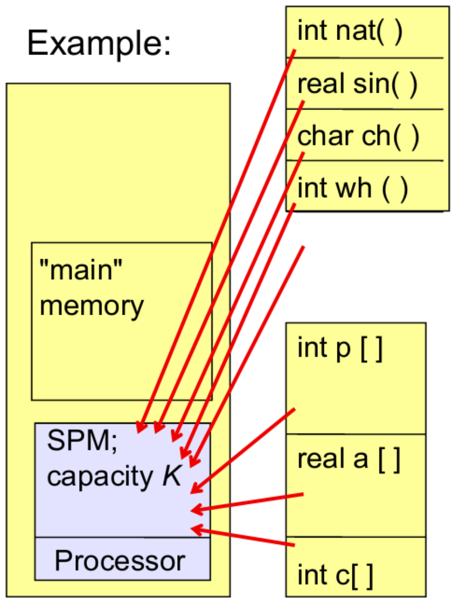
The usual SRAM

- It is "exposed" to programmer and compiler
- Data transfers from/to memory are performed by **software**
- It is mostly used in low-end or specialized embedded processors where worst-case latency to recover data and power are the main issues

Cache

- Usually it is "transparent" to programmer and compiler
- Transfers from/to lower-level memories are performed by **hardware**
- It is mostly used in high-end processors where the power is less critical and average performance should be maximized

Scratch Pad It's a low power consumption solution. The scratch pad is a buffer of memory (no cache) where most frequently used program functions instructions and data are allocated. This approach requires a preliminary "profiling"to decide what instructions have to be placed into the scratch pad.



Allocation usually performed statically at compile time (no HW support). This kind of solution is very used in embedded systems because the program is usually fixed and it is a reasonable tradeoff between performance and power consumption.

Cache A cache is a special kind of memory based on SRAM cells which is used to store the data having the maximum probability to be used. Cache can be either unified (i.e., simultaneously instruction-and data-cache) or split into I-cache and D-cache.

Definitions:

- **Hit**: the item requested by the CPU is present in cache
- **Miss**: the item requested by the CPU is not present in cache
- **Hit rate**: fraction of memory accesses rewarded by hit
- **Miss rate**: fraction of memory accesses resulting in a miss (miss rate = 1 -hit rate)
- **Hit time**: access time (or clock cycles) to cache in the case of success (includes time to determine whether access is met by hit or miss)
- **Miss penalty**: time (or clock cycles) required to substitute a block of data in cache with another block from the lower-level storage
- **Miss time**: miss penalty + hit time, time required to obtain requested data in the case of miss

Cache performance is given in terms of average memory access time (AMAT) $AMAT = hit - time + miss - rate * miss - penalty$

Cache performance affects CPU time $CPU_{time} = IC*(CPI_{execution} + \frac{MEM_{accesses}}{IC}*miss-rate*miss-penalty)*T_{CLK}$

How cache works A cache is made of a number of **blocks** that contain **words** from memory. The size of a block is a power of 2, to simplify addressing (e.g., a block is made of 32 words). When a word is needed, its entire block is loaded from memory and placed somewhere in the cache. Blocks that are next to each other in memory are not necessarily stored next to each other in the cache, and vice-versa.

The cache needs to store a block along with its actual address in memory. The required address is compared to the address of all the blocks in the cache to find a match. If found, it is a hit, otherwise it is a miss.

Problems in cache design

- **Placement problem**:
  Where to place a block transferred from lower to higher level
- **Search (or identification) problem**:
  how to determine whether the requested item is present or not
- **Substitution (or replacement) problem**:
  Which block present in the cache must be replaced by one in lower-level storage, in the case of a miss
- **Write strategy**:
  what happens when a write-to-memory instructions is executed?

