ARCOS

Introduction to instruction level parallelism

Computer Architecture

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Francisco Javier García Blas

ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid

1 Introduction to pipelining

2 Hazards

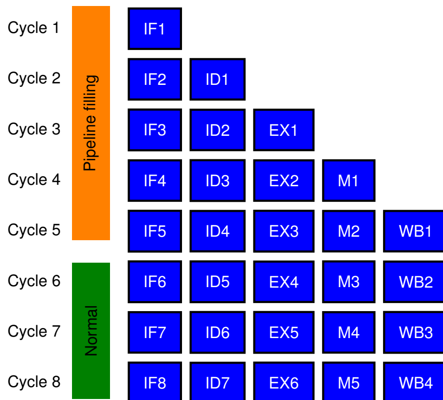
3 Multi-cycle operations

4 Conclusion

Pipeline

- Implementation technique to execute multiple instructions overlapped in time.
  - A **costly** operation is divided into simple sub-operations.
  - Sub-operations are executed into stages.
- **Effects**:
  - **Increases *throughput***.
  - **Latency** is not decreased.

Pipeline



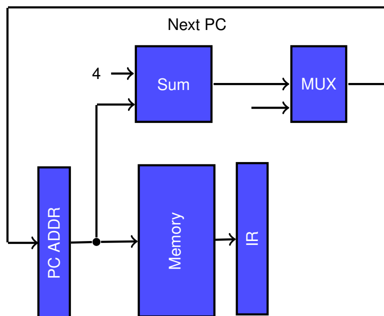
- **Latency**:
  - One instruction requires 5 stages.
  - 5 cycles.
- **Throughput**:
  - One instruction finalized per cycle (once pipeline is full).
  - 1 instruction per cycle.

Pipeline stages

- Simplified model (MIPS):
  - 5 stages.
  - More realistic models require more stages

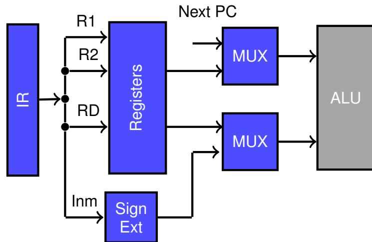
- **Stages**:
  - **Instruction Fetch**.
  - **Instruction Decode**.
  - **Execution**.
  - **Memory**.
  - **Write-back**.

Instruction Fetch



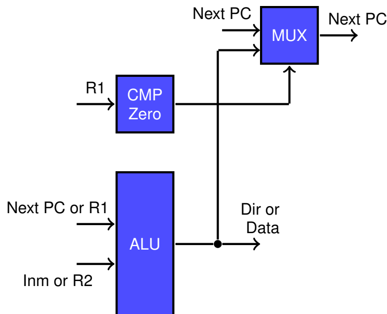
- Send PC to memory.
- Read instruction.
- Update PC.

Instruction Decode



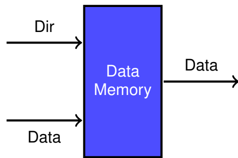
- Decode instruction.
- Read registers.
- Sign extend offsets.
- Compute possible branch address.

Execution



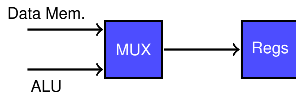
- ALU operation on registers.
- Alternatively, compute effective address.

Memory



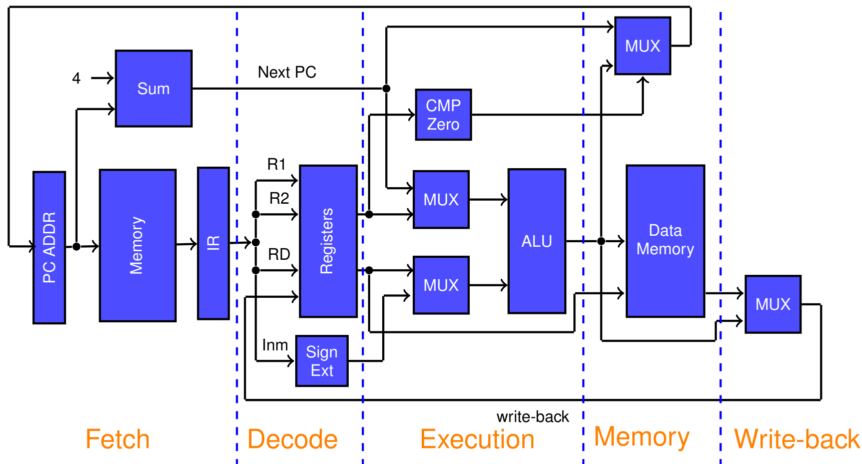
- Read from or write into memory.

