Advanced Computer Architectures

Professor Marco D. Santambrogio

Spring 2022

Contents

1 Introduction

1.1 Flynn Taxonomy

Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle.

Single data: only one data stream is being used as input during any one clock cycle.

Multiples instruction: every processor may be executing different instruction streams.

Multiple Data: every processor may be working with different data streams.

Based on how many instruction/data streams a processor can be working on, the architecture relying on it can be classified as:

- SISD - Single Instruction Single Data – Uniprocessor systems
- MISD - Multiple Instruction Single Data
  No practical configuration and no commercial systems.
- SIMD - Single Instruction Multiple Data
  Simple programming model, low overhead, flexibility, custom integrated circuits.
- MIMD - Multiple Instruction Multiple Data
  Scalable, fault tolerant, off-the-shelf micros.

1.2 Parallelism

The SISD architecture is the oldest and provides deterministic execution. To account for events happening simultaneously we have to introduce parallelism.

The SIMD, parallel architecture, with multiple data each processing unit can operate on a different data element. Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.

The MIMD, nowadays, the most common type of parallel computer, execution can be synchronous or asynchronous, deterministic or non-deterministic.

Type of parallelism:

- DLP, data-level parallelism (SIMD)
- ILP, instruction-level parallelism (SIMD + pipeline)
- TLP, thread-level parallelism (MIMD)
- RLP, request-level parallelism (MIMD)

1.2.1 Instruction-Level Parallelism

Instruction level parallelism refers to the ability of executing in parallel different steps of different instructions (e.g, fetch, decode, execution, etc.) taking place in different physical parts of a processor.

1.2.2 Thread-Level Parallelism

Explicit parallelism implies structuring the applications into concurrent and communicating tasks. Operating systems offer support for different types of tasks. The most important and frequent are:

- processes
- threads

The operating systems implement multitasking differently based on the characteristics of the processor. Multithreading categories:

- **single core**
  - superscalar
  - multiprocessing
- **single core with multithreading support**
  - Fine grained multithreading
    - Switches from one thread to the other at each instruction
    - the execution of more threads is interleaved (often the switching is performed taking turns, skipping one thread if there is a stall)
    - The CPU must be able to change thread at every clock cycle. It is necessary to duplicate the hardware resources.

    →*Advantage:* it can hide both short and long stalls, since instructions from other threads is executed when one thread stalls
    →*Disadvantages:* is it slows down execution of individual threads, since a thread ready to execute without stalls will be delayed by instructions from other threads
  - Coarse grained multithreading
    - switching from one thread to another occurs only when there are long stalls -– e.g., for a miss on the second level cache.
    - Two threads share many system resources (e.g., architectural registers), the switching from one thread to the next requires different clock cycles to save the context.

    →*Advantages:* in normal conditions the single thread is not slowed down
    – Relieves need to have very fast thread-switching
    – Doesn't slow down the single thread, since instructions from other threads are issued only when the thread encounters a costly stall
    →*Disadvantages*: for short stalls it does not reduce the throughput loss
    – the CPU starts the execution of instructions that belonged to a single thread, when there is one stall it is necessary to empty the pipeline before starting the new thread
  - Simultaneous Multithreading (SMT)
    - The system can be dynamically adapted to the environment, allowing (if possible) the execution of instructions from each thread, and allowing that the instructions of a sin-

gle thread uses all functional units if the other thread incurs in a long latency event.

  * More threads grant more (instruction) issues possibilities by the CPU at each cycle; ideally, the exploitation of the issues availabilities is limited only by the unbalance between resources requests and availabilities.

- **multicore** Multicore architectures broaden the prospective, allowing for different type of architectures to interact, trying to exploit the best from each one – e.g, the combination of cpu and gpu in modern computers. A multicore architecture can be:
  - homogeneous: duplication of the same core
  - heterogeneous: different processing architecture interacting

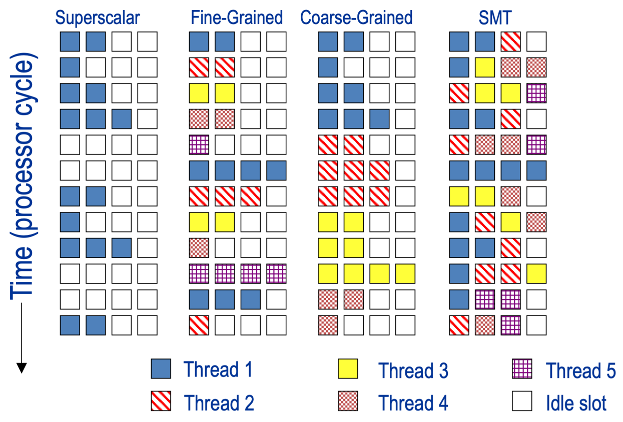


Figure 1: Multithreaded categories

TLP (e.g., superscalar) and ILP (pipelining) exploit two different forms of parallelism in a program, the following questions arise:

- Could a processor oriented at ILP exploit TLP too?
- Functional units are often idle in data path designed for ILP because of either stalls or dependencies in the code. Could the TLP be used as a source of independent instructions that might keep the processor busy during stalls?
- Could TLP be used to employ the functional units that would otherwise lie idle when insufficient ILP exists?

2 Performance and Cost

When we say that one computer is faster than another what do we mean? It depends on what is important.

Two metrics:

Computer system user minimize elapsed time for program execution:
latency: (response time) execution time = time_end - time_start

Computer center manager maximize completion rate $= \frac{\#jobs}{\text{sec}}$
throughput: total amount of work done in a given time

Throughput $= \frac{1}{latency}$ only if there is not overlap between instructions, otherwise throughput > response time.

2.1 Measures

$$\text{"X is k\% faster than Y"} = \frac{\text{execution time (y)}}{\text{execution time (x)}} = 1 + \frac{n}{100}$$

$$\textbf{Performance (x)} = \frac{1}{\text{execution time(x)}}$$

$$\Rightarrow \text{"X is k\% faster than Y"} = \frac{\text{performance (x)}}{\text{performance (y)}} = 1 + \frac{n}{100}$$

$$\textbf{Speed up (x,y)} = \frac{\text{performance (x)}}{\text{performance (y)}}$$

2.2 Amdahl's Law

Suppose that an enhancement E accelerates a fraction F of the task by a factor S (speedup enhanced), and the remainder of the task in unaffected.

Figure 2: In green the (F) fraction enhanced.

$$ExTime_{new} = ExTime_{old} \times \left((1 - F_{en}) + \frac{F_{en}}{S_{en}} \right)$$

$$\mathbf{S_{overall}} = \frac{ExTime_{old}}{ExTime_{new}} = \frac{1}{(1 - F_{en}) + \frac{F_{en}}{S_{en}}}$$

$$\mathbf{Speedup_{overall(max)}} = \frac{1}{(1 - F_{en})}$$

This means that if an enhancement is only usable for a fraction of a task we can't speed up the task by more than the reciprocal of 1 minus the fraction. An upgrade is worth it if $costs < speedup_{overall}$.

Note: in case of threads:

$$\mathbf{Speedup_{overall}} = \frac{1}{s + \frac{p}{N}}$$

s = serial part = 1 - Fraction enhanced
p = 1 - s = parallelized part = Fraction enhanced
N = number of processors or threads = Speedup

2.3 CPU time and CPI

For computer architects:

$$\text{Response time} = \text{CPU time} + \text{I/O wait}$$

The **CPU time** does not include I/O wait time and corresponds to the time spent running the program.

$$\text{CPU time (P)} = \frac{\text{cc needed to execute P}}{\text{clock frequency}}$$
$$= \text{cc needed to execute P} \times \text{cc time}$$

$$\textbf{CPI} = \frac{cycles}{instruction}$$

$$\textbf{IC} = \frac{instructions}{program}$$

$$\textbf{Cycle Time} = \frac{time}{seconds}$$

$$\textbf{CPU time} = IC \times CPI \times Cycle\_Time = \frac{seconds}{program}$$

2.4 MIPS and MFLOPS

MIPS has its shortcomings: it counts every instruction as if they were all equal, CPI varies with different instructions.

$$\textbf{MIPS} = \text{millions of instructions per second}$$
$$= \frac{\text{number of instructions}}{\text{execution time} \times 10^6}$$
$$= \frac{\text{clock frequency}}{\text{CPI} \times 10^6}$$

MFLOPS takes care of the problems related to MIPS and considers only the number of floating point operations, which we assume independent from compiler and ISA.

$$\textbf{MFLOPS} = \frac{\text{Floating point operations in program}}{\text{CPU time} \times 10^6}$$

3 Pipeline

3.1 MIPS

The MIPS architecture is based on the SIMD model, the instruction set (ISA) is based on 32-bit format instruction.

The CPU, central process unit, amounts to a control unit plus a data path. Communication between the CPU and the memory in the computing infrastructure happens through the control, data and address buses.

Execution of MIPS instructions Every instruction in MIPS subset can be implemented in at most 5 clock cycles as follows:

IF Instruction Fetch Cycle
Send the content of Program Counter register to Instruction Memory and fetch the current instruction from Instruction Memory.
Update the PC to the next sequential address by adding 4 to the PC (since each instruction is 4 bytes).

ALU Instructions: `op $x,$y,$z`

Instr. Fetch & PC Increm.	Read of Source Regs. `$y` and `$z`	ALU OP (`$y op $z`)	Write Back of Destinat. Reg. `$x`

Load Instructions: `lw $x,offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. `$y`	ALU Op. (`$y+offset`)	Read Mem. `M($y+offset)`	Write Back of Destinat. Reg. `$x`

Store Instructions: `sw $x,offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. `$y` & Source `$x`	ALU Op. (`$y+offset`)	Write Mem. `M($y+offset)`

Conditional Branch: `beq $x,$y,offset`

Instr. Fetch & PC Increm.	Read of Source Regs. `$x` and `$y`	ALU Op. (`$x-$y`) & (`PC+4+offset`)	Write PC

Figure 3: MIPS instructions

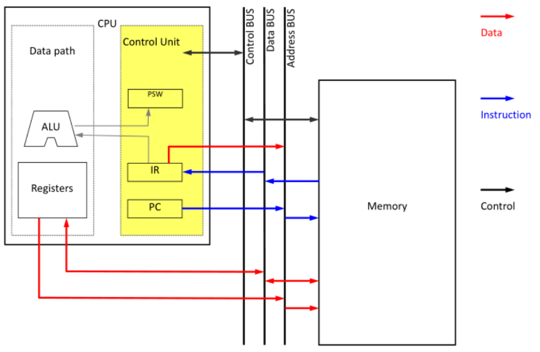


Figure 4: Computing infrastructure

ID Instruction Decode and Register Read Cycle
Decode the current instruction (fixed-field decoding) and read from the Register File of one or two registers corresponding to the registers specified in the instruction fields.
Sign-extension of the offset field of the instruction in case it is needed.

EX Execution Cycle
The ALU operates on the operands prepared in the previous cycle depending on the instruction type:

- Register-Register ALU Instructions: ALU executes the specified operation on the operands read from the RF
- Register-Immediate ALU Instructions: ALU executes the specified operation on the first operand read from the RF and the sign-extended immediate operand
- Memory Reference: ALU adds the base register and the offset to calculate the effective address.
- Conditional branches: Compare the two registers read from RF and compute the possible branch target address by adding the sign- extended offset to the incremented PC.

ME Memory Access
Load instructions require a read access to the Data

Memory using the effective address.

Store instructions require a write access to the Data Memory using the effective address to write the data from the source register read from the RF.

Conditional branches can update the content of the PC with the branch target address, if the conditional test yielded true.

WB Write-Back Cycle

Load instructions write the data read from memory in the destination register of the RF.

ALU instructions write the ALU results into the destination register of the RF.

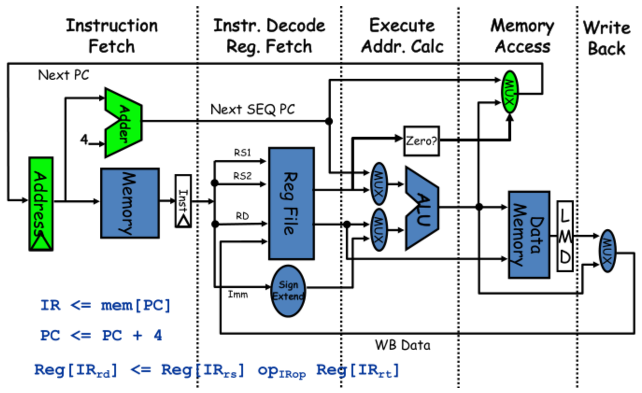
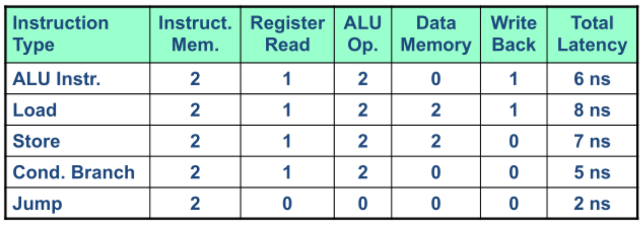


Figure 5: MIPS data path

In the single cycle implementation of MIPS the length of the clock cycle is defined by the critical path given by the load instruction: T = 8ns. Each instruction is executed in a single clock cycle of 8ns.



Instruction Type	Instruct. Mem.	Register Read	ALU Op.	Data Memory	Write Back	Total Latency
ALU Instr.	2	1	2	0	1	6 ns
Load	2	1	2	2	1	8 ns
Store	2	1	2	2	0	7 ns
Cond. Branch	2	1	2	0	0	5 ns
Jump	2	0	0	0	0	2 ns

Figure 6: Instruction latency

In the multi-cycle implementation the instruction is distributed on multiples cycle (5 for MIPS).

The basic cycle is smaller of 2 ns, the instruction latency is 10 ns.

The multi-cycle execution can be sequential of pipelined.

Latency Each instruction is worsened from 8ns to 10ns.
Throughput Is improved by four times, 1 instruction each 8ns to 1 instruction each 2ns.

The Register File is used in WB and ID stages. In the **optimized pipeline** the RF-write in the first half of the clock cycle and the RF-read occurs in the second half of the clock cycle avoiding a possible read after write hazard which otherwise would've required a stall. It is safely to assume that the optimized pipeline will be the standard pipeline throughout the course.

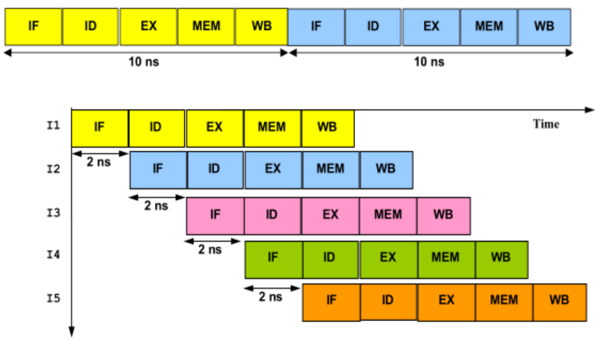


Figure 7: Sequential vs pipelined execution

4 Hazards

Dependencies are created at code-level and can be translated into conflicts when the compiler translate the code into instructions.

A hazard is created whenever there is a conflict (and therefore a dependency) between instructions, and these instructions are close enough that the overlap caused by pipelining would change the order of access to the operands involved in the dependence.