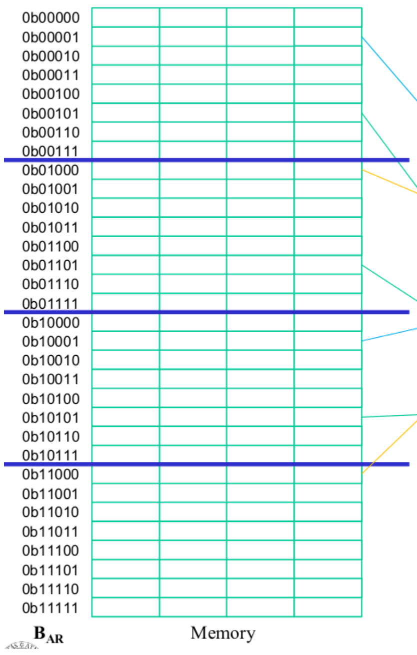
4.1 Direct-mapped cache

Assume the cache is made of an array of N_B blocks. Let B_{AR} be the address of the required block in RAM.

In a direct-mapped cache, the block can only be placed in the cache block at address B_{AC}:

$B_{AC} = B_{AR} mod N_B$

B_{AC} is also called the index of the cache block. If N_B is a power of 2, then the index is equal to the $N_{AC} = log2(N_B$ least significant bits of B_{AR}.



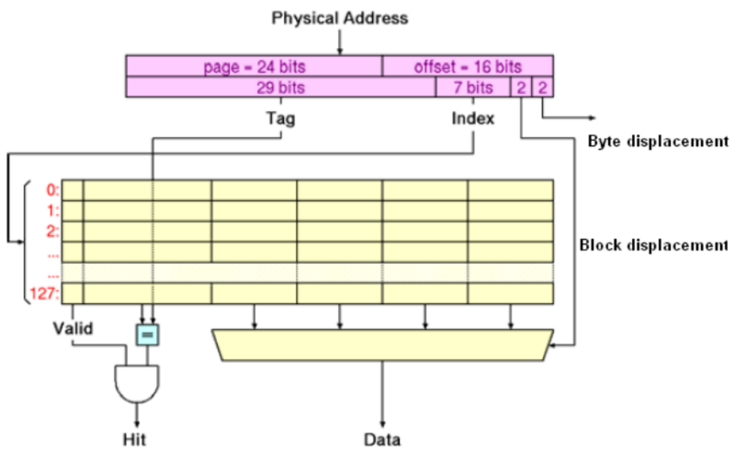
Example:
RAM size: 32 blocks (5 bits + 4 bits for byte and word, not shown)
cache size: $\mathbf{N_B}$ = 8 blocks (3 bits)
$\mathbf{B_{AC} = B_{AR} \text{ mod } N_B = B_{AR} \text{ mod } 8}$
Index is last $\mathbf{N_{AC}} = \mathbf{log_2(N_B)}$ = 3 bits of $\mathbf{B_{AR}}$

All memory addresses ending with 001 map onto cache index 001, etc

Problems

- **Placement problem**:
  Trivial – the block from RAM can be mapped onto one cache block only
- **Search problem**:
  different memory blocks can be placed in the same cache block. How do we decide which location is mapped in the cache?
  Solution: each cache line must be provided with:
  - A **tag field** containing the $N_{AR} - N_{AC}$ most significant bits of B_{AR}
  - A **validity bit** - denotes whether block contains meaningful (valid) data.
- **Replacement problem**:
  Trivial – the new block from RAM can be mapped onto one cache block only. Does not account for temporal locality! Block substituted may have been very recently used.

- Example: 40-bit memory address, 128-line cache with 128-bit block (4 words/block)
  - 2 bits for byte displacement (address of byte within a word, 4 bytes per word)
  - 2 bits for block displacement (address of word within a block, 4 words per block)
  - Index is 7 bits = $\log_2(128)$ because there are 128 blocks in the cache
  - Tag is 40 – 2 – 2 – 7 = 29 bits

