Contents

1 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.

1.1 Measures

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

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

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

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

1.2 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.

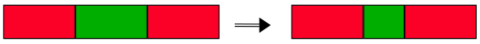


Figure 1: In green the (F) fraction enhanced.

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

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

$$\textbf{Speedup}_{\textbf{overall(max)}} = \frac{1}{(1 - F_{en})}$$

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:

$$\textbf{Speedup}_{\textbf{overall}} = \frac{1}{s + \frac{p}{N}}$$

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

1.3 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.

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

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

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

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

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

1.4 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.

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

2 Pipeline

2.1 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.

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 2: MIPS instructions

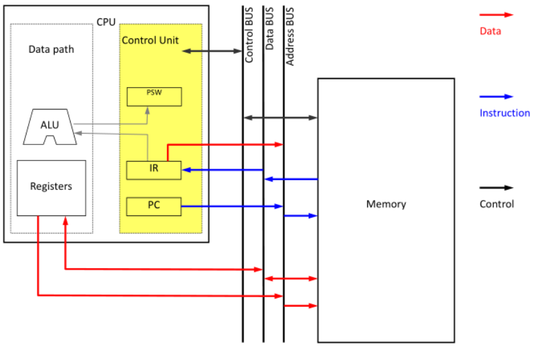


Figure 3: Computing infrastructure

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).

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.

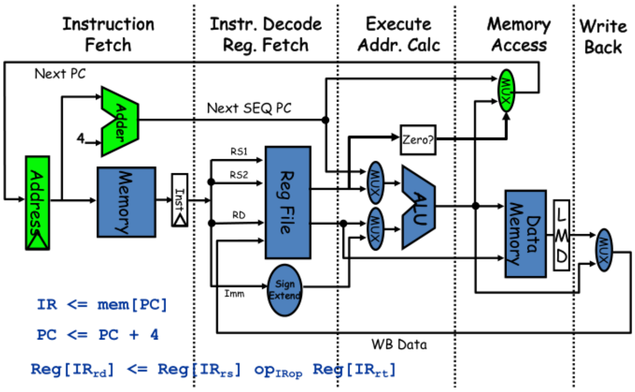
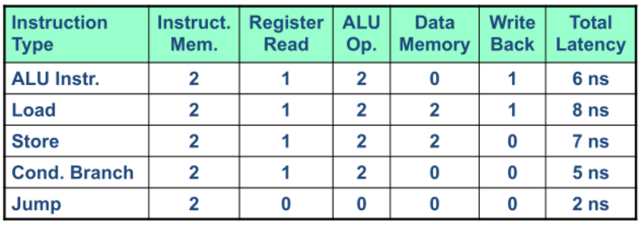


Figure 4: MIPS data path

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 5: 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.

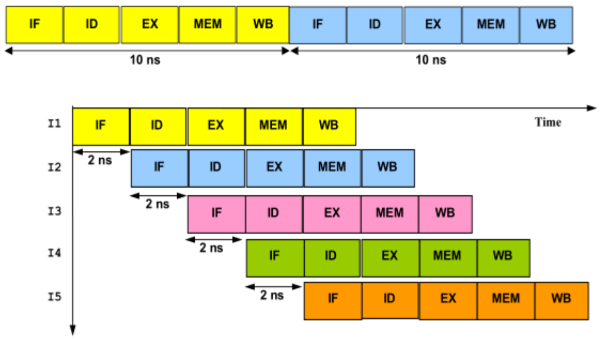


Figure 6: Sequential vs pipelined execution

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.

3 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.

3.1 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.