Hazards prevent the next instruction in the pipeline from executing during its designated clock cycle, reducing the performance from the ideal speedup gained by pipelining.

4.1 Structural hazards

Use of the same resource, such as the ALU, from different instructions simultaneously. There cannot be structural hazards in the MIPS architecture:

Instruction Memory separated from Data Memory, thus allowing the fetch of instructions and the reading of register during the same clock cycle.

Register File (RF) can be accessed twice in the same clock cycle: write access on the rising edge of the clock and read access on the falling edge by another instruction.

4.2 Data Hazard

Attempt to use a result before it is ready, three different types:

RAW, read after write

WAR, write after read

WAW, write after write

The RAW hazard is the only possible hazard in the MIPS architecture since all five stages are accessed in the same order in and between instructions, this is due to the fact that in the MIPS architecture there cannot be simultaneous or out-of-order access to the same resource. The access to the resources follow the order of the instructions.

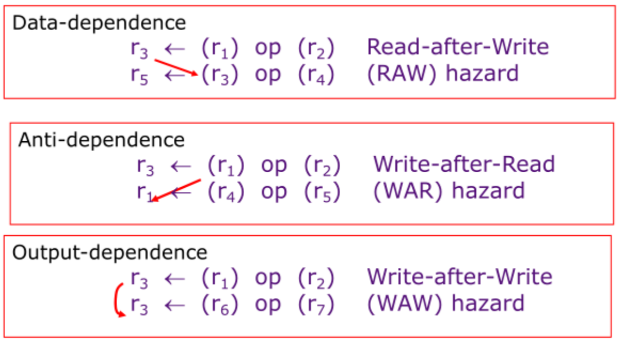

Data-dependence
$r_3 \leftarrow (r_1)$ op (r_2)
$r_5 \leftarrow (r_3)$ op (r_4)
Read-after-Write (RAW) hazard
Anti-dependence
$r_3 \leftarrow (r_1)$ op (r_2)
$r_1 \leftarrow (r_4)$ op (r_5)
Write-after-Read (WAR) hazard
Output-dependence
$r_3 \leftarrow (r_1)$ op (r_2)
$r_3 \leftarrow (r_6)$ op (r_7)
Write-after-Write (WAW) hazard

Figure 8: Data hazards

For this reason write backs are in-order and therefore WAW cannot happen, and the reading of registers of a previous instruction happens always before the write back of a subsequent instruction precluding the WAR hazards.
In MIPS, RAW hazards can be dealt with at compile-time (design solution) or at run-time (hardware solution).

Compilation Techniques

→ Nop instructions, insertion of no operation

→ Instruction Scheduling, to avoid that correlating instructions are too close the compiler tries to insert independent instructions among correlated instructions.

Scheduling is a reordering of independent instructions without changing the overall result; when the compiler does not find independent instructions, it inserts nops.

Hardware Techniques

→ Insertion of stalls (or bubbles) in the pipeline

→ Data Forwarding (or Bypassing)

Forwarding data means shortcutting data from one resource of the CPU, as output, to another resource of the CPU, as input, using one clock cycle for the propagation, this means using temporary result stored in the pipeline instead of waiting for the write back of results in the RF. Forwarding costs are additional multiplexers to allow to fetch inputs from the pipeline.
In the MIPS architecture the common paths for data forwarding are EX-EX, MEM-EX, MEM-MEM.
Note that WB-ID is not a data path, data is written and accessed in the same clock cycle inside the Register File. MEM-ID path is not useful if we divide clock cycle in rising and falling edge (optimized pipeline).

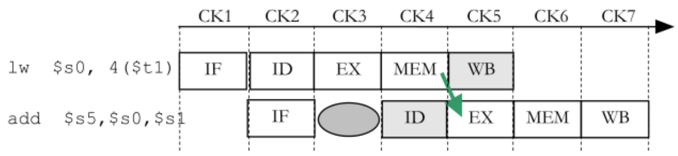


Figure 9: Load/Use hazard requires a stall to use the MEM-EX path

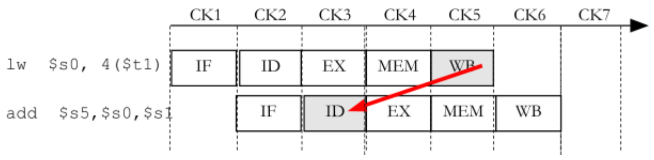


Figure 10: Load/Use without forwarding needs two stalls

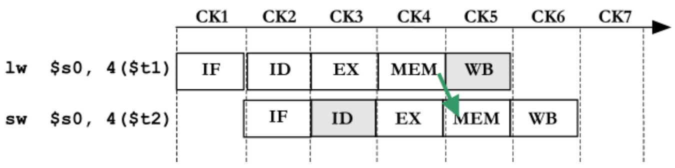


Figure 11: Load/Store hazard uses the MEM-MEM path

4.3 Control hazards

Attempt to make a decision on the next instruction to execute, before the condition is evaluated. A MUX is responsible for the value of PC, which is based on ALU output: until the ALU doesn't evaluate the condition the PC is not known, and we can't know which is the next instruction.
The PC is available at the MEM stage.
Generally true statements are executed in the waiting for the evaluation, as if the condition were true and no jump was needed: just add to the PC plus one, and then see if the branch is taken. For this particular reason it is suggested to write inside the `IF` statements the code that most probably is going to be executed.
In case we wait for the evaluation of the condition the stalls required are two in case of forwarding, three without forwarding.
An additional solution would be the early evaluation of the PC in the ID stage instead of the EXE stage, in this new pipeline the PC adder is anticipated and only one stall would be required (with forwarding) to fetch the correct instruction. This solution brings an issue in presence of a RAW hazard: anticipating the PC adder means that we have to add a stall in order to solve the conflict.

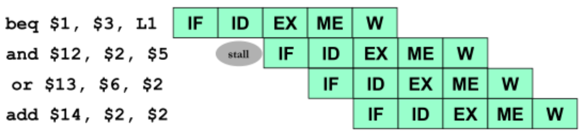


Figure 12: Early evaluation of the PC

5 Branch prediction

When a branch is taken the **branch target address** (BTA) is stored in the PC instead of the address of the next instruction in the sequential instruction stream. The branch outcome and Branch Target Address are ready at the end of the EX stage, conditional branches are solved when PC is updated at the end of the ME stage. Control

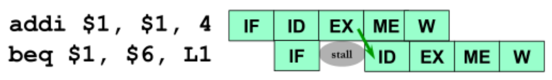


Figure 13: PC early evaluation issue

hazards reduce the performance from the ideal speedup gained by the pipelining since they can make it necessary to stall the pipeline.

$$\text{Pipeline Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Pipe Stall Cycles per Instruction due to Branches}}$$

$$= \frac{\text{Pipeline Depth}}{1 + \text{Branch Frequency x Branch Penalty}}$$

Figure 14: Performance impact by control hazards

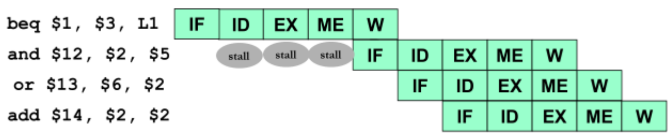


Figure 15: Branch stalls without forwarding

In order to reduce the speed penalty cause by the branch hazards we can use prediction techniques in order to guess the next instruction to be executed.

5.1 Static branch prediction

Static branch prediction are at compile-time:

- Branch always taken - No jump
- Branch always not taken
  In theory always jump, but in reality we have to calculate the BTA before jumping and at this point there is no prediction to make (at least in the MIPS pipeline).
- Backward taken, forward not taken
  Based on the assumption that backward jumps (while and for loops) are taken and forward jumps (if statements) are not taken.
- Profile driven prediction
  We are going make several runs of our program and the choice is based on statistical data for each branch.
- Delayed Branch
  The compiler statically schedules and independent instruction in the branch delay slot, which is executed whether the branch is taken. There are three ways in which the branch delay slot can be scheduled:

  From before, the instruction in the delay slot is always executed.

  From target, this strategy is preferred when the branch is taken with high probability.

  From fall-through, this strategy is preferred when the branch is not taken with high probability. In this case the branch delay slot is scheduled from the not-taken fall-through path.

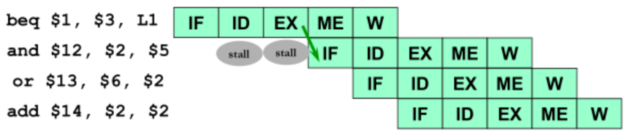


Figure 16: Branch stalls with forwarding

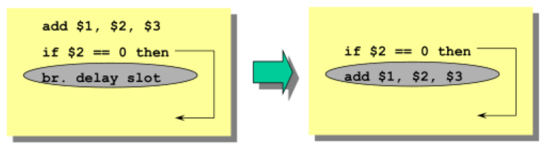


Figure 17: Branch delayed - from before

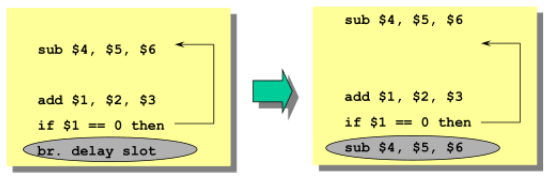


Figure 18: Branch delayed - from target

5.2 Dynamic branch prediction

Dynamic branch prediction happens at run-time by the hardware, and is based on two interacting mechaninsms:

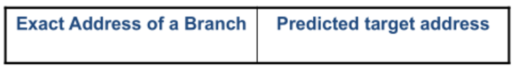
- Branch Outcome Predictor
  To predict the direction of a branch (i.e., taken or not taken).

  Branch History Table
- Branch Target Predictor
  To predict the branch target address in case of taken branch.

  Branch Target Buffer

5.2.1 Branch Target Buffer

The Branch Target Buffer (BTB or Branch Target Predictor) is a cache storing the predicted branch target address for the next instruction after a branch. We access the BTB in the IF stage using the instruction address of the fetched instruction (a possible branch) to index the cache.

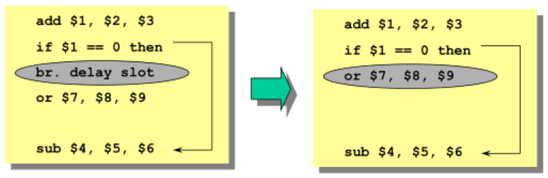


Exact Address of a Branch	Predicted target address

Figure 20: BTB entry

5.2.2 Branch History Table

The Branch History Table (BHT or Branch Prediction Buffer) is indexed by the last k-bits of the branch address, meaning that there are at most 2^k entries, with possible collisions, and each entry contains n-bits that says whether the branch was recently taken.



```
add $1, $2, $3
if $1 == 0 then
br. delay slot
or $7, $8, $9

sub $4, $5, $6
```

```
add $1, $2, $3
if $1 == 0 then
or $7, $8, $9

sub $4, $5, $6
```

Figure 19: Branch delayed - from fall-through

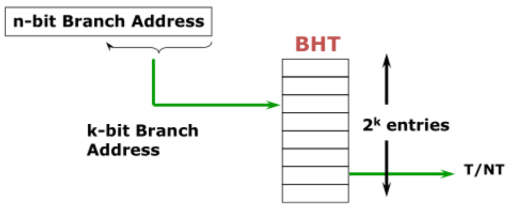


Figure 21: Branch history table

The 1-bit history table tells us if the last time the branch was taken or not taken (e.g., 0 and 1). Differently from the profile drive prediction the BHT is updated whenever the branch is encountered.

For example:
0, prediction not taken, outcome not taken - no update
0, prediction not taken, outcome taken - update 0 to 1
0, prediction taken, outcome not taken - update 1 to 0

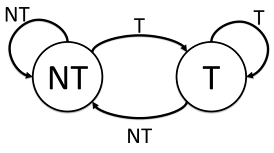


Figure 22: 1-bit BHT automata

A misprediction occurs when:

→ The prediction is incorrect for that branch

→ The same index has been referenced by two different branches, and the previous history refers to the other branch. To solve this problem it is enough to increase the number of rows in the BHT or to use a hashing function.

Among the n-bits BHT the **2-bit history table** is the best one because it is still small in size, it allows for speedy updates maintaining the prediction accuracy high, but differently from the 1-bit the prediction must miss twice before it is changed. In a loop branch we do not need to change the prediction at the last iteration, thus increasing the speed in case of nested loop such as the ones used to inspect matrices.

5.2.3 Correlating Branch Predictor

The idea behind is that the behaviour of recent branches are correlated, that is the recent behaviour of other branches can influence the prediction of the current branch. In general (m, n) correlating predictor records last m branches to choose from 2^m BHTs, each of which

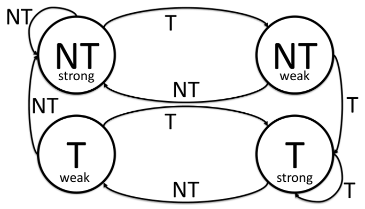
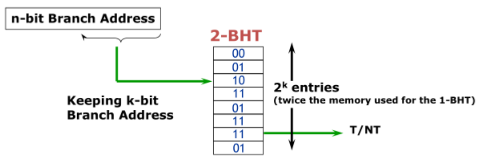


Figure 23: 2-bit BHT automata



n-bit Branch Address
2-BHT
00
01
10
11
01
11
11
01
Keeping k-bit Branch Address
2^k entries
(twice the memory used for the 1-BHT)
T/NT

Figure 24: 2-bit BHT

is a n-bit predictor. The branch prediction buffer can be indexed by using a concatenation of low-order bits from the branch address with m-bit global history (i.e., global history of the most recent m branches).

Example: a (2, 2) correlating predictor with 64 total entries; 6-bit index composed of: 2-bit global history and 4-bit low- order branch address bits.

The (2,2) correlating branch predictor outperforms other predictors such as the 2-bit BHT.

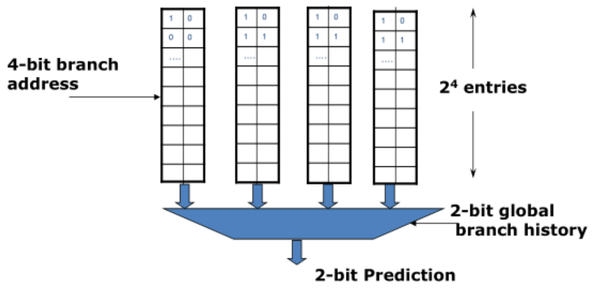


Figure 25: (2,2) correlating branch predictor

6 Instruction level parallelism

ILP: potential overlap of execution among unrelated instructions, possible if:

- no structural hazards
- no raw, waw, war stalls
- no control hazards

$$\text{Pipeline CPI} = \text{Ideal pipeline CPI} + \text{Structural Stalls} + \text{Data Hazards Stalls} + \text{Control Stalls}$$

In the MIPS architecture the only possible structural hazards, WB-ID, where a register in the RF can be both written and read, is solved by splitting the clock cycle in rising edge (WB) and falling edge (ID).