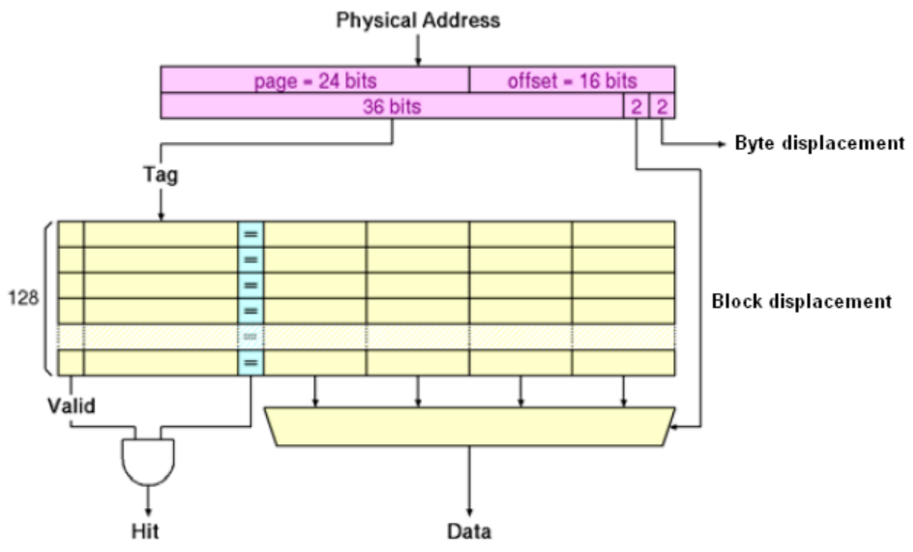


4.2 Fully associative cache

In fully-associative cache every block of RAM can be mapped onto any block of cache (there is no constraint on Placement). The search problem is tackled by running a parallel comparison of the wanted address with all tags. So a large number of comparators is done and so it's very expensive in terms of area and power consumption.

The index is 0 bits and the tag is the complete address of the word minus the bits for block and byte displacements.

Example: 128-line cache, 128-bit block (4 words/block)



Replacement strategies

- **Random**:
  the block to be substituted is chosen randomly.
  - *pro*: It is simple. Minimum HW support (pseudo-random number generator)
  - *cons*: Efficiency in terms of miss rate is comparable to DM cache. In fact, temporal locality not taken into account.
- **Least Recently Used (LRU)**:
  substitute the block left unused for the longest time.
  - *pro*: It fully exploits temporal locality. Maximum efficiency
  - *cons*: It requires HW support. The cost of this solution increases largely with the number of blocks. A counter per each block.
- **First In First Out (FIFO)** (also called Round Robin) of length N:
  substitute the block used N accesses before the present one. Whether it was used during the last N-1accesses or not.
  - *pro*: This is the best tradeoff between performance and complexity. It is adopted in several recent systems.
  - *cons*: It just approximates the locality principle

4.3 N-way set-associative cache

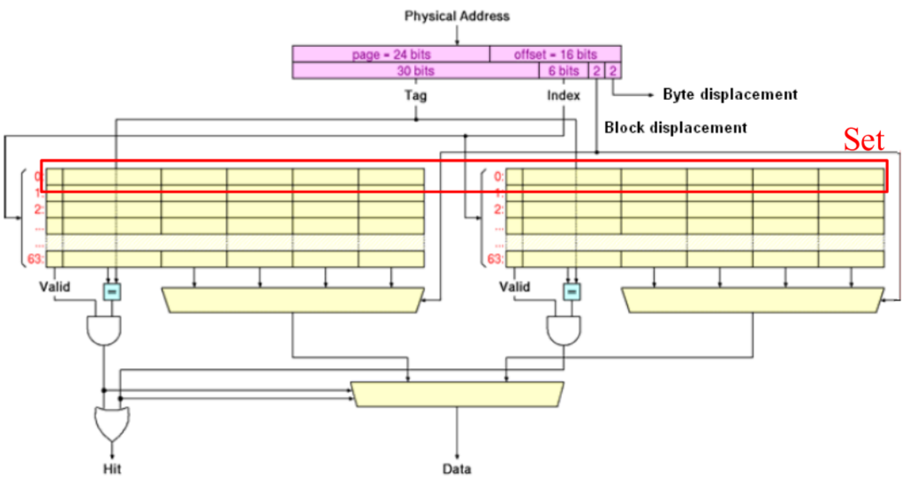
N-way set-associative cache is a tradeoff between DM and fully associative solutions. Cache is arranged in N_{set} sets (N_{set} being a power of 2), each set including N blocks. Notice: $N_{set} = N_B/N$ where N_B is the total number of blocks.

The index I_{set} of the Set in cache corresponding to a given memory block address B_{AR} is given by:
$I_{set} = B_{AR} mod N_{set}$

Each block within the set at index I_{set} can map onto any of the N cache blocks in the set, using fully associative technique.