Write back



- Write result into register file.

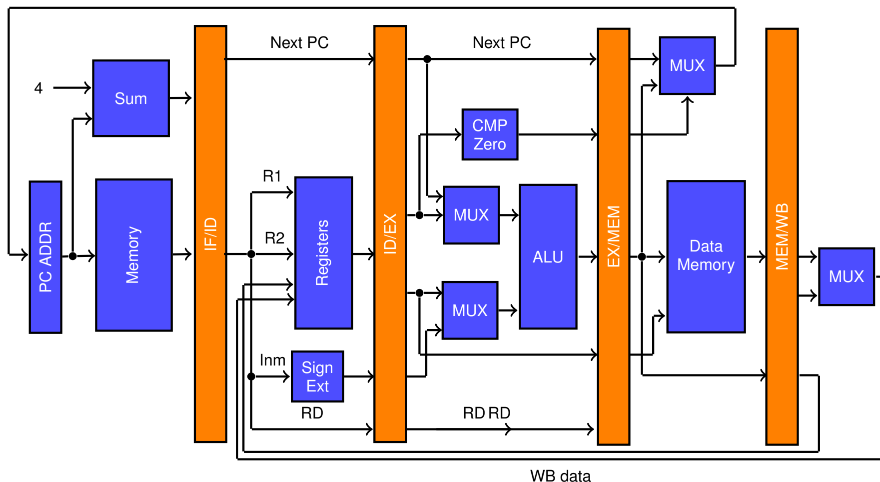
General architecture



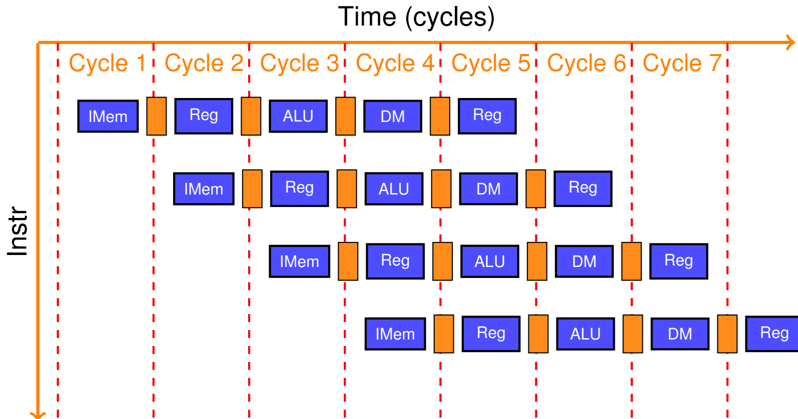
Pipeline effects

- An n **depth** pipeline multiplies by n the needed **bandwidth** compared to a non-pipelined version with the same clock rate.
  - Caching, caching, ...
- Separation among **data** and **instructions caches** suppresses some memory conflicts
- Instructions in the pipeline should not try to use the same resource at the same time.
  - Pipelining registers between successive stages.

Stages communication



Pipeline over time



- Register read in second half of cycle.
- Register write in first half of cycle.

Example

- Non-pipelined processor.
  - Clock cycle: 1 ns.
  - 40% ALU operations $\rightarrow$ 4 cycles.
  - 20% branch operations $\rightarrow$ 4 cycles.
  - 40% memory operations $\rightarrow$ 5 cycles.
  - Pipeline overhead $\rightarrow$ 0.2 ns.
- Which is the pipeline speedup?

$$t_{orig} = cycle_{clock} \times CPI = 1ns \times (0.6 \times 4 + 0.4 \times 5) = 4.4ns$$

$$t_{new} = 1ns + 0.2ns = 1.2ns$$

$$S = \frac{4.4ns}{1.2ns} = 3.7$$

1 Introduction to pipelining

2 Hazards

3 Multi-cycle operations

4 Conclusion

Hazard

- A **hazard** is a situation preventing next instruction to start at the expected clock cycle.
  - Hazards reduce performance in pipelined architectures.

- **Types of hazards**:
  - **Structural hazard**.
  - **Data hazard**.
  - **Control hazard**.