6.1 Complex Pipeline

WAR and WAW were not possible with ADD operation with integers, but MULT and DIV operations use floating

ILP	**PP**
overlap individual machine operations	separate processor getting separate chunks of the program
transparent to the user	nontransparent to the user
goal: speed up	goal: speed up and quality up

Table 1: ILP vs Parallel Processing

point numbers. A new stage introduced, the ISSUE stage, that allows for high performance in presence of:

- long latency or partially pipeline floating-point units
- multiple function and memory units
- memory systems with variable access time
- precise exception

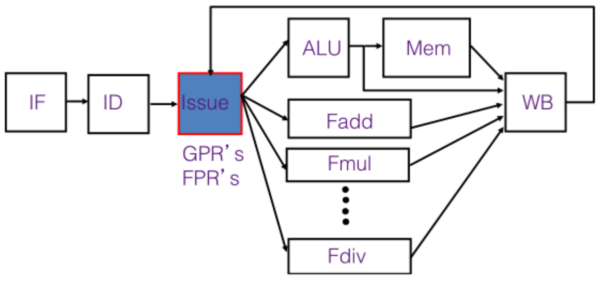


Figure 26: Complex pipeline

New hazards arise from variable latencies of different FUs:

- Structural, stalls arise in:
  - EX stage if some FPU or memory unit is not pipelined and takes more than one cycle.
  - WB stage due to variable latencies of different functional units
- Data
  - WAW and WAR due to variable latencies of the FUs

A solution would be delay the WB in order to have the same latency for all instructions but that would slow down too much single cycle integer operations without forwarding.
Note: in 26 we can see that the MEM stage is absorbed in the EX stage: `load` and `store` instructions first they calculate $offset+base\_address$ and then they access the memory.

When it is safe to issue an Instruction? Things to consider when dispatching an instruction:

- the required FU is available
- the input data is available
- it is safe to write the destination
- there is not a structural conflict at the WB stage

Assumptions Of a general complex pipeline architecture:

- all functional units are pipelined
- registers are read in ISSUE stage: we consider that a register is written (WB) in the first half of the clock cycle and read (IS) in the second half of the clock cycle
- no forwarding
- ALU operations take 1 clock cycle
- FP ALU operations take 2 clock cycles
- memory operations take 2 clock cycles
- write back unit has a single port
- instructions are fetched, decoded and issued in order
- an instruction will only enter the ISSUE stage if it does not cause a WAR or WAW hazard
- only one instruction can be issued at a time, and in the case multiple instructions are ready, the oldest one will go first

To reach higher performance, more parallelism must be achieved, this cannot be done augmenting the CPI of the ideal pipeline because it could create further problems with hazards.
Dependences must be detected and solved, and instructions must be ordered (scheduled) so as to achieve highest parallelism of execution compatible with available resources.
Three types of dependencies:

- data dependencies, RAW
- control dependencies
- name dependencies, two types:
  - antidependencies, WAR
  - output dependencies, WAW

  Generated by the lack of registers.

Dependencies are a property of the program, while hazards are a property of the pipeline.
The main techniques to eliminate dependencies are:

- register renaming
- scheduling
  - static, compiler
  - dynamic, hardware

Steps to exploit more ILP in terms of hardware optimization:

→ sequential
  ↪ pipeline
  single-issue in-order-execution
    ↪ dynamic scheduling
    single-issue out-of-order execution
      ↪ superscalar
      multiple-issue out-of-order execution

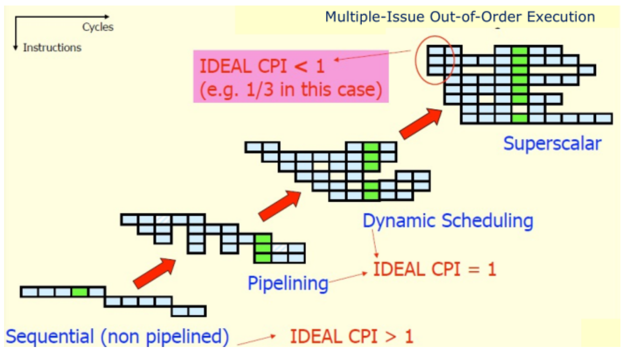


Figure 27: Instruction Level Parallelism

7 Static scheduling

Compilers can use sophisticated algorithms for code scheduling to exploit ILP. The amount of parallelism available within a **basic block**, a straight-line code sequence with no branches in except to the entry and no branches out except at the exit, is quite small (statistically it varies from 4 to 7 instructions).

Data dependence can further limit the amount of ILP we can exploit within a basic block to much less than the average basic block size.

To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks (i.e., across branches).

Static detection and resolution of dependencies are accomplished by the compiler, dependencies are avoided by code reordering.

The output of the compiler is a reordered dependency-free code, which can be processed, for example, by the VLIW (Very Long Instruction Word) processors.

Limits of Static Scheduling

- unpredictable branches
- variable memory latency
- code size explosion
- compiler complexity

Till know we had a single-core multi-cycle parallelism, thanks to the single-issue pipelined architecture. The single-issue architecture means we cannot aim to have an ideal CPI bigger than one.

Multi-issue architectures are the next step to improve the CPI:

- Superscalar
- VLIW

7.1 VLIW architectures

Each instruction word contains more than one operation, the processor can initiate multiple operations per cycle specified completely by the compiler. This leads to a low hardware complexity:

- no scheduling
- reduced support of variable latency instructions
- single control flow (1 PC)
- explicit parallelism

An instruction can be a set of operations that are intended to be issued simultaneously, the VLIW has multiple operations packed into one instruction, and each type of operation has its own slot.

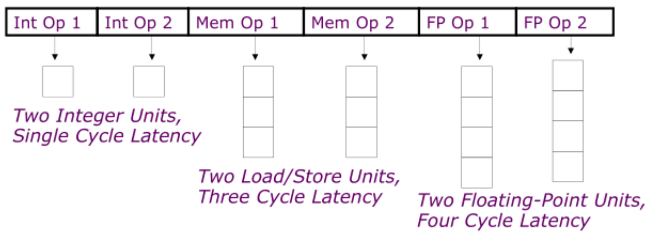

Int Op 1
Int Op 2
Mem Op 1
Mem Op 2
FP Op 1
FP Op 2
Two Integer Units, Single Cycle Latency
Two Load/Store Units, Three Cycle Latency
Two Floating-Point Units, Four Cycle Latency

Figure 28: Very Long Instruction Word

This type of architecture requires guarantees of:

- parallelism within an instruction does not generate RAW data hazard
- no data use before the data is ready, no data interlocks

Compiler responsibilities:

- maximize parallel execution
  - exploit ILP and LLP (Loop Level Parallelism)
  - it is necessary to map the instructions over the machine functional units
  - this mapping must account for time constraints and dependencies among the tasks
- guarantee intra-instruction parallelism
- avoid hazards (no interlocks), typically separates operations with explicit nops
- minimize the total execution time

Pros:

simple HW

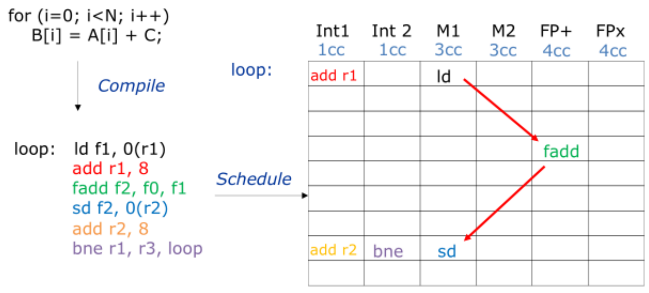
good compilers can effectively detect parallelism

Cons:

- huge number of registers to keep active the FUs
  - needed to store operands and results
- large data transport capacity between
  - FUs and register files
  - Register files and Memory
- high bandwidth between i-cache and fetch unit
- large code size
  - use of (big) nops in the VLIW
  - unpredictable branches have different optimal schedules that varies with branch path
- knowing branch probabilities requires significant extra steps in build process due to profiling

An example of scheduling of a loop in VLIM architecture:

operations	ls, sd	add, bne	fadd
clock cycles	3	1	4



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

loop:  ld f1, 0(r1)
       add r1, 8
       fadd f2, f0, f1
       sd f2, 0(r2)
       add r2, 8
       bne r1, r3, loop
```

Int1	Int 2	M1	M2	FP+	FPx
add r1		ld			
				fadd	
add r2	bne	sd			

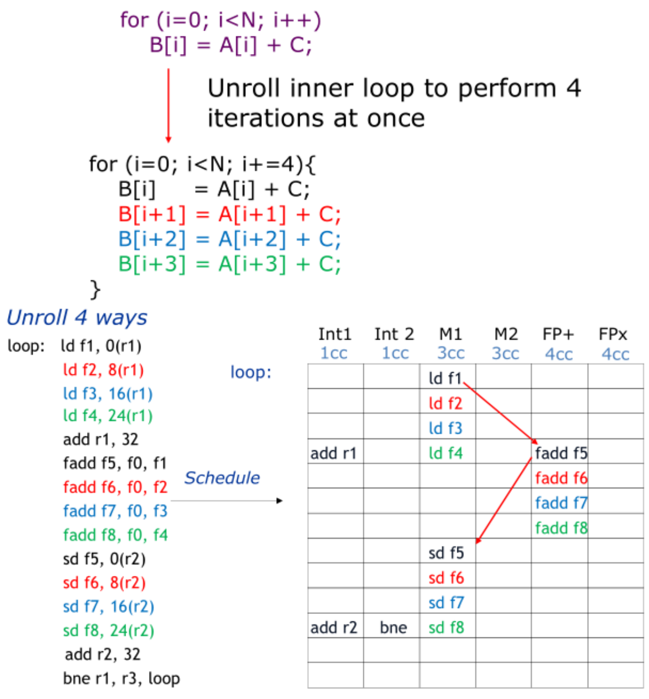
Figure 29: Loop execution on VLIM

Many blank lines in the scheduler corresponds to nops, leading to $\frac{1 \text{ FP ops}}{8 \text{ cycles}} = 0,125$, which is far from our ideal CPI > 1. To increase parallelism we use loop unrolling.

7.2 Loop unrolling

The idea is that we can extend the loop body as to include a finite number of subsequent iterations of the loop, increasing the amount of available ILP. Unrolling simply replicates the loop body multiple times, adjusting the loop termination code, by doing so the loop *overhead* decrease relatively to the loop *body*.

$$\text{Loop} = (4 \times \textit{loop body} + \text{loop overhead}) \times \frac{n}{4} \text{ iterations}$$



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

for (i=0; i<N; i+=4){
    B[i]     = A[i] + C;
    B[i+1] = A[i+1] + C;
    B[i+2] = A[i+2] + C;
    B[i+3] = A[i+3] + C;
}
```

```
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       sd f8, 24(r2)
       add r2, 32
       bne r1, r3, loop
```

Int1	Int 2	M1	M2	FP+	FPx
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4		fadd f5	
				fadd f6	
				fadd f7	
				fadd f8	
		sd f5			
		sd f6			
		sd f7			
add r2	bne	sd f8			

Figure 30: Loop unrolling

Note that non FP operations can have multiple schedule locations, e.g. (see 30), the `ld f2` could be scheduled also in the first cycle to M2, but if does not decrease the CPI, it is recommended using a new instruction. In this case we end up with $\frac{4 \text{ FP ops}}{11 \text{ cycles}} \approx 0,36$, more than doubling the previous result.

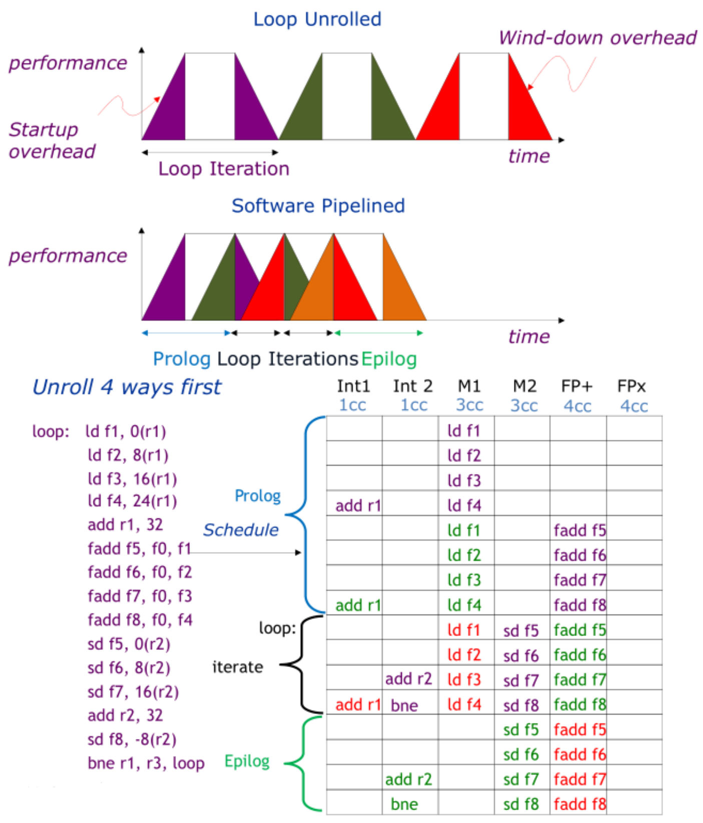
The **limits** of loop unrolling:

code size

number of register

That is, we have an upper bound to the number of replications of the loop body, which has to be considered constant in regard to the $\theta(n)$ loop iterations: we reduced the *prolog* and *epilog* of the loop by a constant.

We can optimize the results of loop unrolling with **software pipelining**, further increasing the ILP.



```
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       add r2, 32
       sd f8, -8(r2)
       bne r1, r3, loop
```

Int1	Int 2	M1	M2	FP+	FPx
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4			
		ld f1		fadd f5	
		ld f2		fadd f6	
		ld f3		fadd f7	
add r1		ld f4		fadd f8	
		ld f1	sd f5	fadd f5	
		ld f2	sd f6	fadd f6	
	add r2	ld f3	sd f7	fadd f7	
add r1	bne	ld f4	sd f8	fadd f8	
			sd f5	fadd f5	
			sd f6	fadd f6	
	add r2		sd f7	fadd f7	
	bne		sd f8	fadd f8	

Figure 31: Loop unrolled vs Software pipelined

$$\text{SW Pipelined Loop} = \text{prolog} + n \textit{ iterations} + \text{epilog}$$

In this case we have just one prolog and one epilog for all the loop iterations and can be considered $O(1)$ with respect to the n iterations. For this reason we consider only the *iterate* part for our performance evaluation which is $\frac{4 \text{ FP ops}}{4 \text{ cycles}} = 1$.

Note: in case of short loops, loop unrolling looses performance due to the costs of starting and closing the iterations.

7.3 Trace Scheduler

Loop unrolling is useful but does not cover other type of branches that limit the basic **block size** in control-flow intensive irregular code. In these cases is difficult to find ILP in individual basic blocks.

Trace scheduling focus on traces:

- a loop-free sequence of basic blocks embedded in the control flow graph
- it is an execution path which can be taken for some set of inputs
- the chances that a trace is actually executed depends on the input set that allows its execution

Note: a trace may include branches but not loops.