3.2 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

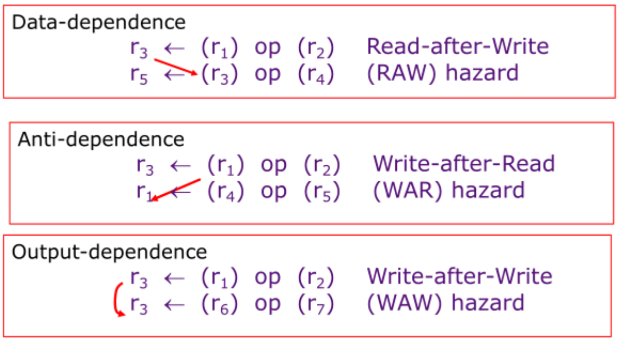

Data-dependence
$r_3 \leftarrow (r_1)$ op (r_2)
$r_5 \leftarrow (r_3)$ op (r_4)
Read-after-Write
(RAW) hazard
Anti-dependence
$r_3 \leftarrow (r_1)$ op (r_2)
$r_1 \leftarrow (r_4)$ op (r_5)
Write-after-Read
(WAR) hazard
Output-dependence
$r_3 \leftarrow (r_1)$ op (r_2)
$r_3 \leftarrow (r_6)$ op (r_7)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

Figure 7: Data hazards

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.

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).

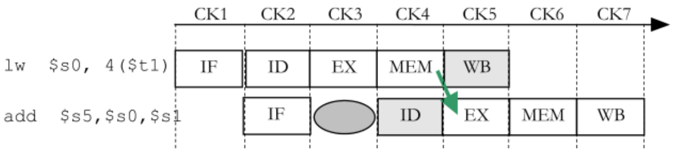


Figure 8: Load/Use hazard requires a stall to use the MEM-EX path

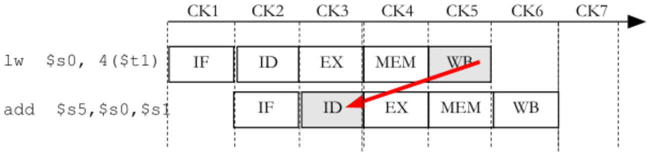


Figure 9: Load/Use without forwarding needs two stalls

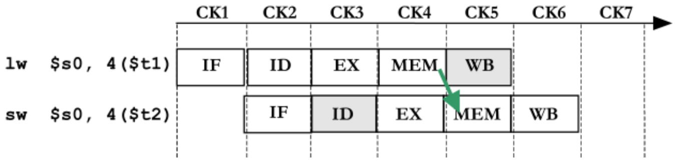


Figure 10: Load/Store hazard uses the MEM-MEM path

3.3 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.

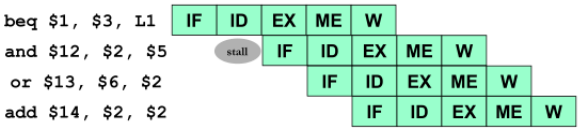


Figure 11: Early evaluation of the PC

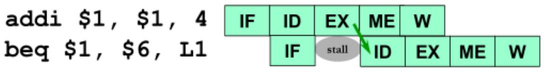


Figure 12: PC early evaluation issue

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

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

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

Figure 13: Performance impact by control hazards

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.

4.1 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.

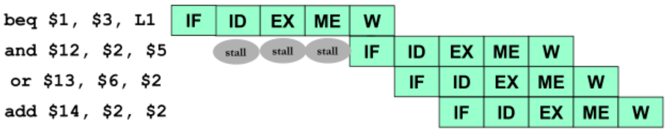


Figure 14: Branch stalls without forwarding

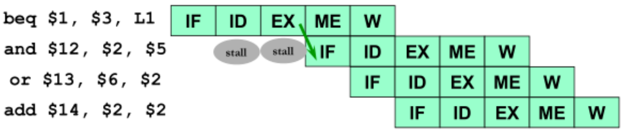


Figure 15: Branch stalls with forwarding

- 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.

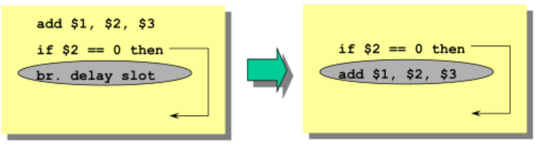


Figure 16: Branch delayed - from before

4.2 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

4.2.1 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.