- Simple approach to hazards:
  - Stall the instruction flow.
  - Already started instructions will continue.

2 Hazards
- Structural hazards
- Data hazards
- Control hazards

Structural hazard

- Happens when hardware cannot support all possible instruction sequences.
  - In the same cycle two pipeline stages need to use the **same resource**.

- **Reasons**:
  - Functional units that are not fully pipelined.
  - Functional units which are not duplicated.

- These hazards can be avoided at the cost of a more expensive hardware.

Pipeline speedup

- **Speedup**:
  - $t_{nonpipelined}$: Average instruction time in non-pipelined architecture.
  - $t_{pipelined}$: Average instruction time in pipelined architecture.

$$S = \frac{t_{nonpipelined}}{t_{pipelined}} = \frac{CPI_{nonpipelined} \times cycle_{nonpipelined}}{CPI_{pipelined} \times cycle_{pipelined}}$$

- **Ideal case**: pipelined **CPI** is 1.
  - Need to add stall cycles per instruction.

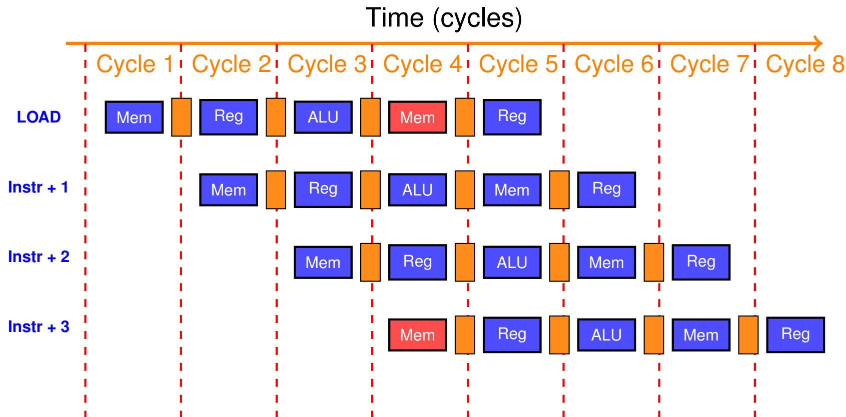
Pipeline speedup

- In case of non-pipelined processor:
  - $CPI = 1$, with $cycle_{nonpipelined} > cycle_{pipelined}$.
  - $cycle_{nonpipelined} = N \times cycle_{pipelined}$.
  - $N \rightarrow$ **Pipeline depth**.

Speedup

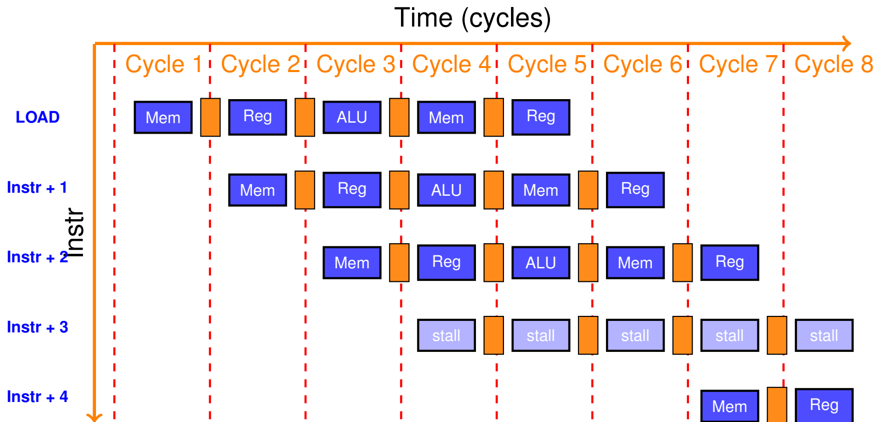
$$S = \frac{N}{1 + \text{stalls per instruction}}$$

Structural hazards: example



- Assuming single port memory.

Structural hazards: example



- Assuming single port memory.