Algorithm Idea, some traces are executed much more frequently than others:

- pick string of basic blocks, a trace, that represents most frequent branch path
- use profiling feedback or compiler heuristics to find common branch paths
- schedule whole "trace" at one
- add fixup code to cope with branches jumping out of trace

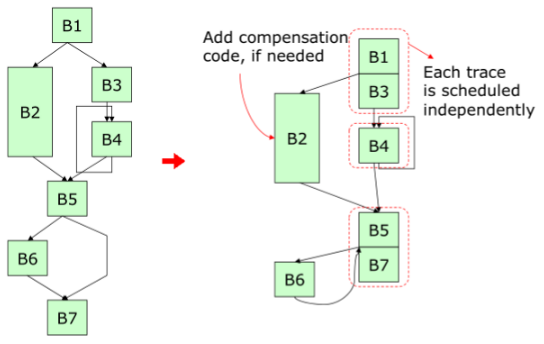

B1
B2
B3
B4
B5
B6
B7
Add compensation code, if needed
Each trace is scheduled independently

Figure 32: Trace scheduling

For example we suppose that {B1, B3, B4, B5, B7} is the most frequently executed path, therefore the traces are {B1, B3}, {B4} and {B5, B7}.
Note: Traces are scheduled as if they were basic blocks, by removing control hazards we increase ILP.

Problems Compensation codes are difficult to generate, especially entry points (of a different path). In addition to need of compensation codes there are restrictions on movement of a code in a trace:

- **dataflow** of the program must not change, it is guaranteed to be correct by maintaining:
  - data dependencies
  - control dependencies
- the exception behaviour must be preserved

Solutions There are two approaches to eliminate control dependency:

- use of **predicate** instructions (Hyperblock scheduling). Useful for mispredicted branches that limits ILP.
- use of **speculative** instructions (Speculative Scheduling), and speculatively move an instruction before the branch. Useful for scheduling long latency loads early.

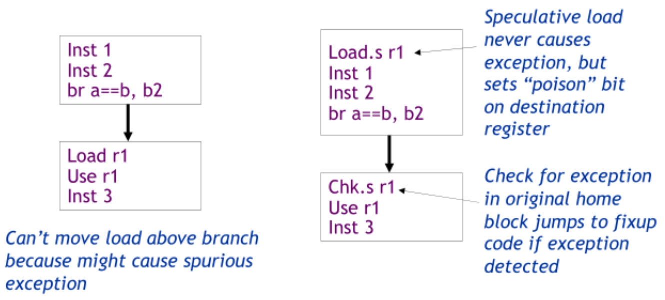

Inst 1
Inst 2
br a==b, b2
Load r1
Use r1
Inst 3
Can't move load above branch because might cause spurious exception
Load.s r1
Inst 1
Inst 2
br a==b, b2
Chk.s r1
Use r1
Inst 3
Speculative load never causes exception, but sets "poison" bit on destination register
Check for exception in original home block jumps to fixup code if exception detected

Figure 33: Speculative instruction

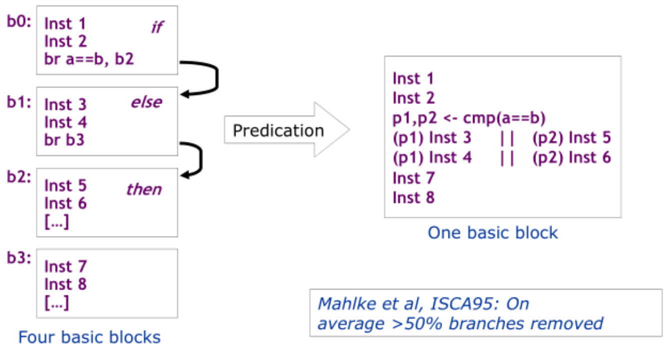


Figure 34: Predicated execution

7.4 Rotating Register Files

Scheduled loops require lots of registers, lots of duplicated code in prolog and epilog. **Solution**: allocate new set of registers for each loop iteration.

Rotating Register Base (RRB) register points to base of current register set. A value is added to the *logical* register specifier to give *physical* register number.

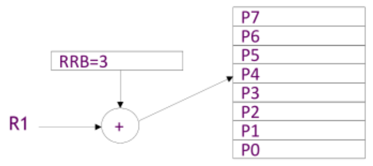


Figure 35: Rotating Register Base

If we know the clock cycles that each operation takes we can use rotate new register by

$$\#logical\ R = \#physical\ R + (\#\text{iteration} \mod \#\text{clock cycles required})$$

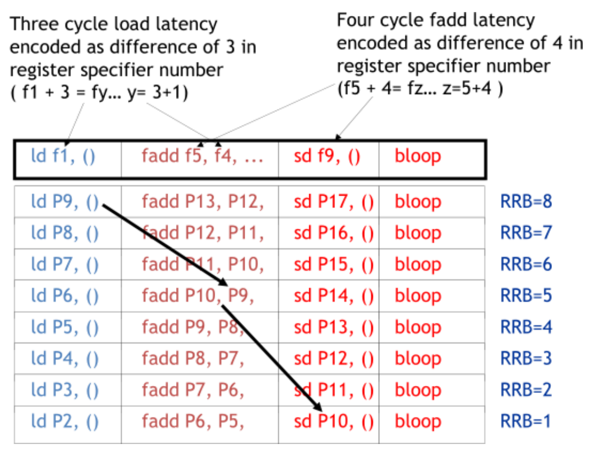


Figure 36: Rotating registers

8 Dynamic Scheduling

The hardware reorders the instruction execution to reduce pipeline stalls while maintaining data flow and exception behavior.

Main advantages

It enables handling some cases where dependencies are unknown at compile time

It simplifies the compiler complexity

It allows compiled code to run efficiently on a different pipeline.

Costs Advantages are gained at a cost:

A significant increase in hardware complexity

Increased power consumption

Could generate imprecise exception

In-order Issue Instructions are fetched and issued in program order (in-order-issue). The execution begins as soon as operands are available, possibly, out of order execution.
Note: possible even with pipelined scalar architectures.

Out-of-order Execution Out-of order execution introduces possibility of WAR, WAW data hazards. It implies out-of-order completion unless there is a re-order buffer to get in-order completion.

Example

```
DIVD F0,  F2, F4
ADDD F10, F0, F8
SUBD F12, F8, F14
```

→RAW on F0, causes the `SUBD` to stall without dynamic scheduling.

Note: As mentioned before, here we are talking strictly about the hardware, there could be software pipelining.

8.1 Scoreboard

Suppose a data structure keeps track of all the instructions in all the functional units. The following checks need to be made before the Issue stage can dispatch an instruction:

- is the required function available?
- is the input data available? (RAW?)
- is it safe to write the destination? (WAR? WAW?)

Data structure The instruction i at the Issue stage consults this table

FU available? check the busy column

RAW? search the dest column for i's sources

WAR? search the source columns for i's destination

WAW? search the dest column for i's destination

Name	Busy	Op	Dest	Src1	Src2
Int					
Mem					
Add1					
Add2					
Add3					
Mult1					
Mult2					
Div					

Table 2: Example of data structure for correct issues

CDC6600 Scoreboard

In-order issue Instructions dispatched in-order to functional units provided no structural hazard or WAW:

- stall on structural hazard, no functional units available
- only one pending write to any register

Out-of-order read & execution Instructions wait for input operands in case of RAW hazards. Once they read operands can execute out-of-order.

Out-of-order write Instructions wait for output register to be read by preceding instructions in case of WAR hazard, the result is held in functional unit until register is free, that is right after the register causing the hazard is read.

Centralized hazard management Every instruction goes through scoreboard:

- it determines when the instruction can **read** its operands and **begin execution**
- it monitors changes in hardware and decides when a stalled instruction can **resume execution**
- it controls then instruction can **write** results

8.1.1 Scoreboard Pipeline

New pipeline ID stage split in two parts:

ID		EX	WB
Issue	Read Regs	Execution	Write

Table 3: Scoreboard Pipeline

Issue Decode and check structural hazard.
Instructions issued in program order (for hazard checking).
If a functional unit for the instruction is free and no other active instruction has the same destination register (WAW), the scoreboard issues the instruction to the functional unit and updates its internal data structure.
If a structural or a WAW hazard exists, then the instruction issue stalls, and no further instructions will issue until these hazards are cleared.

Read Operands Wait until there are no data hazards, then read operands.
A source operand is available if:

- no earlier issue active instructions will write it
- a functional unit is writing its value in a register

When the source operands are available, the scoreboard tells the functional unit to proceed to read the operands from the registers and begin execution. RAW hazards are resolved dynamically in this step, and instructions may be sent into execution out-of-order. No forwarding of data in this model.

Execution Operate on operands.
The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard that it has completed execution. FUs are characterized by:

- latency (the effective time used to complete one operation)
- Initiation interval (the number of cycles that must elapse between issuing two operations to the same functional unit).

Write result Finish execution.
Once the scoreboard is aware that the functional unit has completed execution, the scoreboard checks for WAR hazards: if none, it writes results. If there's a WAR, then it stalls the instruction. Assume we can overlap issue and write.

Scoreboard implications Solution for WAW: – Detect hazard and stall issue of new instruction until the other instruction completes. No register renaming. Need to have multiple instructions in execution phase —> Multiple execution units or pipelined execution units. Scoreboard keeps track of dependencies and state of operations.

Example

```
DIVD  F0,  F2,  F4
ADDD  F6,  F0,  F8
SUBD  F8,  F8,  F14
MULD  F6,  F10, F8
```

SUBD in the WB stage, waiting that ADDD reads F0 and F8 and MULD in the issue stage until ADDD writes F6. Cannot be solved without register renaming.

8.1.2 Scoreboard Structure

Instruction Status Tells witch instruction is being executed.

FU Status Indicates the state of the functional unit (FU):

Busy – Indicates whether the unit is busy or not
Op - The operation to perform in the unit (+,-, etc.)
Fi - Destination register
Fj, Fk – Source register numbers
Qj, Qk – Functional units producing source registers (RAW)
Rj, Rk – Flags indicating when Fj, Fk are ready (WAR)

Register Result Status Indicates which functional unit will write each register. Blank if no pending instructions will write that register.

8.2 Tomasulo

Distributed management control The control logic and the buffers are distributed with FUs (vs centralized in scoreboard).

8.2.1 Reservation Stations

Operand buffers are called Reservation Stations. Each instruction is an entry of a reservation station. Compo-

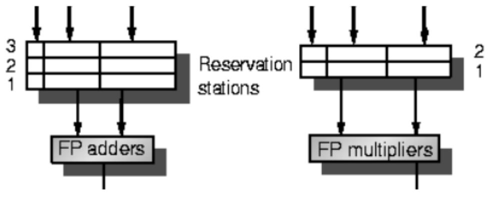


Figure 37: Reservation stations (RS)

nents:

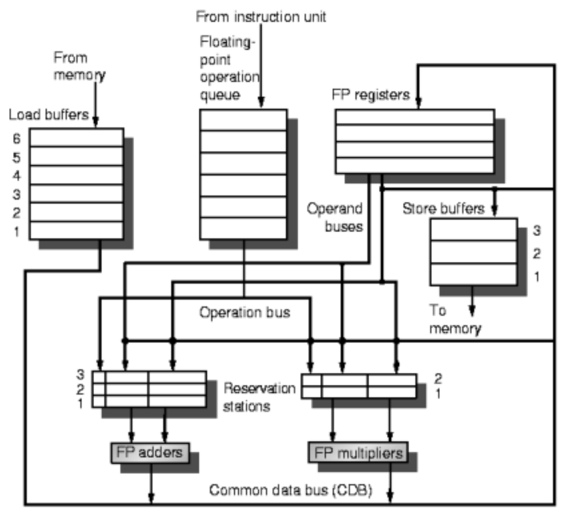


Figure 38: Tomasulo architecture

Tag	identifying the RS
OP	Operation to perform on the component.
Vj,Vk	Value of the source operands.
Qj,Qk	Pointers to RS that produce Vj,Vk. Zero value = Source op is already available in Vj or Vk.
Busy	Indicates RS Busy.

Note: Only one of V-field or Q-field is valid for each operand.

Reservation stations allow for:

→ Register Renaming: the operands of the RSs are replaced by values or pointers:

  – avoid WAR and WAW hazards

  – # reservation stations > # registers, make possible better optimization than a compiler

→ Loop unrolling in hardware: permit instruction issue to advance past integer control flow operations (thanks to register renaming)

→ Distribute management control

8.2.2 Common data bus

Results are dispatched to other FUs through a Common Data Bus (CDB). It offers a *serialized* access to write back. It provides values before they are saved into registers, that is data forwarding: it makes a point-to-point connection between the FU providing the result and the FU(s) waiting for input. The `load` and `store` operations are treated as FUs.

8.2.3 Other Components

The Register File and the Store Buffer have a *value* (V) and a *pointer* (Q) field.
Q corresponds to the reservation station(s) producing the result to be stored in RF or store buffer. If zero (or `null`), there are no active instructions producing the result (RF or store buffer content has the correct value).

	V	**Q**
F0		
F1		
...		

Table 4: Register File - RF

Load and store buffers have an *address* (A) field, and a *busy* field.

	busy	**A**
Load1		
Load2		

Table 5: Load buffer

	busy	**A**	**Q**
Store1			
Store2			

Table 6: Store buffer

Note: The *address* field holds info for memory address calculation for load/store. Initially contains the instruction offset (immediate field), after address calculation stores the effective address.

8.2.4 Tomasulo Stages

Issue Get an instruction I from the queue.
If it is an FP op check if a RS is empty (i.e., check for structural hazards).
If it is a `ld` or `sd` check if the corresponding buffers have a free spot.
Rename registers:

→ WAR resolution
If I writes Rx, read by an instruction K already issued, K knows already the value of Rx or knows what instruction will write it. So the RF can be linked to I (with Q pointer).

→ WAW resolution
Since we use in-order issue, the RF can be linked to I

Execution When both operands are ready then execute.
If not ready, watch the CDB for results: delay execution until operands are available, RAW hazards are avoided.
Note: multiple instructions could become ready in the same clock cycle for the same FU, start execution only if there's a free FU.
Load and stores, two-step process:

1.a-b compute effective address, place it in load or store buffer (in A field).

2.a Loads in Load Buffer execute as soon as memory unit is available.

2.b Stores in store buffer wait for the value pointed by Q before being sent to memory unit.

Write When result is available, write on Common Data Bus and from there into RF and into all RSs (including store buffers) waiting for this result then mark reservation stations available.
Note: In the general case of concurrent writes, choose first the one that belongs to the critical path.

8.2.5 Tomasulo in detail

All writes occur in Write Result, simplifying Tomasulo algorithm.

A Load and a Store can be done in different order, provided they access different memory locations. Otherwise, a WAR (interchange load-store sequence) or a RAW (interchange store-load sequence) may result (WAW if two stores are interchanged). Loads can be reordered freely.
Note: registers are renamed but memory locations remain the same.

To detect such hazards: data memory addresses associated with any earlier memory operation must have been computed by the CPU (i.e., address computation executed in program order)

Load executed out of order with previous store: assume address computed in program order. When Load address has been computed, it can be compared with A fields in active Store buffers: in the case of a match, Load is not sent to Load buffer until conflicting store completes.

Stores must check for matching addresses in both Load and Store buffers (dynamic disambiguation, alternative to static disambiguation performed by the compiler)

Drawback: amount of hardware required.

Each RS must contain a fast associative buffer; single CDB may limit performance.

8.2.6 Tomasulo Drawbacks

- complexity
- large amount of hardware
- many associative store (CDB) at high speed
- performance limited by CDB

Multiple CDBs $\Longrightarrow$ more FU logic for parallel associative stores.