The set in cache corresponding to block in RAM is identified as in DM cache (Index identifies Set). Within the set, block is identified with associative mechanisms (multiple tag comparisons in parallel).

Example: 2-way 128 block cache example (4 words/block)



The direct mapped cache is a 1-way set associative cache (set = 1 block). The fully associative cache has only one big set, the size of the whole cache.

Advantages compared to fully associative cache: comparators are fewer and smaller so cheaper and faster solutions.

Advantages compared to direct-mapped cache: better exploitation of temporal locality at a reasonable cost

The replacement of a block is performed by one of the techniques seen for associative cache (random, LRU or round-robin).

4.4 Causes of cache miss

- **Compulsory misses**:
  at the time of the first access, a block is never present in cache. Not dependent on cache size and architecture (dependent on block size, though)

- **Capacity misses**:
  if the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved. This kind of misses decreases with cache size

- **Conflict misses**:
  in direct-mapped or set-associative caches, blocks that are replaced may have to be reloaded later in the same set – leading to collision ("conflict") misses

4.4.1 The write problem

Write in caches are less frequent than reads. Instruction fetch only requires memory reads. "Load" instructions more frequent than "store".

In Write to memory operations we ask: **Speed**: implies writing to cache; **Consistency**: information in lower-layer memories must always be consistent with information in cache.

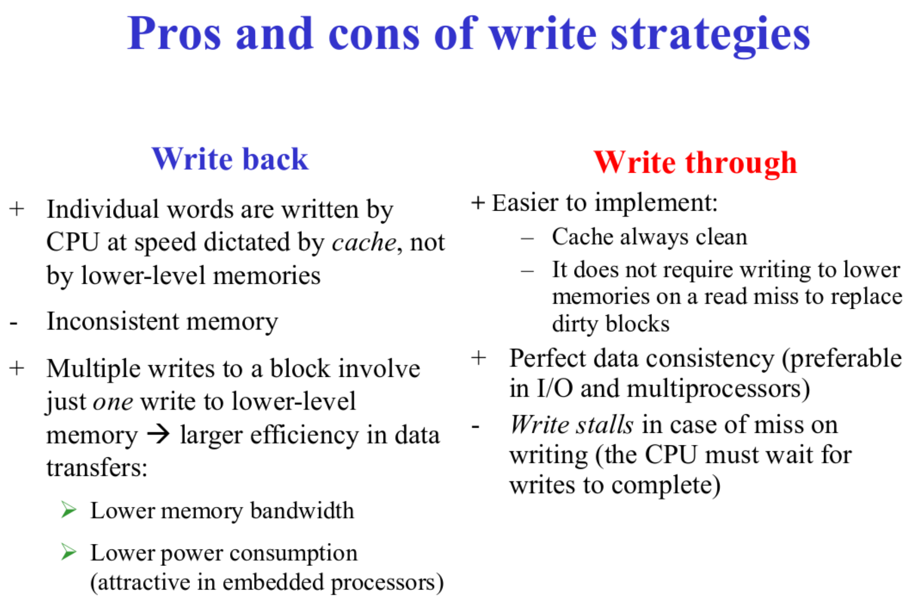
Write strategies:

Write-through information is written **simultaneously** in cache block and in the main memory block. Consistency always respected, access time for writes is that of the lower level of memory, so lower performance.

Write-back when a write instruction is executed, data are written **only in a cache** block. Lower-level memory block is modified instead **only when a substitution occurs**.

It's added a "dirty bit" to specify if the cache block is dirty or clean, so if the data is useful.

Generally, write-back is preferred to write-through because locality principle leads to high probabilities of writing again in a block recently accessed for a write. So there is a good probability of multiple writes before a substitution occurs.