Example

- Two alternative designs:
  - **A**: No structural hazards.
    - Clock cycle $\rightarrow 1ns$
  - **B**: With structural hazards.
    - Clock cycle $\rightarrow 0.9ns$
  - Data access instructions with hazards $\rightarrow$ 30%.
- Which one is the fastest alternative?

$$t_{inst}(A) = CPI \times cycle = 1 \times 1ns = 1ns$$

$$t_{inst}(B) = CPI \times cycle = (0.7 \times 1 + 0.3 \times (1 + 1)) \times 0.9ns = 1.17ns$$

2 Hazards

- Structural hazards
- Data hazards
- Control hazards

Data hazard

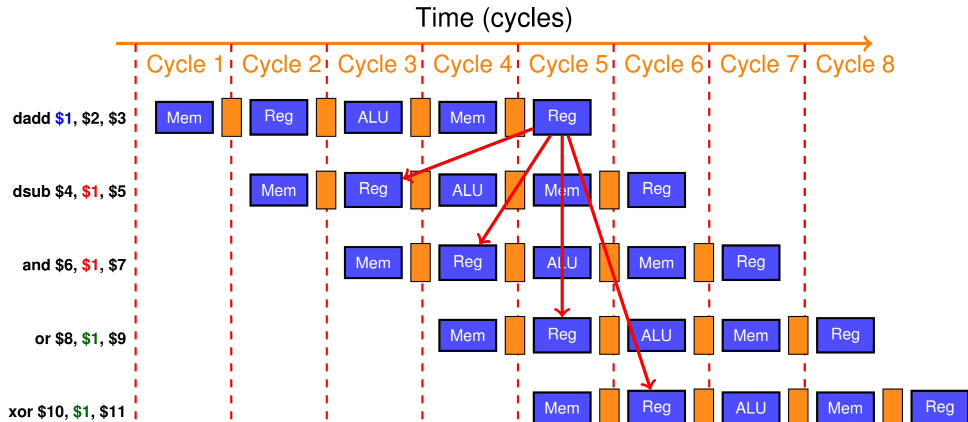
- A **data hazard** happens when the pipeline changes the read/write access to operands ordering.

Example

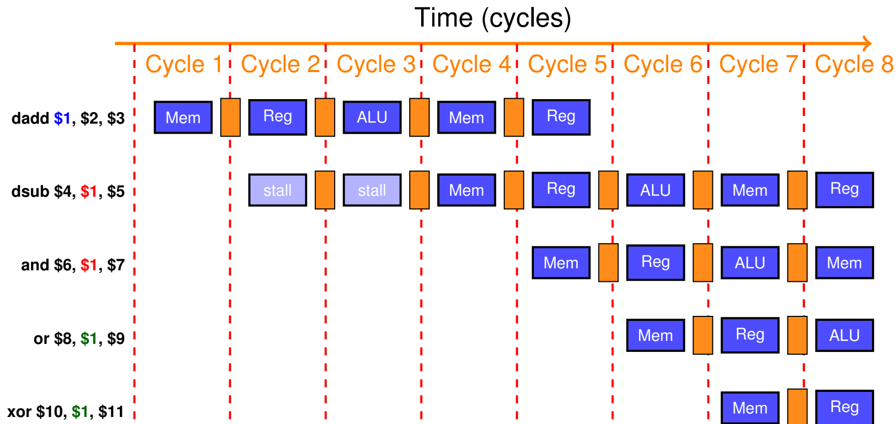
```
I1: dadd $1, $2, $3
I2: dsub $4, $1, $5
I3: and  $6, $1, $7
I4: or   $8, $1, $9
I5: xor  $10, $1, $11
```

- **I2** reads **R1** before than **I1** modifies it.
- **I3** reads **R1** before than **I1** modifies it.
- **I4** gets the right value.
  - Register file is read in second half of cycle.
- **I5** gets the right value.

Data hazards



Stalls in data hazards



Data hazards: RAW

- **Read After Write**.
  - Instruction **i+1** tries to read a datum before instruction **i** writes it.

Example

```
i :     add $1, $2, $3
i+1:    sub $4, $1, $3
```

- If there is a data dependency, instructions:
  - Can neither be executed in parallel nor overlap.
  - Instruction **sub** needs value from **$1** produced by instruction **add**.

- **Solutions**:
  - **Hardware detection**.
  - **Compiler control**.

Data hazards: WAR

- **Write After Read**.
  - Instruction **i+1** modifies operand before instruction **i** reads it.

Example

```
i :   sub $4, $1, $3
i+1: add $1, $2, $3
i+2: mul $6, $1, $7
```

- Also known as **anti-dependence** in compiler technology.
  - Name reuse.
  - Instruction **add** modifies **$1** before **sub** reads it.

- Cannot happen in a MIPS with **5-stages pipeline**.
  - All instructions with 5 stages.
  - Reads always happen in stage 2.
  - Writes always happen in stage 5.