Scoreboard vs Tomasulo

→ structural hazards in scoreboard

→ lack of forwarding in scoreboard

9 Exception handling

The **interrupt** is an exception, cause by an internal or external event which alters the normal flow of control: this must be processed by another program, the interrupt handler.

9.1 Causes

Asynchronous, external events:

- I/O device service-request
- timer expiration
- power disruptions, hw failure

Synchronous internal event:

- undefined opcode, privileged instruction
- arithmetic overflow, FPU exception
- misaligned memory access
- virtual memory exceptions: page faults, TLB miss, protection violations
- traps: system calls, e.g., jumps into kernel

9.2 Exception classes

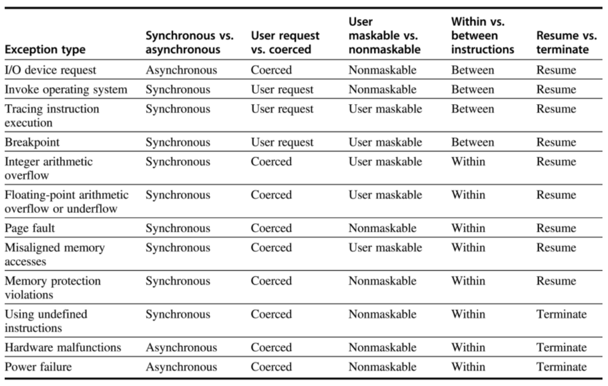
Sync vs Async async caused by devices external to the CPU and memory and can be handled easily

User request vs coerced user requested are predictable: treated as exceptions because they use the same mechanism that are used to save and restore the state handled after the instruction has completed. Coerced are caused by some HW event not under control of the program.

User maskable vs user nonmaskable The mask simply controls whether the hardware responds to the exception or not.

Within vs between instructions Exceptions that occur within instructions are usually synchronous since the instruction triggers the exception. The instruction must be stopped and restarted.
Asynchronous that occur between instructions arise from catastrophic situations and cause program termination.

Resume vs terminate Terminating event: program's execution always stops after the interrupt. Resuming event: program's execution continues after the interrupt.



Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory protection violations	Synchronous	Coerced	Nonmaskable	Within	Resume
Using undefined instructions	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunctions	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

Figure 39: Exception classes

9.3 Interrupt handler

When an I/O device requests attention, by asserting one of the prioritized interrupt request lines, the CPU, if possible, invokes the interrupt handler. When the processor decides to process the interrupt:

- It stops the current program at instruction I, completing all the previous instructions (precise interrupt)
- It saves the PC of instruction I in a special register (EPC)
- It disables interrupts and transfers control to a designated interrupt handler running in the kernel mode.

The cpu we have two modes: user-mode and kernel-mode.

The **interrupt handler** saves PC before allowing nested interrupts. Needs to read a *status register* that indicates the cause of the interrupt. Uses a special indirect jump instruction RFE (Return-From-Exception) which:

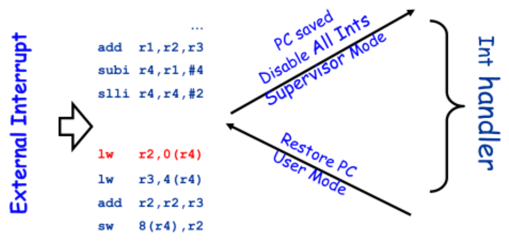
- enables interrupts
- restores the processor to the user mode
- restores hardware status and control state

9.4 Synchronous interrupts

A sync interrupt is caused by a particular instruction. In general, the instrucion cannot be completed and needs to restarted after the exception has been handled. In case of a system call trap, the instruction is considered to have been completed.

9.5 Precise Interrupts/Exceptions

An interrupt or exception is considered precise if there is a single instruction (or interrupt point) for which all instructions, before that instruction, have committed their state and no following instructions including the interrupting instruction have modified any state. This means, effectively, that you can restart execution at the interrupt point.



```
        ...
add  r1,r2,r3
subi r4,r1,#4
slli r4,r4,#2

lw   r2,0(r4)
lw   r3,4(r4)
add  r2,r2,r3
sw   8(r4),r2
```

Figure 40: Precise interrupt

10 HW speculation

HW-based speculation combines three ideas:

Dynamic Branch Prediction to choose which instruction to execute

Dynamic Scheduling supporting out-of-order execution but in-order commit to prevent any irrevocable actions (such as register update or taking exception) until an instruction commits

Speculation To execute instructions before control dependencies are solved

The idea is allowing instruction to execute freely and out-of-order, based on speculation, but allow to update the RF or the memory only when an instruction is no longer speculative.

Mechanisms are necessary to handle incorrect speculation, hardware speculation extends dynamic scheduling beyond a branch, i.e., behind the basic blocks.

10.1 Reorder buffer

A buffer that holds instruction results before they are committed. When an instruction completes execution, the results are placed into ROB, which holds the instructions in FIFO order, exactly as issued.

The result in the ROB buffer are tagged, they are used instead of the reservation stations to supply operands to other instruction between execution and commit.
The instruction at the **head** of ROB can safely commit, all its predecessor have already committed.

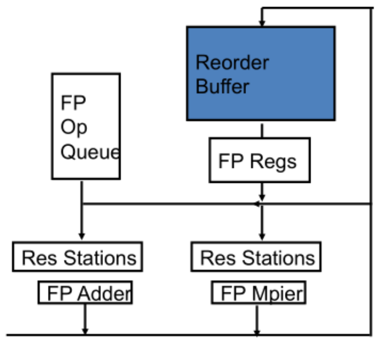


Figure 41: ROB

Separating Completion from Commit Re-order buffer holds the register result from completion until commit:

- entries are allocated in program order during decode
- it buffers *completed values* and *exception state* until commit point
- completed values can be used by dependents before committed (bypassing)
- each entry holds a *PC*, *instruction type*, *destination register* specifier and *value* and, if any, *exception status*.

Memory reordering It needs special data structures: speculative store address and data buffers, speculative load address and data buffers.

Precise interrupts and Speculation If a speculation is wrong, we need to be able to back and restart execution to a point before out incorrect prediction (e.g., branch prediction), it happens the same as precise exceptions. The recovery technique for both is **in-order commit**.

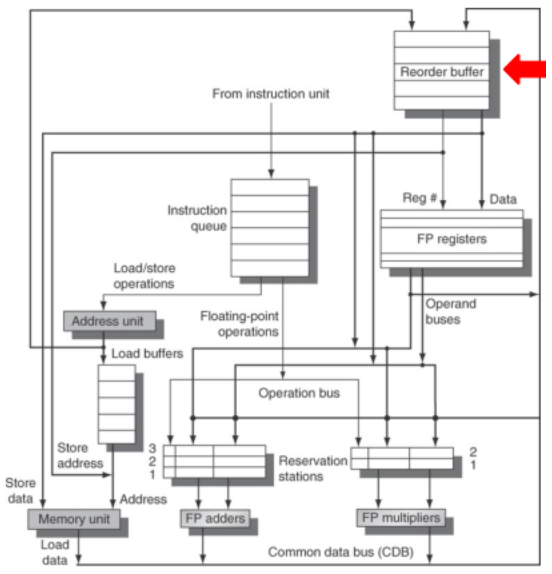


Figure 42: Speculative tomasulo

10.2 Speculative Tomasulo

10.2.1 Stages

Issue get instruction from FP Op Queue.
If the reservation station and reorder buffer slot are free: issue instruction, send operands, reorder buffer number for destination.

Execution operate on operands when ready.
If not ready, watch CDB for result: when both are in the reservation station, execute then check for RAW.

Write result finish execution.
Write on Common Data Bus to all awaiting FUs and reorder buffer. Mark the reservation station available.

Commit update register with reorder result.
When and instruction at the head of ROB and the result is present, update the register or memory, and remove instruction from ROB. In case of mispredicted branch flush the reorder buffer.
Three types of commit:

- normal commit
- store commit
- flush results if incorrect branch prediction

10.2.2 ROB entries

Each ROB entries contains four fields:

Instruction type field indicates whether instruction is a branch (no destination result), a store (has memory address destination), or a load/ALU (register destination)

Destination field supplies register number (for loads and ALU instructions) or memory address (for stores) where results should be written

Value field holds value of result until instruction commits

Ready filed indicates that instruction has completed execution, ready to commit

		I type	**dest**	**value**	**ready**
head =>	1	ADDD	F4	2.0	yes
	2	MULD	F8	-	no
	3	ADDD	F6	-	no
	4	SUBI	F0	1.0	yes

Table 7: ROB example

The register file associated with this ROB:

	V	**Q**
F0	1.0	rob4
F2	5.5	-
F4	3.2	rob1
F6	4.0	rob3
F8	0.0	rob2

Table 8: Register File - RF

Since the head of ROB is ready we can commit and update the tables:

		I type	**dest**	**value**	**ready**
	1	-	-	-	-
head =>	2	MULD	F8	-	no
	3	ADDD	F6	-	no
	4	SUBI	F0	1.0	yes

	V	**Q**
F0	1.0	rob4
F2	5.5	-
F4	**2.0**	-
F6	4.0	rob3
F8	0.0	rob2

10.2.3 ROB extension

The addition of the reorder buffer introduces changes in tomasulo:

- ROB completely replaces store buffers
- renaming function of reservation stations completely replaced
- reservation stations now only queue operations (and operands) to FUs between issue and execution.
- results are tagged with ROB entry number rather than with RS number, ROB entry must be tracked in the reservation stations
- all instructions excluding incorrectly predicted branches (or incorrectly speculated loads) commit when reaching head of ROB
- when a incorrectly predicted branch reaches the head, ROB is flushed, execution restarts at correct successor of branch $\rightarrow$speculation is easily undone
- processor with ROB can dynamically speculate while maintaining a precise interrupt model

11 Explicit register renaming

Tomasulo provides *implicit* register renaming by the means of reservation stations or ROB. Now we introduce *explicit* register renaming: use a physical register file that is larger than the number of registers specified by the ISA.

Idea: allocate a new physical destination register for every instruction that writes:

- removes WAW, WAR hazards
- allows out-of-order completion (like tomasulo)
- similar to SSA, Static Single Assignment transformation done the compiler

Mechanism Keep a translation table:

- map ISA *logical* register to physical register
- when a register is written, replace the map entry with new register from freelist
- physical register becomes free when not used by any active instructions

Unified physical register file

- rename all architectural (logical) registers into a single physical register file during decode, no register values read
- FUs read and write from single unified register file holding committed and temporary registers in execution
- commit only updates mapping of architectural register to physical register, no data movement

HW register renaming

Renaming map simple data structure that supplies the physical register number of the register that currently corresponds to the requested architectural register

Instruction commit update permanently the renaming table to indicate that the physical register holding the destination values corresponds to the actual architectural register

ROB Use reorder buffer to enforce in-order commit

11.1 Explicit renaming - Scoreboard

11.1.1 Stages

Issue decode instructions and check for structural hazards and allocate new physical register for result:

- instructions issued in program order (for hazard checking)
- don't issue if there are no free physical registers
- don't issue if there's a structural hazard

Read operands wait until there are no more hazards, then read operands. All real dependencies solved in this stage, since we wait for instructions to write data back.

Execution operate on operands. The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard.

Write result finish execution

Note: no check for WAR and WAW hazards.

11.2 Register renaming vs ROB

- instruction commit simpler than with ROB
- deallocating register more complex
- dynamic mapping or architectural to physical registers complicates design and debugging.

12 ILP limits

12.1 Superscalar

Superscalar execution allows multiples-issue and out-of-order execution. A superscalar processor can execute more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to different execution units on the processor. It therefore allows more throughput than would otherwise be possible at a given clock rate. Each execution unit is not a separate processor (or a core if the processor is a multicore processor), but an execution resource within a single CPU such as an arithmetic logic unit. In Flynn's taxonomy, a single-core superscalar processor is classified as an SISD processor, though a single-core superscalar processor that supports short vector operations could be classified as SIMD (single instruction stream, multiple data streams). A multicore superscalar processor is classified as an MIMD processor (multiple instruction streams, multiple data streams). While a superscalar CPU is typically also pipelined, superscalar and pipelining execution are considered different performance enhancement techniques. The former executes multiple instructions in parallel by using multiple execution units, whereas the latter executes multiple instructions in the same execution unit in parallel by dividing the execution (instruction) unit into different phases.

Key requirements

- Fetching more instructions per clock cycle: no major problem provided the instruction cache can sustain the bandwidth and can manage more requests at the same time.
- Decide on data and control dependencies: dynamic scheduling and dynamic branch prediction

Beyond CPI = 1

- issue multiple instruction per clock-cycle
- varying number of instruction per cycle (1 to 8)
- scheduled by the compiler or HW

$$CPI_{ideal} = \frac{1}{issue - width}$$

With single-issue ILP we can be reach an ideal CPI = 1. With superscalar execution our *ideal* CPI can decrease furthermore.

12.2 Assumptions

The ideal machine must have:

Register renaming infinite virtual registers and all WAW and WAR hazards are avoided

Branch prediction perfect, no mispredictions

Jump prediction all jumps perfectly predicted, machine with perfect speculation and an unbounded buffer of instructions available

Memory address alias analysis [1] addresses are know and a store can be moved before a load provided the addresses not equal

One cycle latency for all instructions unlimited number of instructions issued per clock cycle

12.3 Limits on window size

Dynamic analysis is necessary to approach perfect branch prediction (impossible at compile time!). A perfect dynamic-scheduled CPU should:

- Look arbitrarily far ahead to find set of instructions to issue, predict all branches perfectly
- Rename all registers uses (no WAW, WAR hazards);
- Determine whether there are data dependencies among instructions in the issue packet, rename if necessary
- Determine if memory dependencies exist among issuing instructions, handle them
- Provide enough replicated functional units to allow all ready instructions to issue.

Size affects the number of comparisons necessary to determine RAW dependencies. The number of comparisons to evaluate for data dependencies among n register-to-register instructions in the issue phase is $n^2 - n$.

13 Multiprocessors

Till now we considered single cores architectures and how to increase the ILP in(side) such architectures.

Beyond ILP We have seen the superscalar architecture, which can issue multiple instructions per clock cycle but these systems are very complex to design and the returns are diminishing. Another step forward towards parallelism is adding multithreading (or multiprocess) support to our single cores, but the throughput of such systems reach their limits soon in terms of number of threads, what if we are willing to have much more threads running? Can we increase our throughput and support large-scale parallel systems?

[1]Alias analysis is a technique used to determine if a storage location may be accessed in more than one way

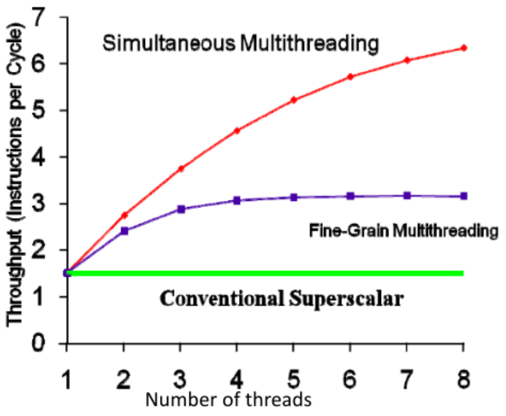


Figure 43: Superscalar vs multithreading

Physical limitations

- difficult to increase performance and clock frequency of the single core
- a deep pipeline:
  - heat dissipation
  - speed light transmission problems in wires
  - difficulties in design and verification
  - requirement of very large design groups (higher complexity)

Designing faster architectures may lead to worse throughput than running in parallel existing architectures; furthermore, the design of a new architecture takes more time and cost. Replication of cores doesn't come for free, to connect multiple microprocessor in a complex system we have to implement some communication policies between cores.