Pros and cons of write strategies

Write back

+ Individual words are written by CPU at speed dictated by *cache*, not by lower-level memories

- Inconsistent memory

+ Multiple writes to a block involve just *one* write to lower-level memory → larger efficiency in data transfers:

- Lower memory bandwidth
- Lower power consumption (attractive in embedded processors)

Write through

+ Easier to implement:

- Cache always clean
- It does not require writing to lower memories on a read miss to replace dirty blocks

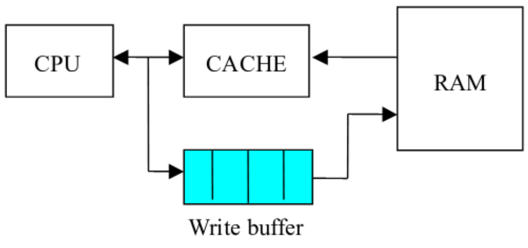
+ Perfect data consistency (preferable in I/O and multiprocessors)

- *Write stalls* in case of miss on writing (the CPU must wait for writes to complete)

Improving the efficiency of write through In order to reduce the latency due to write stalls, a write buffer between cache and lower-layer memory is inserted. CPU writes to cache and write buffer. The write buffer is a simple FIFO. Data are transferred from CPU to buffer at cache speed, and from buffer to memory at memory speed. Accordingly a write stall occurs only when the buffer is full,.

if a read occurs from a word still in write buffer:

1. Solution: read miss waits for write buffer to be empty (lower performance)
2. Solution: on a read miss, check contents of Write Buffer with word address requested by read, if no match serve the read miss first



Write misses Consists of attempts to write to an address not present in cache. Possible solutions:

- **Write allocate** (also fetch on write): the block is allocated in cache on a write miss, and then written into the lower-level memory (according to the chosen write policy)
  More used by write back due to locality principle.
- **No write allocate** (also write around): the block is modified directly in lower-level memory, without being transferred to cache.
  More used by write through because there would be no gain in passing through cache.

4.5 Improving cache performance

Aspects to consider

- Miss rate (compulsory, capacity, conflict)
- Miss penalty
- Hit time

Increase cache size Decreases miss rate but may increase hit time.

Larger block size to reduce miss rate Larger block size **reduces compulsory misses** (because of spatial locality)
Larger block size **increases miss penalty** (larger block to read from memory)
Larger block size implies fewer blocks for a fixed cache size, which **may increase conflict misses and capacity misses** (if cache is small)

4.6 Split cache

In a split cache we have two first level cache, one for the instructions and one for the data. Statistics prove that instruction caches have lower miss rate than data caches.
Instruction cache: "read-mostly" – very good spatial locality
Data cache: locality strongly application-dependent.

Average Memory Access Time for split caches:

$AMAT = \%Instr\_references(hit\_time+miss\_rate\_I*miss\_penalty)+ \%Data\_references(hit\_time+miss\_$

miss_rate_I = miss rate for instruction cache
miss_rate_D = miss rate for data cache
%Instr_references = fraction of total memory references for instructions
%Data_references = fraction of total memory references for data

Cache locking some sections of cache can be locked by the programmer to be used as fast local memories. The locked portion of cache is visible to the programmer. The caching hardware of the locked section (e.g. tags, replacement logic) is turned off (i.e. cache becomes a normal RAM). This is possible only with associative or set-associative caches.

4.7 Multi-level cache

The total AMAT is given by:

$$T_A = Hit\_time_{L1} + Miss\_rate_{L1} * Miss\_penalty_{L1} \; = Hit\_time_{L1} + Miss\_rate_{L1} * \; (Hit\_time_{L2} + Miss\_rate_{L2}$$

4.7.1 Type of miss rates

- **Local miss rate**:
  number of misses in a cache divided by the total number of accesses to that cache ($Miss\_rate_{L1}$ for L1 and $Miss\_rate_{L2}$ for L2)
- **Global miss rate**:
  number of misses in a cache divided by total number of memory accesses generated by CPU (either $Miss\_rate_{L1}$ for L1 and $Miss\_rate_{L1} * Miss\_rate_{L2}$ for L2)

L1 and L2 caches should be different to improve overall performance.