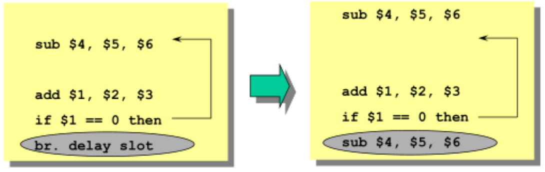


Figure 17: Branch delayed - from target

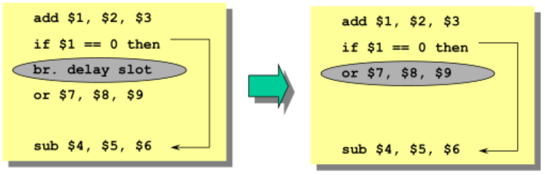


Figure 18: Branch delayed - from fall-through

Exact Address of a Branch	Predicted target address

Figure 19: BTB entry

4.2.2 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.

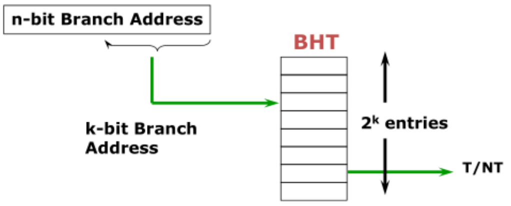


Figure 20: Branch history table

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

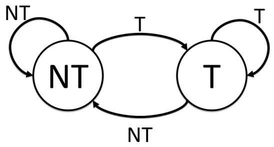


Figure 21: 1-bit BHT automata

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.

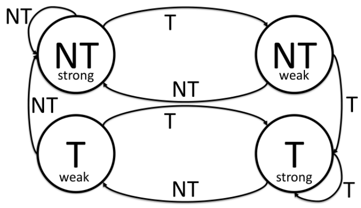


Figure 22: 2-bit BHT automata

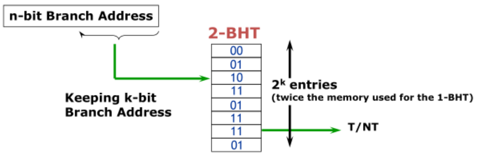

n-bit Branch Address
2-BHT
00
01
10
11
01
11
11
01
Keeping k-bit Branch Address
2^k entries
(twice the memory used for the 1-BHT)
T/NT

Figure 23: 2-bit BHT

4.2.3 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 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.

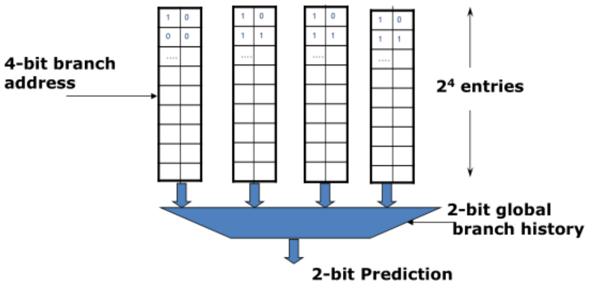


Figure 24: (2,2) correlating branch predictor

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

5 Instruction level parallelism

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

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

Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazards Stalls + 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).

ILP	**PP**
overlap individual machine operations	separate processor getting separate chuncks 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

5.1 Complex Pipeline

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

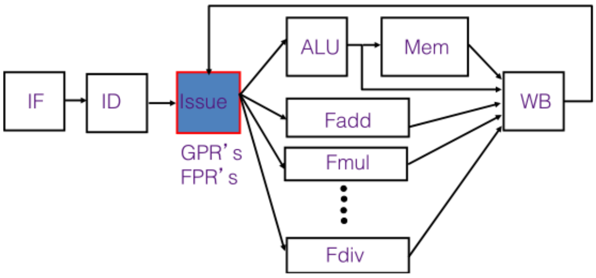


Figure 25: Complex pipeline

New hazards arise from variable latencies of different FUs:

- Structural
  - 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 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.