Communication architecture requires:

- abstractions (HW/SW interfaces)
- different structures to realize abstraction efficiently

13.1 Parallel architectures

"A parallel computer is a collection of processing elements that cooperates and communicate to solve large problems fast"[2]

13.1.1 SISD

A serial (non-parallel) computer with deterministic execution, i.e., the architectures seen till now, the oldest and most common type of computer.

13.1.2 MISD

Another parallel computer, where a single data stream is fed into multiple processing units. Each processing unit operates on the data independently via independent instruction streams. Niche use cases such as regex expressions.

[2]Almasi and Gottlieb, Highly Parallel Computing, 1989

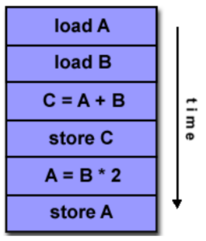


Figure 44: SISD

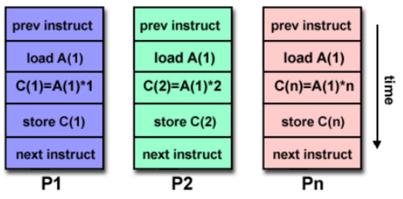


Figure 45: MISD

13.1.3 SIMD

A type of parallel computer, where all processing units execute the same instruction at any given clock cycle, each one operates on a different data stream. Data pipelining into processing unit, useful for example for operations on matrices. Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing (GPUs) or machine learning. Processors are typically special-purpose. There is only one single instruction memory and control processor to fetch and dispatch instructions.

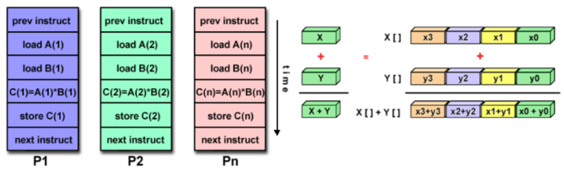


Figure 46: SIMD

A central controller broadcasts instructions to multiple processing elements (PEs):

- only requires one controller for whole array
- only requires storage for one copy of program
- all computations are fully synchronized

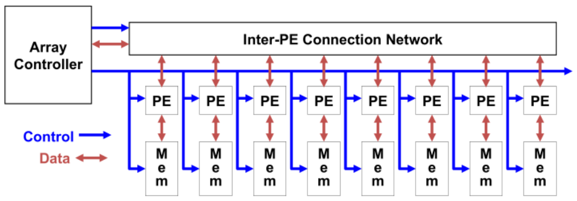


Figure 47: SIMD scheme

→ Synchronized units: single Program Counter

→ Each unit has its own addressing registers
 – Can use different data addresses

→ Motivations for SIMD:
 – Cost of control unit shared by all execution units
 – Only one copy of the code in execution is necessary

→ Real life:
 – SIMD have a mix of SISD instructions and SIMD
 – A host computer executes sequential operations
 – SIMD instructions sent to all the execution units, which has its own memory and registers and exploit an interconnection network to exchange data

Alternative Model: Vector Processing Although being similar to SIMD, cannot be categorized under the Flynn's taxonomy. SIMD instruction sets lack crucial features when compared to vector processor instruction sets, the most important of these is that vector processors, inherently by definition and design, have always been variable-length since their inception.

Vector processors have high-level operations that work on linear arrays of numbers: "vectors". A vector processor consists of a pipelined scalar unit (may be out-of order or VLIW) plus a vector unit.

Styles of vector architectures:

- *memory-memory vector processors:* all vector operations are memory to memory
- *vector-register processors:* all vector operations between vector registers (except load and store) , vector equivalent of load-store architectures

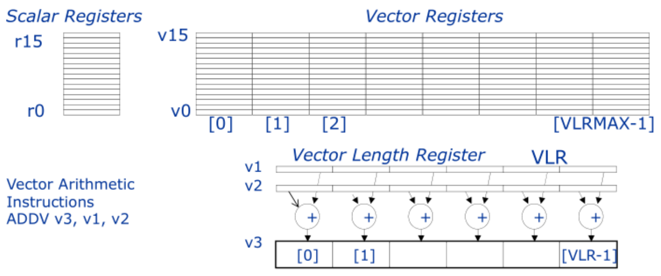


Figure 48: Vector programming model

add stuff from slide +
vector registers, more area more cost and more energy
vector machina distance between functional units can be bottleneck, use 3d structure.
situation: tons of data that we want to study

Sony Playstation 2000: a system running in parallel a MIPS cpu for I/O together with parallel vector processors for image pre-processing.

13.1.4 MIMD

The most common parallel computer nowadays, every processor may have a different instruction and data stream. See chapter 15.

```
# C code
for (i=0;i<64; i++)
C[i] = A[i]+B[i];
```

```
# Scalar Code
LI R4, #64
loop:
L.D F0, 0(R1)
L.D F2, 0(R2)
ADD.D F4, F2, F0
S.D F4, 0(R3)
DADDIU R1, 8
DADDIU R2, 8
DADDIU R3, 8
DSUBIU R4, 1
BNEZ R4, loop
```

```
# Vector Code
LI VLR, #64
LV V1, R1
LV V2, R2
ADDV.D V3,V1,V2
SV V3, R3
```

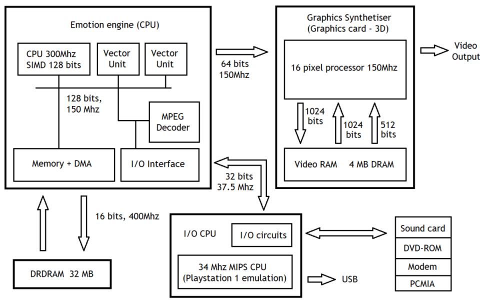


Figure 49: Sony Playstation 2000 architecture

14 GPU

14.1 Procedural synthesis

Procedural synthesis about making optimal use of system bandwidth and main memory by dynamically generating lower-level geometry data from statically stored higher-level scene data.

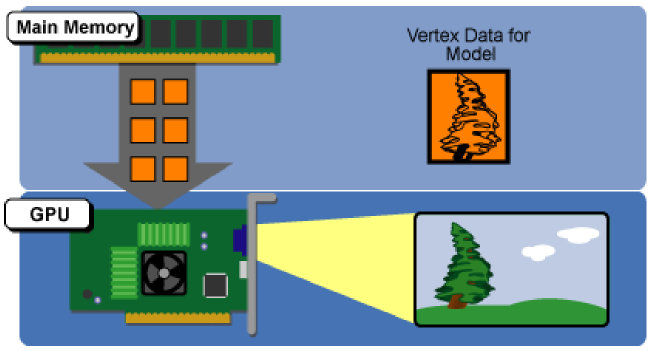


Figure 50: Procedural synthesis

For 3D games:

- Artists use a 3D rendering program to produce content for the game
- Each model is translated into a collection of polygons
- Each polygons is represented in the computer's memory as collections of vertices

When the computer is rendering a scene in a game in real-time:

- Models that are being displayed on the screen start out in main memory as stored vertex data
- That vertex data is fed from main memory into the GPU where it is then rendered into a 3D image and output to the monitor as a sequence of frames.

Limitations there are two problems:

- The costs of creating art assets for a 3D game are going through the roof along with the size and complexity of the games themselves
- Console hardware's limited main memory sizes and limited bus bandwidth

Xbox 360's solution Store high-level descriptions of objects in main memory. Give the CPU the task of procedurally generate the geometry of the objects on the fly – e.g., the vertex data are generated by one or more running threads. The GPU then takes the pre-processed vertex information and renders the tress normally, just as if it had gotten that information from main memory.

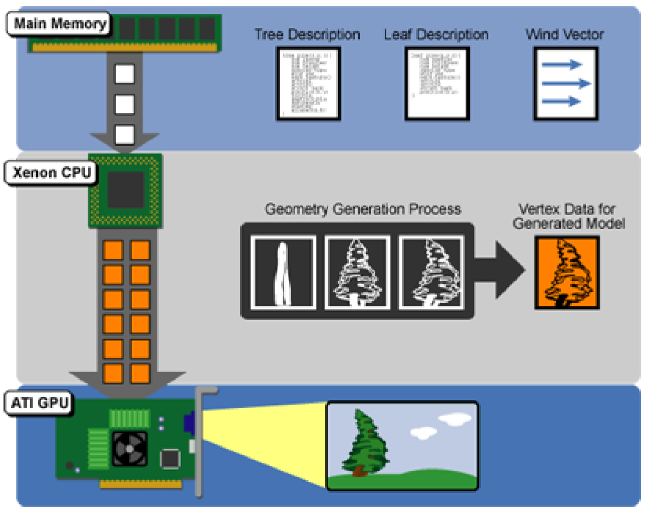


Figure 51: Xbox 360 image processing

14.2 GPU vs CPU

A GPU is tailored for high parallel operation while a CPU executes programs serially. For this reason, GPUs have many parallel execution units and higher transistor counts, while CPUs have few execution units and higher clock speeds. A GPU is for the most part deterministic in its operation (though this is quickly changing). GPUs have much deeper pipelines (several thousand stages vs 10–20 for CPUs). GPUs have significantly faster and more advanced memory interfaces as they need to shift around a lot more data than CPUs.

14.3 GPU pipeline

The GPU receives geometry information from the CPU as an input and provides a picture as an output. The five stages are:

- host interface
- vertex processing
- triangle setup
- pixel processing
- memory interface

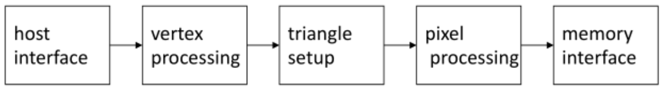


Figure 52: GPU pipeline

14.3.1 Host interface

The host interface is the communication bridge between the CPU and the GPU. It receives commands from the CPU and also pulls geometry information from system memory. It outputs a stream of vertices in object space with all their associated information (normals, texture coordinates, per vertex color, etc.)

14.3.2 Vertex processing

The vertex processing stage receives vertices from the host interface in object space and outputs them in screen space. This may be a simple linear transformation, or a complex operation involving morphing effects. Normals, texcoords, etc., are also transformed. No new vertices are created in this stage, and no vertices are discarded (input/output has 1:1 mapping).

14.3.3 Triangle setup

In this stage geometry information becomes raster information: screen space geometry is the input, pixels are the output. Prior to rasterization, triangles that are back-facing or are located outside the viewing frustum are rejected. Some GPUs also do some hidden surface removal at this stage.

14.3.4 Fragment processing

Each fragment provided by triangle setup is fed into fragment processing as a set of attributes (position, normal, textcoord, etc.), which are used to compute the final color for this pixel. The computation taking place here include texture mapping and math operations. Typically the bottleneck in modern applications.

14.3.5 Memory interface

Fragment colors provided by the previous stage are written to the framebuffer. It used to be the biggest bottleneck before fragment processing took over. Before final write occurs, some fragments are rejected by the zbuffer, stencil and alpha tests. On modern GPUs, z and color are compressed to reduce framebuffer bandwidth (but not size).

14.4 Programmability in the GPU

Vertex and fragment processing, and now triangle set-up, are programmable. The programmer can write programs that are executed for every vertex as well as for every fragment. This allows fully customizable geometry and shading effects that go well beyond the generic look and feel of old 3D applications.

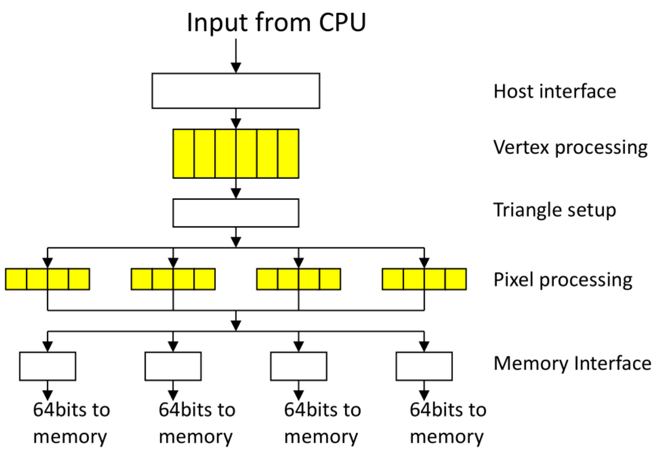


Figure 53: GPU pipeline

14.5 Command buffer CPU-GPU

The CPU and GPU inside the *heterogeneous* system work in parallel with each other. There are two "threads" going on, one for the CPU and one for the GPU, which communicate through a command buffer:

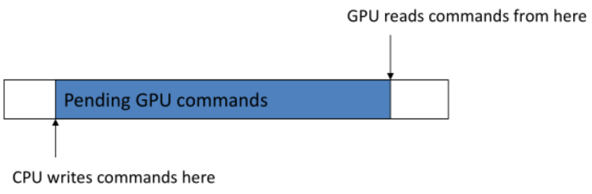


Figure 54: GPU command buffer

If this command buffer is drained empty, we are CPU limited and the GPU will spin around waiting for new input. If the command buffer fills up, the CPU will spin around waiting for the GPU to consume it, and we are effectively GPU limited.

Another important point to consider is that programs that use the GPU do not follow the traditional sequential execution model. In the CPU program below, the object is not drawn after statement A and before statement B:

statement A
API call to draw object
statement B

Instead, all the API call does, is to add the command to draw the object to the GPU command buffer.

14.5.1 Synchronization issues

Referring data In figure 55, the CPU must not overwrite the data in the "yellow" block until the GPU is done with the "black" command, which references that data.

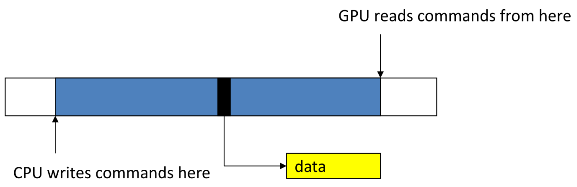


Figure 55: Referring data

Moderns APIs implement semaphore style operations to keep this from causing problems. If the CPU attempts to modify a piece of data that is being referenced by a pending GPU command it will have to spin around waiting, until the GPU is finished with that command. While this ensures correct operation it is not good for performance since there are a million other things we'd rather do with the CPU instead of spinning. The GPU will also drain a big part of the command buffer thereby reducing its ability to run in parallel with the CPU.

Inlining data One way to avoid these issues is to inline all data to the command buffer and avoid references to separate data:

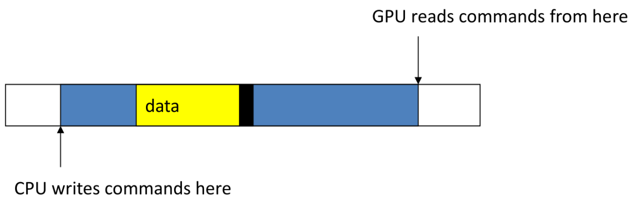


Figure 56: Inlining data

However, this is also bad for performance, since we may need to copy several megabytes of data instead of merely passing around a pointer.

Renaming data A better solution is to allocate a new data block and initialize that one instead, the old block will be deleted once the GPU is done with it. Modern APIs do this automatically, provided you initialize the entire block (if you only change a part of the block, renaming cannot occur).