4.7.2 Inclusion vs exclusion policies

- **Multilevel inclusion**:
  L1 data always present in L2 (it ensures consistency between I/O and caches). *Problem*: performance analysis suggests that L1 blocks may be smaller than L2 ones so that substitution might become difficult.
- **Multilevel exclusion**:
  L1 data are never found in L2. Reasonable if L1 and L2 have similar size

4.7.3 Split vs unified

Performance analysis showed that the best approach is to use

- L1 cache: Split
  High bandwidth for simultaneous data and instruction access. Separate optimization
- L2 cache: Unified
  Reduced area size. Better flexibility on memory usage

4.7.4 Write policy

- L1 cache: Write through + no-write-allocate
  Avoids problems with coherence in L1. Simpler implementation for on-chip solution
- L2 cache: Write back + write-allocate
  Reduces overall bus traffic. Captures temporal and spatial locality

4.7.5 Block size

A larger block size assures a better use of spatial locality. Compulsory misses decrease. Conflict and capacity misses may increase, since larger blocks means fewer blocks in cache. However, miss penalty increases so there would be a higher access time.

4.7.6 Degree of associativity

- L1 cache: No clear winner
  DM: faster cycle time, lower hit rate
  Set associative: longer hit-time vs. better hit rate
- L2 cache: No clear winner
  Set-associative less advantageous for really large caches
  Set-associative L2 gives flexibility

Critical Word First request missed word from memory, send it to CPU as soon as it arrives; CPU runs while the rest of the cache block is filled (also called wrapped fetch, request word first).

Early restart words are fetched in normal order, as soon as the requested one arrives it is sent to the CPU.

Victim cache A small, fully associative cache (victim cache) can be inserted between cache and its refill path; Victim cache contains only blocks discarded from cache on a miss (victims) that are checked on a miss before going to lower-level memory.
If requested data are in victim cache, victim block and cache block are swapped.

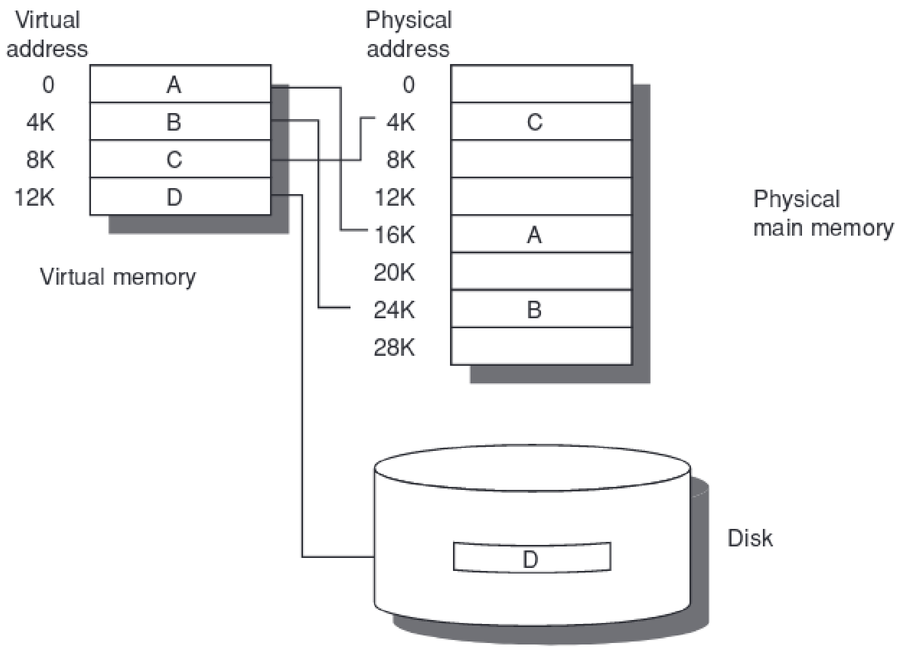
4.8 Virtual Memory

Computers run multiple processes, each with its own address space. Can't give each process a full address space worth of memory.

The solution is to divide physical memory into blocks and allocate them to different processes. Need a protection scheme so that processes can access only their blocks.

Virtual memory also useful to offload to secondary storage (disks) parts of (block) programs and data that can't fit in main memory. The swapping between main memory and secondary storage is automatically managed by the operating system.

Process sees a virtual address space. Blocks in virtual space are allocated anywhere in physical memory or in secondary storage.



Similar to caches: blocks usually called pages or segments and misses are called page faults.

The virtual address generated by the process must be translated into the physical address. It's called memory mapping or address translation.