When it is safe to issue an Instruction? Things to consider before 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

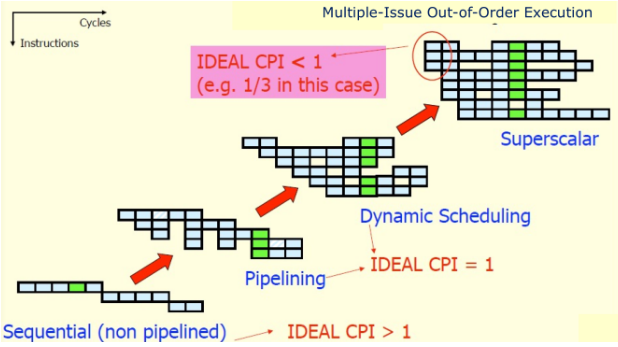


Figure 26: Instruction Level Parallelism

- 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

6 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

6.1 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.

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

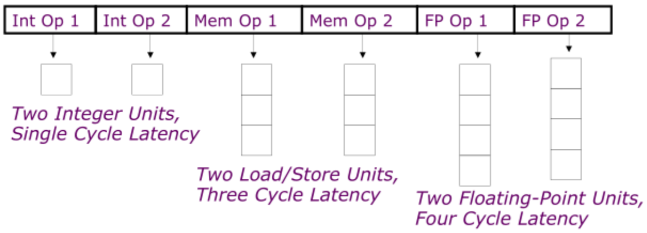

Int Op 1
Int Op 2
Mem Op 1
Mem Op 2
FP Op 1
FP Op 2
Two Integer Units, Single Cycle Latency
Two Load/Store Units, Three Cycle Latency
Two Floating-Point Units, Four Cycle Latency

Figure 27: Very Long Instruction Word

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

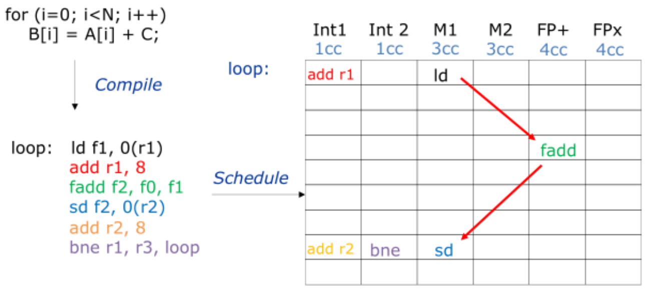
Cons:

- 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:

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.

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
```

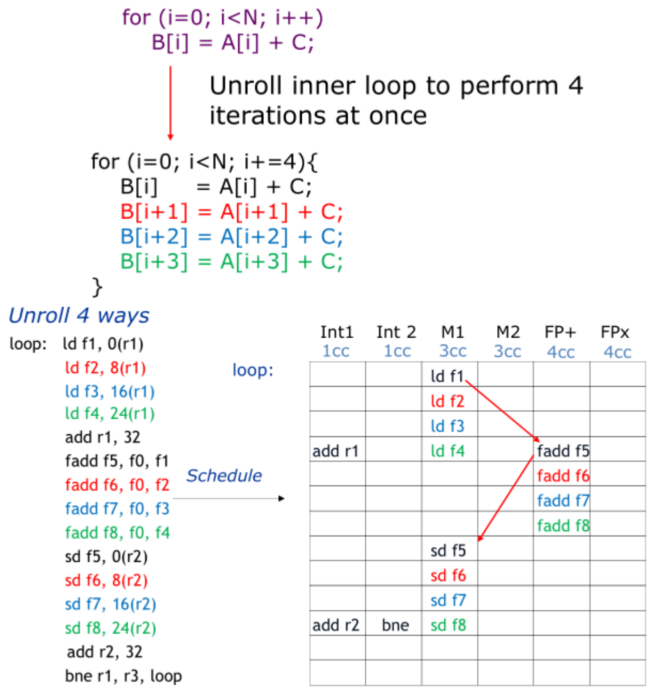
Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
add r1		ld			
				fadd	
add r2	bne	sd			

Figure 28: Loop execution on VLIM

6.2 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*.

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



```
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 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		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 29: Loop unrolling

Note that non FP operations can have multiple schedule locations, e.g., 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.