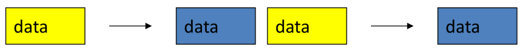


Figure 57: Renaming data

Better yet, allocate all your data at startup and don't change them for the duration of execution (not always possible, however).

14.5.2 GPU read-backs

The output of a GPU is a rendered image on the screen, what will happen if the CPU tries to read it? The GPU must be synchronized with the CPU, i.e., it must drain its entire command buffer, and the CPU must wait while this happens. We lose all parallelism, since first the CPU waits for the GPU, then the GPU waits for the CPU (because the command buffer has been drained). Both CPU and GPU performance take a nosedive. **Bottom line:** the image the GPU produces is for your eyes, not for the CPU (treat the CPU →GPU highway as a one way street).

14.5.3 GPU tips

Since the GPU is highly parallel and deeply pipelined, try to dispatch large batches with each drawing call. Sending just one triangle at a time will not occupy all of the GPU's several vertex/pixel processors, nor will it fill its deep pipelines.

Since all GPUs today use the zbuffer algorithm to do hidden surface removal, rendering objects front-to-back is faster than back-to-front (painters algorithm), or random ordering. Of course, there is no point in front-to-back sorting if you are already CPU limited.

nVidea G80 GPU Characteristics:
→128 streaming floating point processors @1.5Ghz
→1.5 Gb Shared RAM with 86Gb/s bandwidth
→500 GFLOPS on one chip (single precision)

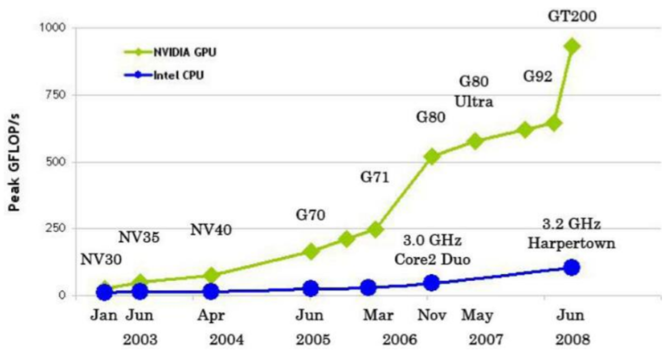


Figure 58: CPU vs GPU comparison

Why are GPU's so fast?

- Entertainment Industry has driven the economy of these chips
- Moore's Law ++
- Simplified design (stream processing)
- Single-chip designs.

Modern GPU has more ALUSs, devoting more transistors to data processing.

Very efficient for:

- Fast Parallel Floating Point Processing
- Single Instruction Multiple Data Operations

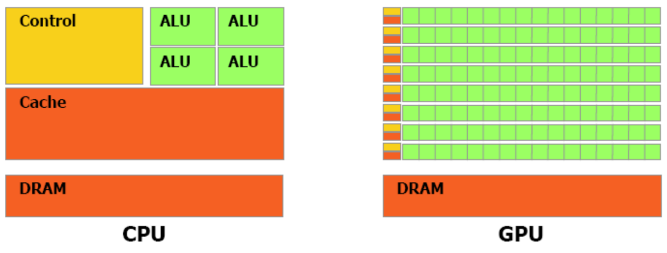


Figure 59: ALUs in CPU vs GPU

- High Computation per Memory Access

Not as efficient for:

- Double Precision
- Logical Operations on Integer Data
- Branching-Intensive Operations
- Random Access, Memory-Intensive Operations

15 MIMD

The execution can be synchronous or asynchronous, deterministic or non-deterministic.

Advantages: Multipurpose processor that can be built starting from **standard CPUs** (often off-the-shelf microprocessors) that act independently one from the other: each processor fetches its own instructions and operates on its own data. **Scalable** to a variable number of processor nodes.
Flexible:

- single-user machines focusing on high-performance for one specific application
- multi-programmed machines running many tasks simultaneously
- some combination of these functions.

Cost/performance advantages due to the use of off-the-shelf microprocessors these can be combined in the same system.

Disadvantage: Fault tolerance issues.

Exploting MIMD To exploit a MIMD with n processors, we need:

- at least n threads or processes to execute
- independent threads typically identified by the programmer or created by the compiler
- parallelism is contained in the threads $\rightarrow$thread-level parallelism

Note: parallelism is identified by the software.

15.1 Memory architecture

Existing MIMD machines fall into 2 classes, depending on the number of processors involved, which in turn dictates a memory organization and interconnection strategy.

Key design issues

- how many processors?
- how powerful are processors?
- how do parallel processors share data?
- where to place the physical memory?
- how do parallel processors cooperate and coordinate?
- what type of interconnection topology?
- how to program processors?
- how to maintain cache coherency?
- how to maintain memory consistency?
- how to evaluate system performance?

15.1.1 Centralized shared-memory architectures

- at most few dozen processor chips ($<$ 100 cores)
- large caches, single memory multiple banks
- often called symmetric multiprocessors (SMP) and the style of architecture called Uniform Memory Access (UMA), see section 15.4

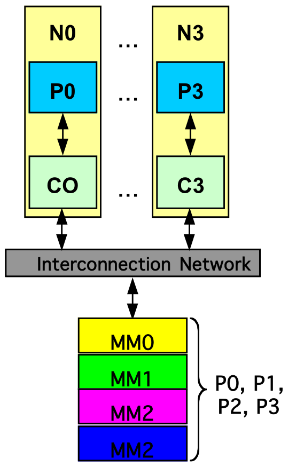


Figure 60: Shared memory architecture

15.1.2 Distributed memory architectures

- to support large processor counts
- requires high-bandwidth interconnect
- disadvantage: need of data communication among processors
- Non Uniform Memory Access (NUMA)

Tile64 is a VLIW ISA multicore processor manufactured by Tilera. It consists of a mesh network of 64 "tiles", where each tile houses a general purpose processor, cache, and a non-blocking router, which the tile uses to communicate with the other tiles on the processor. The short-pipeline, in-order, three-issue cores implement

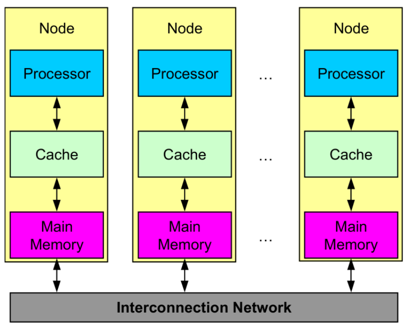


Figure 61: Distributed memory architectures

a MIPS-inspired VLIW instruction set. Each core has a register file and three functional units: two integer arithmetic logic units and a load-store unit. Each of the cores ("tile") has its own L1 and L2 caches plus an overall virtual L3 cache which is an aggregate of all the L2 caches. A core is able to run a full operating system on its own or multiple cores can be used to run a symmetrical multiprocessing operating system.

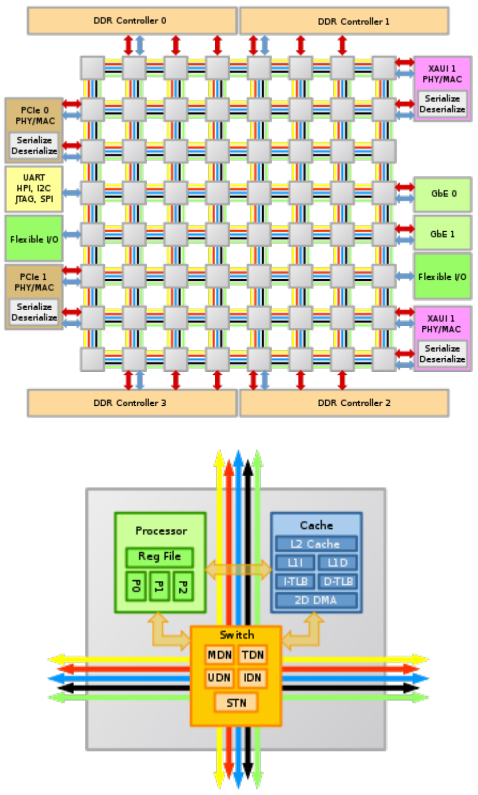


Figure 62: Tile64

15.1.3 Examples

Cell: PS3 Cell is a heterogeneous chip multiprocessor:

- One 64-bit Power core
- 8 specialized co-processors
  →based on a novel single-instruction multiple-data (SIMD) architecture called SPU (Synergistic Processor Unit)

Xenon: XBOX360 Xenon is a homogeneous chip multiprocessor:

- three symmetrical cores, each two way SMT-capable and clocked at 3.2 GHz
- SIMD: VMX128 extension for each core
- 1 MMB L2 cache (lockable by the GPU) running at half-speed (1.6 GHz) with a 256-bit bus

Microsoft envisions a procedurally rendered game as having at least two primary components:

- Host thread: a game's host thread will contain the main thread of execution for the game
- Data generation thread: where the actual procedural synthesis of object geometry takes place

These two threads could run on the same PPE, or they could run on two separate PPEs. In addition to the these two threads, the game could make use of separate threads for handling physics, artificial intelligence, player input, etc.

15.2 Memory Address Space Model

How do parallel processors share data? We already talked about how the physical memory implementation divides the *machines* in two classes, now we look how the virtual memory is from the perspective of a process.

15.2.1 Single logically shared address space

A memory reference can be made by any processor to any memory location, this model adapts well to the *shared memory architectures*: the address space is shared among processors, the same physical address on 2 processors refers to the same location in memory.

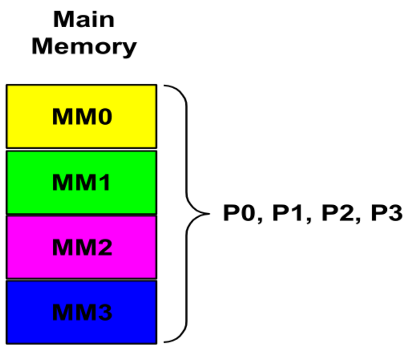


Figure 63: Single logically shared address space

Shared Address

- the processors communicate among them through shared variables in memory
- implicit management of the communication through load/store operations to access any memory locations

- shared memory does not mean that there is a single centralized memory

15.2.2 Multiple and private address spaces

The processors communicate among them through send/receive primitives, this model adapts well to *message passing architectures*: the address space is logically disjoint and cannot be addressed by different processors, the same physical address on 2 processors refers to 2 different locations in two different memories.

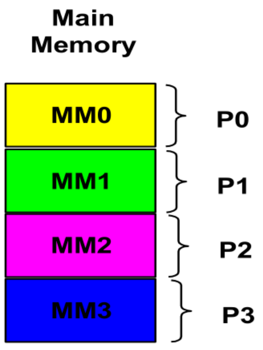


Figure 64: Multiple and private address spaces

Private address

- the processors communicate among them through sending/receiving messages
- explicit management of the communication through send/receive primitives to access private memory locations
- no cache coherency problem among processors

Note: The concepts of addressing space (single/multiple) and the physical memory organization (centralized/distributed) are orthogonal to each other: they could be combined freely. Multiprocessor systems can have single addressing space and distributed physical memory.

15.3 Programming Model

15.3.1 Shared Memory

The program is a collection of threads of control, each thread:

- can be created dynamically, mid execution, in some languages.
- has a set of *private* variables, e.g., local stack variables
- also a set of *shared* variables, e.g., static variables, shared common blocks, or global heap

Processes communication:

- implicitly by writing and reading shared variables
- coordination by synchronizing on shared variables

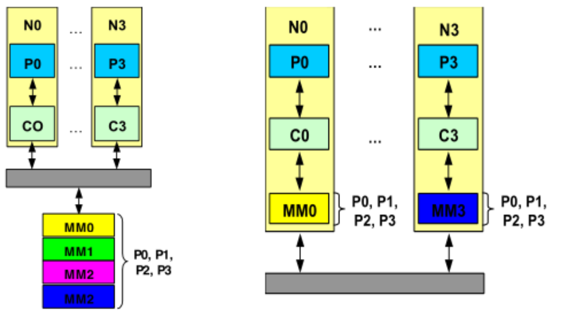


Figure 65: Single shared space vs physical memory

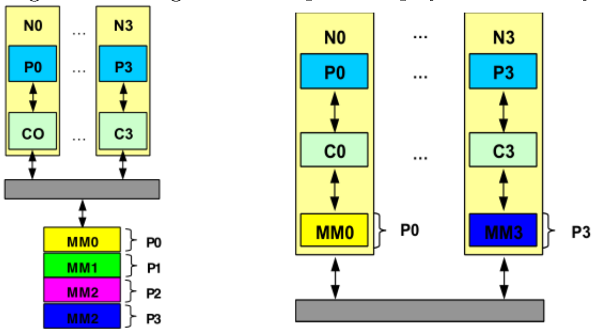


Figure 66: Multiple private addresses vs physical memory

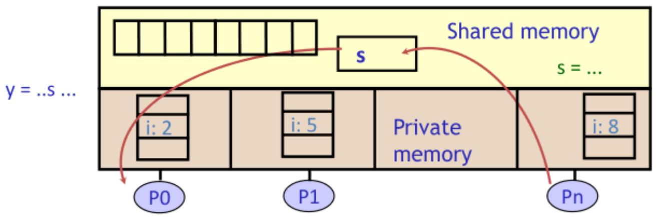


Figure 67: Shared memory

Advantages

- implicit communication (loads/stores)
- low overhead when cached

Disadvantages

- complex to build in way that scales well – e.g., distance between core and memory may become too much cutting down performance
- requires synchronization operations
- hard to control data placement within caching system

15.3.2 Message Passing

The program consist of a collection of **named** processes:

- usually fixed at program startup time
- thread of control plus local address space, no shared data
- logically shared data is partitioned over local processes

Processes communication:

- explicitly by send/receive pairs

- coordination is implicit in every communication event
- MPI (Message Passing Interface) is the most commonly used SW

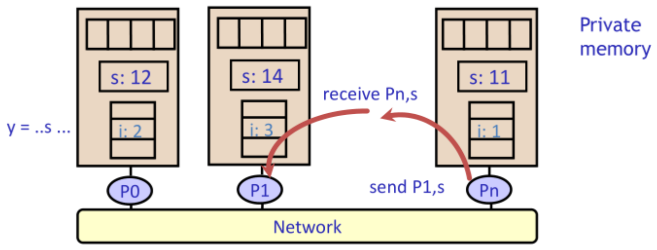


Figure 68: Message passing

Advantages

- explicit communication (sending/receiving of messages)
- easier to control data placement (no automatic caching)

Disadvantages

- message passing overhead can be quite high
- more complex to program
- introduces question of reception technique (interrupts/polling)

Massively Parallel Processors Problems

- all data must be handled by software: cannot retrieve remote data except with message request/reply
- message passing has high software overhead:
  - early machines had to invoke OS on each message
  - even user level access to network interface has dozens of cycle overhead
  - sending messages can be cheap (just like stores)
  - receiving messages is expensive, need to poll or interrupt

15.4 Bus-based symmetric shared memory

The most popular architecture till now sees each core and its own cache memory, with a simple networking solution, the bus, which connects processors with each other and to the shared memory.

Note: here we refer to *physical* shared memory architecture (UMA).