4.8.1 Differences with caches

Replacement

- Caches: controlled by the hardware
- Virtual memory: controlled by the operating system. The high miss penalty means it is more important to make a good replacement!

Size

- Processor address bus size determines size of virtual memory
- Cache size independent of processor address bus size

Secondary storage Also used for file system

Blocks can be placed anywhere in main memory, it's like a **fully associative** cache.

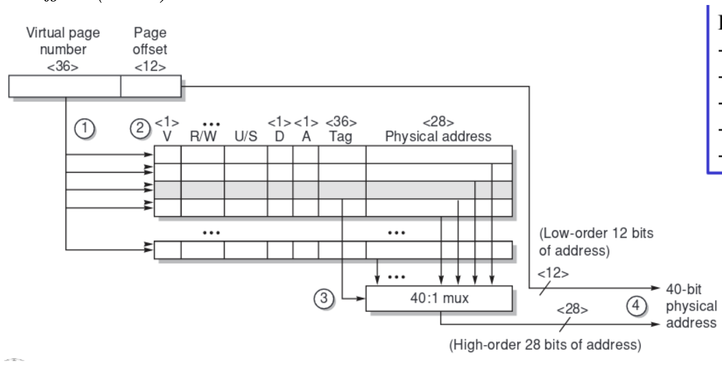
On a page fault for replacing a block most operating systems use **LRU**.

For writing a page all virtual memory systems use **write-back**.

To search a block in main memory it is used a table to record correspondence between virtual page number and physical address. Sometimes a hash table is used, the size of the actual number of pages in physical memory in use.

Fast address translation Page tables are usually large so are stored in main memory and paged. 2 memory accesses are needed: one for translation, one for data.

Address translation uses a special fully associative cache: *Translation lookaside buffer (TLB)*



Bits:
- V: valid
- R/W: read/write
- U/S: user/system
- D: dirty
- A: accessed

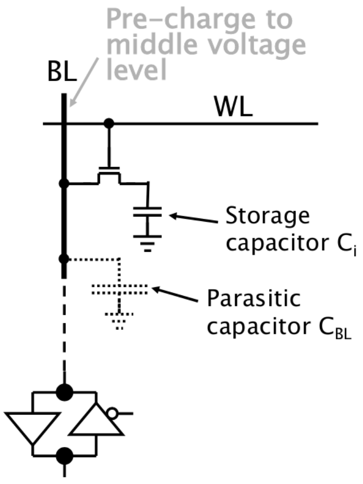
4.9 A Dynamic RAM (DRAM) cell

In most dynamic memories 1 cell consists of just 1-transistor as a storage capacitor C_i. Charge sharing with bit line capacitance ($C_i << C_{BL}$). Due to leakage currents, memory cell content must be refreshed periodically.

Reading much slower than writing because it implies 3 operations:

- Precharge
- Reading
- Write back

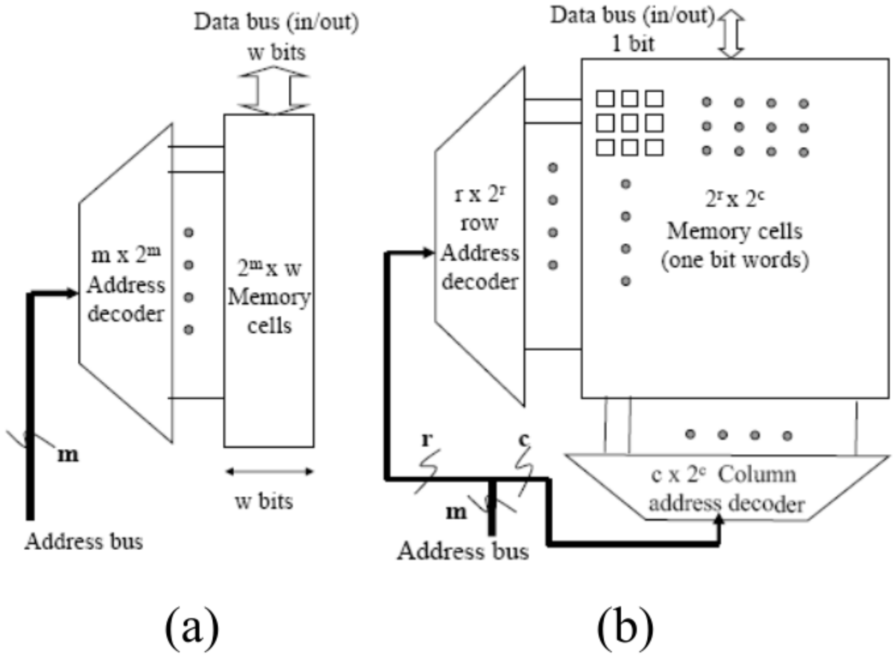
DRAM much slower than SRAM and it is also more power hungry.