Data hazards: WAW

- **Write After Write**.
  - Instruction **i+1** modifies operand before instruction **i** modifies it.

Example

```
i :   sub $1, $4, $3
i+1: add $1, $2, $3
i+2: mul $6, $1, $7
```

- Also known as **output dependency** in compiler technology
  - Name reuse.
  - Instruction **add** modifies **$1** before **sub** modifies it.

- Cannot happen in a MIPS with **5-stages pipeline**.
  - All instructions with 5 stages.
  - Writes always happen in stage 5.

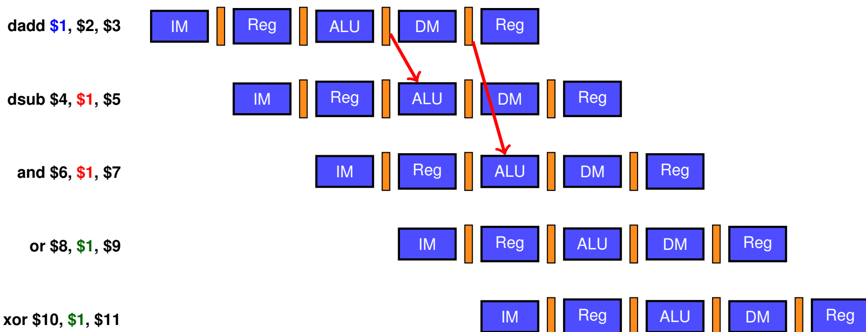
Solutions to data hazards

- **RAW dependencies**:
  - **Forwarding**.
- **WAR and WAW dependencies**:
  - **Register renaming**.
    - Done statically by compiler.
    - Done dynamically by hardware.

Forwarding

- Technique to avoid some data stalls.
- **Basic idea**:
  - No need to wait until result is written into register file.
  - Result is already in pipeline registers.
  - Use that value instead of the one from the register file.
- **Implementation**:
  - Results from EX and MEM stages written into ALU input registers.
  - *Forwarding* logic selects between real input and *forwarding* register.

Forwarding

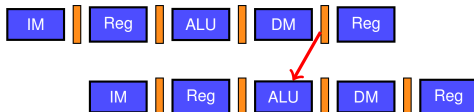


forwarding limitations

- Not every hazard can be avoided with forwarding.
  - You cannot travel backwards in time!

Example

```
I1: lw    $1, (0)$2
I2: dsub  $4, $1, $5
I3: and   $6, $1, $7
I4: or    $8, $1, $9
I5: xor   $10, $1, $11
```



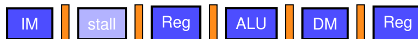
- If hazard cannot be avoided, a stall must be introduced.

Stalls in memory accesses

lw $1, 0($2)

dsub $4, $1, $5

and $6, $1, $7



or $8, $1, $9

xor $10, $1, $11

2 Hazards
- Structural hazards
- Data hazards
- Control hazards

Control hazard

- A **control hazard** happens in a **PC** modification instruction.

- **Terminology**:
  - **Taken branch**: If PC is modified.
  - **Not-taken branch**: if PC is not modified.

- Problem:
  - Pipelining assumes that branch will **not** be taken.
  - What if, after ID, we find out that branch needs to be taken?

Alternatives in control hazards

- **Compile time**: Fixed asumption for the full program execution.
  - Software may try to minimize impact if hardware behavior is known.
  - Compiler can do this job.

- **Run-time**: Variable behavior during program execution.
  - Hardware tries to predict what software will do.

Control hazards: static solutions

1. Pipeline freezing.
2. Fixed prediction.
   - Always not taken.
   - Always taken.
3. Delayed branching.

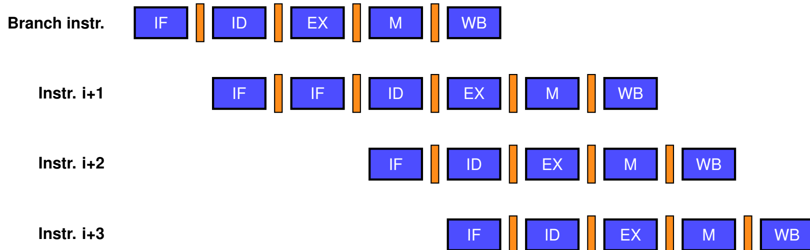
- In many cases, compilers need to know what will be done to reduce impact.