Attractive as throughput servers and for parallel programs:

- Fine-grain resource sharing
- Uniform access via loads/stores

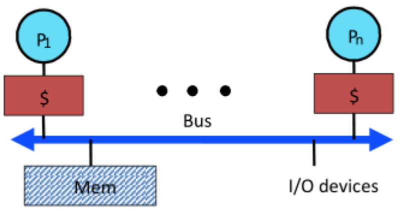


Figure 69: Bus-based symmetric multiprocecessors

- Automatic data movement and coherent replication in caches
- Cheap and powerful extension

It uses the normal uni-processor mechanisms to access data, the key to make it viable is the extension of memory hierarchy to support multiple processors.

Shared memory machines Two main categories:

- non cache coherent
- hardware cache coherent

Will work with any data, but might be slow, we can choose to optimize only critical portions of code. Load and store instructions used to communicate data: no OS involvement →low software overhead. Usually there are some special synchronization primitives. In large scale systems, the logically distributed shared memory is implemented as physically distributed memory modules.[3]

15.5 Cache coherence

Shared memory architectures cache both **private data**, used by a single processor, and **shared data**, used by multiple processors to provide communication. When shared data is cached, the shared valued may be replicated in multiple caches. In addition to the reduction in access latency and required memory bandwidth, this replication provides a reduction of shared data contention when multiple processors have to read simultaneously the same data. Private processors caches create a problem:

- copies of a variable can be present in multiple caches
- a write by one processor may not become visible to others

The use of multiple copies of the same data introduces a new problem: cache coherency.

Writing policies When a system writes data to cache, it must at some point write that data to the backing store as well. The timing of this write is controlled by what is known as the write policy. There are two basic writing approaches:

- **write-through**: write is done synchronously both to the cache and to the backing store
- **write-back** (also called write-behind): initially, writing is done only to the cache. The write to the

[3]Some notions might be wrong in this section, double check

backing store is postponed until the modified content is about to be replaced by another cache block.

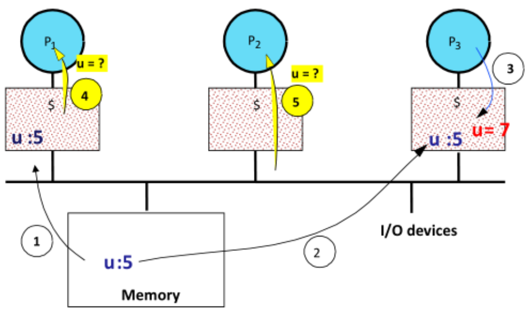


Figure 70: Cache coherence problem

Note: processors see different values for u after event 3. With write back caches, the value written back to memory depends on which cache flushes or writes back values first: processes accessing main memory may see very stale value.

What does coherency means Informal definition: *"any read must return the most recent write* →too strict and difficult to implement.
A better definition: *any write must eventually be seen by a read*, all write are seen in **serialized order**, that is two write to the same location by any two processors are seen in the same order by all processors plus the latest (in time) will be seen (in memory). If P writes x and P1 reads it, P's write will be seen by P1 if the read and write are sufficiently far apart and no other writes to x occur between the two accesses: **propagation of writes** after a certain amount of time.

When a written value will be seen? We cannot require that a read of x by P1 can instantaneously see the write of x by another processor that precedes by a small amount of time. This is a problem of **memory consistency**, coherency and consistency are complementary:

- coherence defines the behavior of reads and writes to the same memory location
- consistency defines the behavior of reads and write with respect to accesses to other memory locations

Assumptions for now:

- a write does not complete (and allow the next write to occur) until all processors have seen the effect of that write
- the processor does not change the order of any write with respect to any other memory access

Coherent caches A program running on multiple processors will normally have copies of the same data in several caches. In a coherent multiprocessor the caches provide both **migration** and **replication** of shared data items:

- Migration: a data item can be moved to a local cache and used there in a transparent fashion, reduces both the latency to access a shared data item that is allocated remotely and the *bandwidth demand* on the shared memory
- Replication for shared data that are being simultaneously read: caches make a copy of the data item in the local cache, reduces both latency of access and *contention for a shared read*

Potential solutions HW-based solutions to maintain coherency: cache-coherence protocols. Key issue: implement a cache coherent protocol in multiprocessors needs tracking the status of any sharing of a data block. Two classes of protocols:

- snooping protocols
- directory-based protocols

15.5.1 Snooping Protocols

→ All cache controllers monitor (snoop) on the bus to determine whether or not they have a copy of the block requested on the bus and respond accordingly.

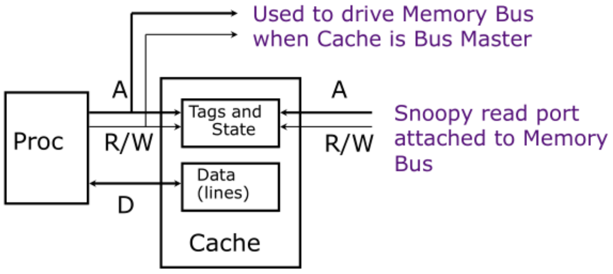
→ Every cache that has a copy of the shared block, also has a copy of the sharing status of the block, and no centralized state is kept.

→ Send all requests for shared data to all processors.

→ Require broadcast, since caching info is at processors.

→ Suitable for Centralized Shared-Memory Architectures, and in particular for small scale multiprocessors with single shared bus.

Snoopy cache Goodman 1983 Idea: have cache watch (or snoop upon) DMA[4], and then *do the right thing*. Snoopy caches tags are dual-ported:


Used to drive Memory Bus when Cache is Bus Master
A
A
Proc
R/W
R/W
Tags and State
Data (lines)
D
Cache
Snoopy read port attached to Memory Bus
Figure 71: Snoopy cache goodman

Bus is a broadcast medium and caches know what they have. Cache controller "snoops" all transactions on the shared bus:

- relevant transaction if the cache contains that block
- take action to ensure coherence (→invalidate, update, or supply value):

  depends on state of the block and the protocol

Since every bus transaction checks the cache address tags, this checking can interfere with the processor operations. When there is interference, the processor will likely

[4] Direct Memory Access

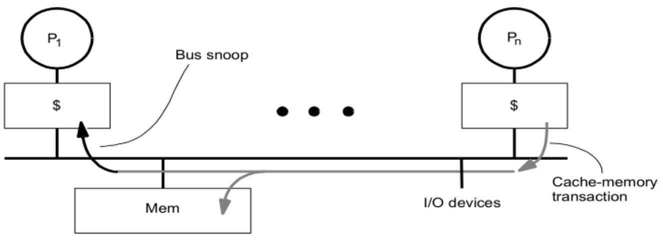


Figure 72: Cache snooping

stall because the cache is unavailable. To reduce the interference with the processor's accesses to the cache, we duplicate the address tag portion of the cache (not the whole cache) for snooping activities. In practice, an extra read port is added to the address tag portion of the cache, hence the dual-ported cache.

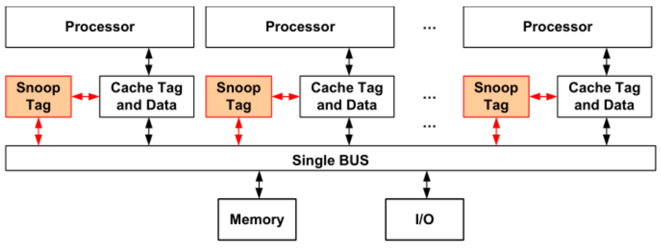


Figure 73: Dual-ported cache

Types of snooping protocols Snooping Protocols are of two types depending on what happens on a write operation:

- Write-Invalidate Protocol
- Write-Update or Write-Broadcast Protocol

15.5.2 Write-invalidate Protocol

The writing processor issues an invalidation signal over the bus to cause all copies in other caches to be invalidated before changing its local copy. The writing processor is then free to update the local data until another processor asks for it. All caches on the bus check to see if they have a copy of the data and, if so, they must invalidate the block containing the data. This scheme allows multiple readers but only a single writer. This scheme uses the bus only on the first write to invalidate the other copies. Use bus itself to serialize: write cannot complete until bus access is obtained Subsequent writes do not result in bus activity. This protocol provides similar benefits to write- back protocols in terms of reducing demands on bus bandwidth. Read Miss: snoop in caches to find the most recent copy.

15.5.3 Write-update Protocol

The writing processor broadcasts the new data over the bus; all caches check if they have a copy of the data and, if so, all copies are updated with the new value. This scheme requires the continuous broadcast of writes to shared data (while write-invalidate deletes all other copies so that there is only one local copy for subsequent writes). This protocol is like write-through because all writes go over the bus to update copies of the shared data. This protocol has the advantage of making the new values appear in caches sooner →reduced latency. Read Miss: memory always up-to-date.

Note: writing policies and snooping protocols are orthogonal concepts: a policy – e.g, write back – can be combined with any of the two snooping protocols. The latter is about cache coherence the former about when to schedule writing transitions.

15.5.4 Update vs Invalidate

Question about program behavior: is a block written by one processor later read by others before it is overwritten? For a better performance we need to look at program reference patterns and hardware complexity.

	Invalidate	**Update**
yes	readers will take a miss	avoids misses on later references
no	multiple writes without additional traffic, also clears out copies that will never be used again	multiple useless updates

Table 9: Update vs Invalidate

Most part of commercial cache-based multiprocessors uses:

- Write-Back Caches to reduce bus traffic and allow more processors on a single bus.
- Write-Invalidate Protocol to preserve bus bandwidth

The write serialization still due to bus serializing request: a write to a shared data item cannot actually complete until it obtains bus access.

Write back cache How to identify the most recent data value of a cache block in case of cache miss? It could be in (another) cache or in memory. We use the same snooping scheme both for cache misses and writes:

- each processor snoops every address placed on the bus
- if a processor finds that it has a dirty copy of the requested cache block, it provides the cache block in response to the read request
- memory access is aborted

Write back cache, write invalidate Each *block of memory* is in one of three states:

- **Clean** in all cache and up-to-date in memory (shared)
- **Dirty** in exactly one cache (exclusive)
- not in any caches

Each *cache block* can be in one of three states:

- **Clean** (or shared)(read only): the block is clean, not modified, and can be read
- **Dirty** (or modified or Exclusive): cache has only copy, its writeable, and dirty (block cannot be shared)
- **Invalid**: block contains no valid data

MSI invalidate protocol Three states:

- **M**: "Modified"
- **S**: "Shared"
- **I**: "Invalid"

Read (*PrRd*) obtains block in *shared*, even if there is only the cache copy.
Before writing a process must obtain exclusive ownership:

- *BusRdx* causes others to invalidate (demote)
- if M in another cache, it will flush
- we need to send *BusRdx* even if hit is in S (promote to M – upgrade)

What about replacement? S →I, M→I.

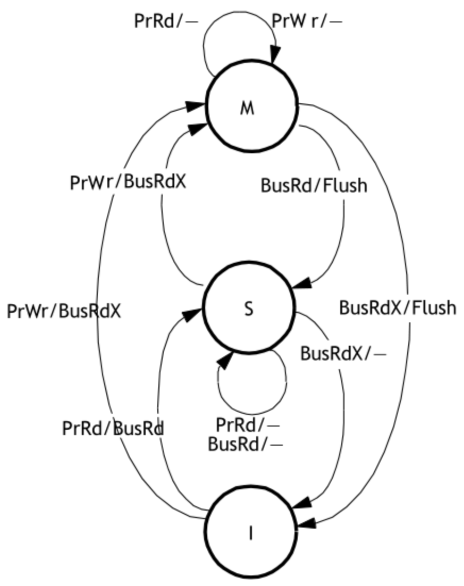


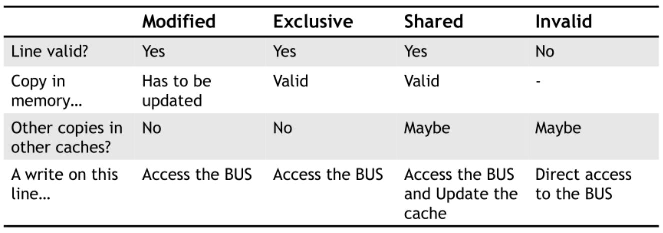
Figure 74: MSI protocol

Complications for the basic MSI protocol Operations (detect a miss, acquire bus, receive a response) are not atomic: creates the possibility of deadlock and races. One solution would be the processors that send the invalidate can hold the bus until other processors receive the invalidate.

MESI write-invalidate protocol To prevent the need to write invalidate on a write we can add an exclusive state to indicate that the clean block is in only one cache (owned state). Each cache block can be in four states:

- **Modified:** the block is dirty and cannot be shared, the cache has only copy, its writable
- **Exclusive:** the block is clean and cache has only copy
- **Shared:** the block is clean and other copies of the block are in cache
- **Invalid:** block contains no valid data

The exclusive state lets us distinguish between exclusive (writable) and owned (written).



	Modified	Exclusive	Shared	Invalid
Line valid?	Yes	Yes	Yes	No
Copy in memory...	Has to be updated	Valid	Valid	-
Other copies in other caches?	No	No	Maybe	Maybe
A write on this line...	Access the BUS	Access the BUS	Access the BUS and Update the cache	Direct access to the BUS

Figure 75: MESI states

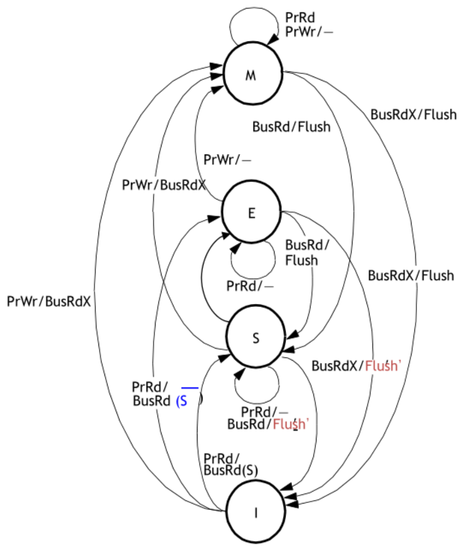


Figure 76: MESI protocol

BusRd(S) means shared line asserted on BusRd transaction. Flush if there is a cache-to-cache transfer: only one cache flushes data. Replacement: S→I can happen without telling other caches, E →I, M→I.
In both S and E, the memory has an up-to-date version of the data. A write to a E block does not require to send the invalidation signal on the bus, since no other copies of the block are in cache. A write to a S block implies the invalidation of the other copies of the block in cache.

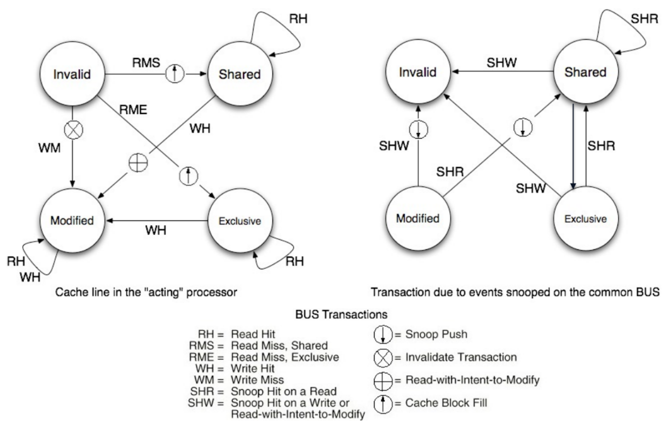


Figure 77: MESI automaton