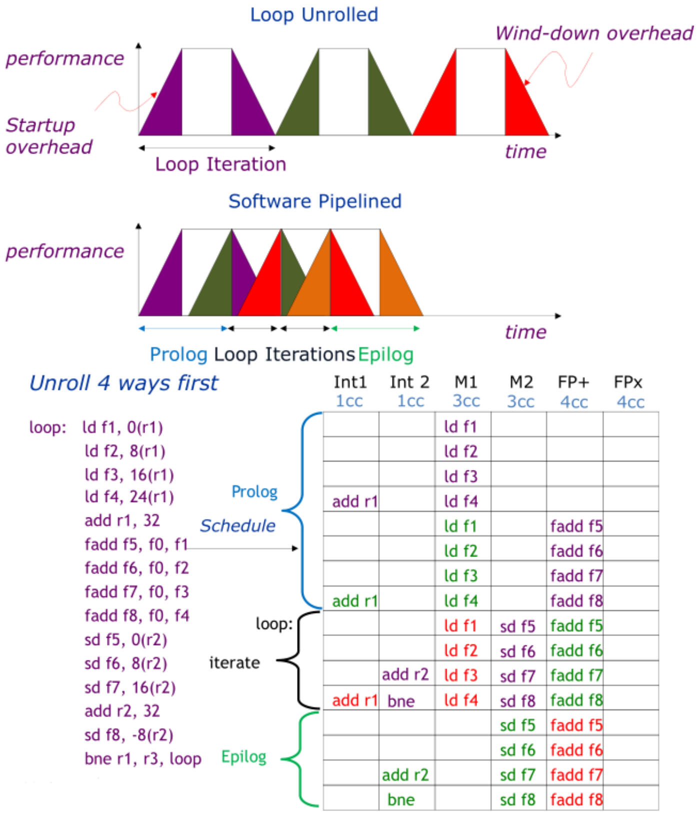
The **limits** of loop unrolling:

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 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		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 30: Loop unrolled vs Software pipelined

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

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.

6.3 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

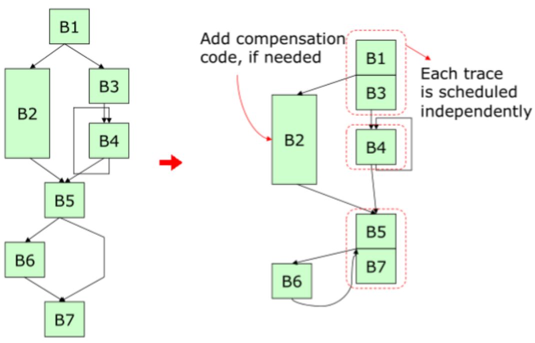

B1
B2
B3
B4
B5
B6
B7
Add compensation code, if needed
B1
B3
Each trace is scheduled independently
B2
B4
B5
B7
B6

Figure 31: 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.