Pre–charge to middle voltage level
BL
WL
Storage capacitor C_i
Parasitic capacitor C_{BL}

4.10 "Typical" memory structure

The SRAM memory has usually a cell array layout.



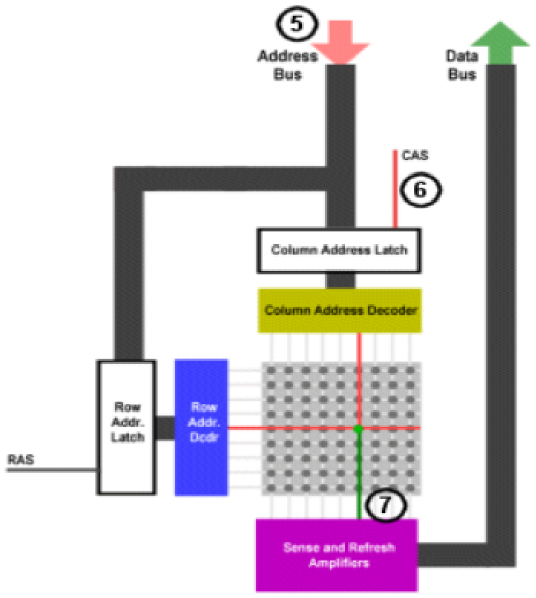
Ideally it should be linear (a), but this approach is not suitable:

- Long data bus so large capacity
- Address decoder will have a lot of logic levels
- Memory layout too long and thin

It is better to split the m-bit address into an r-bit row address and a c-bit column address (b). We reach smaller decoders, almost square memory layout, faster architecture (lower WL and BL delays)

4.11 DDR commands

- **PRECHARGE**
  Close current row, get ready for an activate
- **ACTIVATE (RAS)**
  Open a row and store row result in a register. Row is open until next precharge command.
- **READ (CAS)**
  Read values from open row
- **WRITE (CAS)**
  Write values to open row
- **REFRESH**
  Restore values in rows (essentially a periodic ACTIVATE)



1. The row address is placed on the address pins via the address bus
2. The Row Access Strobe (RAS) is activated, which places the row address onto the Row Address Latch
3. The Row Address Decoder selects the proper row to be sent to the sense amps
4. The column address is placed on the address pins via the address bus
5. The Column Address Strobe (CAS) is activated, which places the column address on the Column Address Latch
6. The CAS pin also serves as the Output Enable, so the sense amps place the data from the selected row and column on the Data Out pin

4.12 Synchronous DRAM (SDRAMs)

- A specific CLK signal is transmitted over a dedicated control line
- Communication times are based on a multiple of the clock period
- Commands are sent and managed like in a pipeline
- Burst transfer(i.e. transfer of chunks of data after a single addressing phase) are possible
- In double data rate (DDR) SDRAM memories, both clock edges are used to transfer data

4.13 Magnetic Disks

- *Cylinder*: all the tracks under the arms at a given point on all surfaces.
- *Seek time (Ts)*:time to move the arm to the desired track (5-12 ms on average)
- *Rotation latency(or delay) (Tr)*: time for the requested sector to rotate under the head. The average latency to the desired information is obviously halfway around the disk (3-5 ms)
- *Transfer time (Tt)*: time to transfer a block of bits, typically a sector, under the read-write head. This time is a function of the block size, disk size, rotation speed, recording density of the track, and speed of the electronics. A disk buffer (4-16 MB) can improve Tt by exploiting spatial locality
- *Controller time (TC)*: overhead the disk controller imposes in performing an I/O access

$Access\_time = T_s + T_r + T_t + T_c$

Chapter 5

Branch Prediction