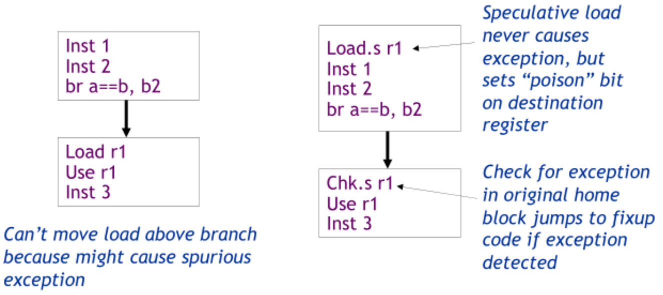

Inst 1
Inst 2
br a==b, b2
Load r1
Use r1
Inst 3
Can't move load above branch because might cause spurious exception
Load.s r1
Inst 1
Inst 2
br a==b, b2
Chk.s r1
Use r1
Inst 3
Speculative load never causes exception, but sets "poison" bit on destination register
Check for exception in original home block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

Figure 32: Speculative instruction

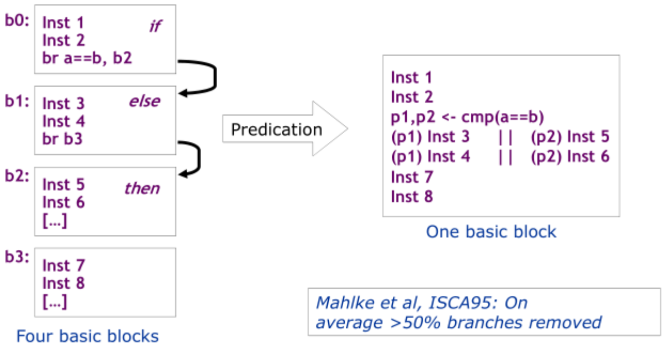


Figure 33: Predicated execution

6.4 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.

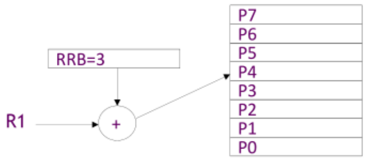


Figure 34: Rotating Register Base

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

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

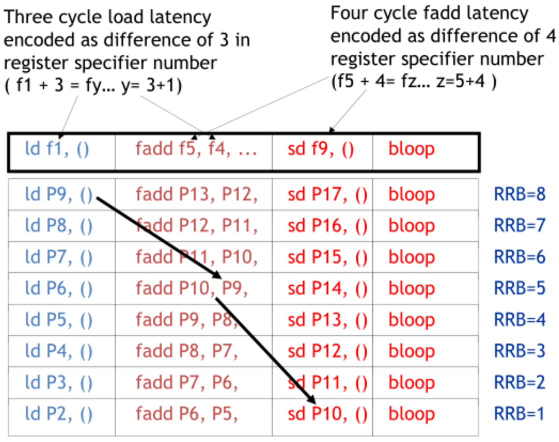


Figure 35: Rotating registers

7 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.

7.1 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

7.1.1 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.

7.1.2 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.

Scoreboard: RAW ID-read stalls WAR WB stalls WAW not issued (stall the issue), register renaming not used

7.2 Tomasulo

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

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

Tag	identifying the RS
OP	Operation to perform on the component.
Vj,Vk	Value of the source operands.

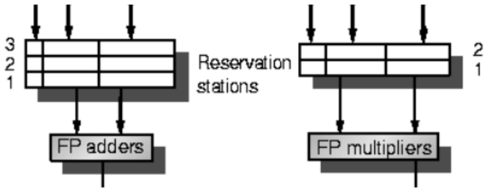


Figure 36: Reservation stations (RS)

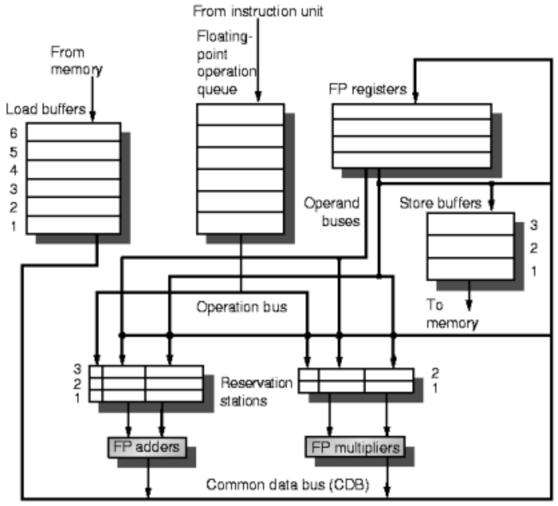


Figure 37: Tomasulo architecture

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.

Register Renaming The operands of the RSs are replaced by values or pointers. This allows to:

- avoid WAR and WAW hazards
- reservation stations are more than registers, make possible better optimization than a compiler.

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.

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

Load buffers have an address field (A), and a busy field. Store Buffers have also an address field (A).
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.

7.2.1 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).

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

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 common data bus for results. By delaying execution until operands are available, RAW hazards are avoided. Notice that several instructions could become ready in the same clock cycle for the same FU.

Load and stores: two-step process:

- First step: compute effective address, place it in load or store buffer.
- Loads in Load Buffer execute as soon as memory unit is available
- Stores in store buffer wait for the value to be stored 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.

Stores write data to memory during this stage.

Mark reservation stations available.

—
in case of concurrent writes choose the one that belongs to the critical path. limitations see slide 78 —

7.2.2 Tomasulo in detail

Loads and stores go through a functional unit for effective address computation before proceeding to effective load and store buffers.

Loads take a second execution step to access memory, then go to Write Result to send the value from memory to RF and/or RS:

1 cc (offset + base address) + **1 cc** access memory = **2 cc** execution

Then at next clock cycle it commits to the CDB.

Stores complete their execution in their Write Result stage (writes data to memory).

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.

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.

—

7.2.3 Tomasulo Loop

Why can Tomasulo overlap iterations of loops? • Register renaming – Multiple iterations use different physical destinations for registers (dynamic loop unrolling). – Replace static register names from code with dynamic register "pointers" – Effectively increases size of register file – Permit instruction issue to advance past integer control flow operations. • Crucial: integer unit must "get ahead" of floating point unit so that we can issue multiple iterations Þ Branch Prediction Other idea: Tomasulo building "DataFlow" graph.

Tomasulo Drawbacks • Complexity – Large amount of hardware – delays of 360/91, MIPS 10000, IBM 620? • Many associative stores (CDB) at high speed • Performance limited by Common Data Bus – Multiple CDBs => more FU logic for parallel assoc stores

Summary • HW exploiting ILP – Works when can't know dependence at compile time. – Code for one machine runs well on another • Reservations stations: renaming to larger set of registers + buffering source operands – Prevents registers as bottleneck – Avoids WAR, WAW hazards of Scoreboard – Allows loop unrolling in HW • Not limited to basic blocks (integer units gets ahead, beyond branches) • Lasting Contributions – Dynamic scheduling – Register renaming – Load/store disambiguation

—-

Scoreboard vs Tomasulo: - structural hazards in scoreboard - lack of forwarding in scoreboard

Tomasulo (IBM) versus Scoreboard (CDC) • Issue window size=14 • No issue on structural hazards • WAR, WAW avoided with renaming • Broadcast results from FU • Control distributed on RS • Allows loop unrolling in HW • Issue window size=5 • No issue on structural hazards • Stall the completion for WAW and WAR hazards • Results written back on registers. • Control centralized through the Scoreboard.

Control & buffers distributed with Function Units (FU) vs. centralized in scoreboard; – FU buffers called "reservation stations"; have pending operands • Registers in instructions replaced by values or pointers to reservation stations(RS); called register renaming ; – avoids WAR, WAW hazards – More reservation stations than registers, so can do optimizations compilers can't • Results to FU from RS, not through registers, over Common Data Bus that broadcasts results to all FUs • Load and Stores treated as FUs with RSs as well • Integer instructions can go past branches, allowing FP ops beyond basic block in FP queue

8 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.

8.1 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

8.2 Exception classes

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

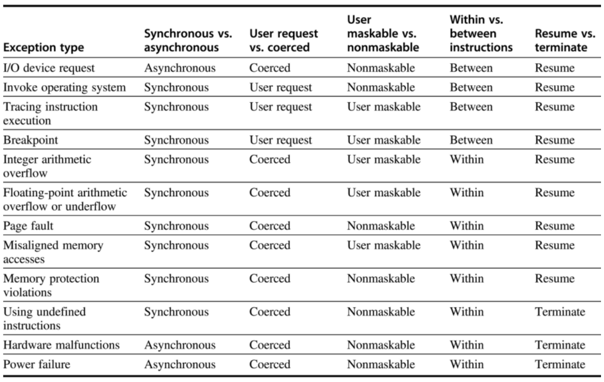
User request vs coerced user requested are predictable: treated as exceptions because they use the same mechanism that are used to save and restore the state handled after the instruction has completed. Coerced are caused by some HW event not under control of the program.

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 38: Exception classes

8.3 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:

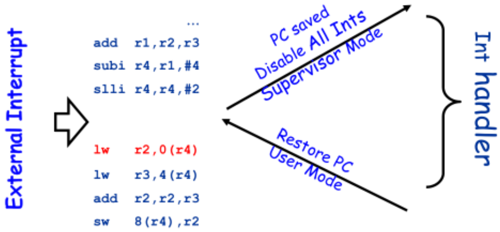
- enables interrupts
- restores the processor to the user mode
- restores hardware status and control state

8.4 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.

8.5 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 39: Precise interrupt

9 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.

9.1 Reorder buffer

A buffer that holds instruction results before they are committed. When an instruction complete 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, supplying operands to other instruction between execution and commit. The instruction at the **head** of ROB can safely commit, all its predecessor have already committed.

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 program counter, instruction type, destination register specifier and value if any, and exception status

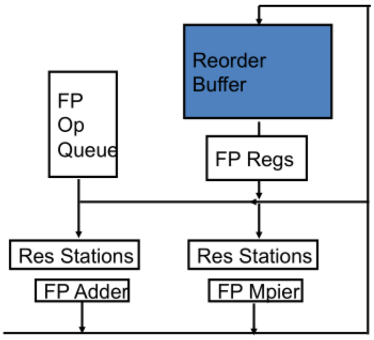


Figure 40: ROB

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**.

9.2 Speculative Tomasulo

9.2.1 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

9.2.2 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)

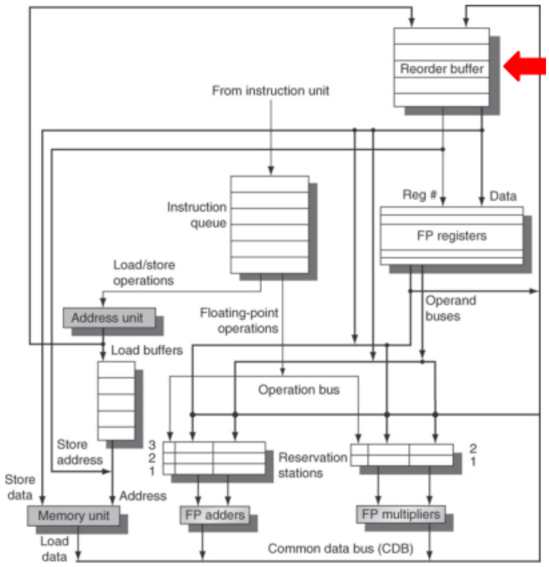


Figure 41: Speculative tomasulo

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

9.2.3 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 →speculative active are easily undone
- processor with ROB can dynamically speculate while maintaining a precise interrupt model

10 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

10.1 Explicit renaming - Scoreboard

10.1.1 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.

10.2 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.

11 ILP limits

11.1 Superscalar

Superscalar execution allows multiples-issue and out-of-order execution.

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.

11.2 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

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

11.3 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 $n^2 - n$.