Pipeline freezing

- **Idea**: If current instruction is a branch $\rightarrow$ stop or flush subsequent instructions from pipeline until target is known.
  - Run-time penalty is known.
  - Software (compiler) cannot do anything.

- Branch target is known at **ID** stage.
  - Repeat next instruction **FETCH**.

Repeating FETCH

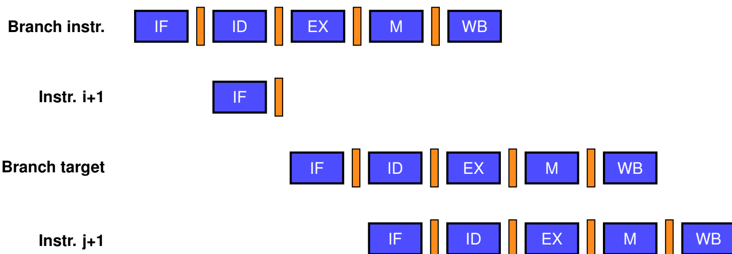


- Repeating **IF** is equivalent to a stall.
- A branch stall may lead to a **performance loss** from 10% to 30%.

Fixed prediction: not taken

- **Idea**: Assume branch will be **not taken**.
  - Avoid updating processor state until branch not taken is confirmed.
  - If branch is finally taken, subsequent instructions are retired from pipeline and instruction from target address is fetched.
    - Transform instructions into **NOP**s.
- **Compiler task**:
  - Organize code setting most frequent option as not-taken and inverting condition if needed.

Fixed prediction: not taken



- When hardware knows the branch will be taken the new instruction (*j+1*) is fetched.

Fixed prediction: taken

- **Idea**: Assume branch will be **taken**.
  - As soon as branch instruction is decoded and target address is computed, target instruction starts to be fetched.
  - In a 5-stages pipeline does not provide improvements.
    - Target address is not known before branch outcome is known.
    - Useful in processors with complex and slow conditions.

- **Compiler task**:
  - Organize code setting the most frequent option as taken and inverting condition if needed.

Delayed branching

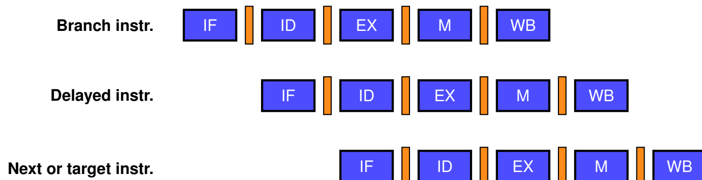
- **Idea**: Branch happens after executing **n** subsequent instructions to branch instruction.
  - **In a 5-stages pipeline** $\rightarrow$ 1 delay slot.

Example

```
I0:    bnez  $1, etiq
I1:    addi  $2, $2, 1
I2:    mul   $3, $2, $4
...
IN:    sub   $1, $1, 1
IN+1:  mul   $3, $3, $4
```

- Instructions **I1**, **I2**, . . . , **IN** are executed independently of the branch condition outcome.
- Instruction **IN+1** is only executed if branch is not taken.

Delayed branching



- Case with **delayed branch** and one **delay slot**.
- One instruction is always executed before taking the branch.
- Programmer are responsible for putting useful code in the slot.

Compilers and delay slot

Original

```
       add  $1, $2, $3
       beqz $2, etiq
       nop
       xor  $5, $6, $7
etiq:  and  $8, $9, $10
```

Improved

```
       beqz $2, etiq
       add  $1, $2, $3
       xor  $5, $6, $7
etiq:  and  $8, $9, $10
```

Original

```
etiq:  add  $1, $2, $3
       xor  $5, $6, $7
       beqz $5, etiq
       nop
       and  $8, $9, $10
```

Improved

```
       add  $1, $2, $3
etiq:  xor  $5, $6, $7
       beqz $5, etiq
       add  $1, $2, $3
       and  $8, $9, $10
```

Original

```
       add  $1, $2, $3
       beqz $1, etiq
       nop
       xor  $5, $6, $7
etiq:  and  $8, $9, $10
```

Improved

```
       add  $1, $2, $3
       beqz $1, etiq
       xor  $5, $6, $7
etiq:  and  $8, $9, $10
```

Delayed branching

- Compiler effectiveness for the case of 1 slot.
  - **Fills** around 60% of slots.
  - Around 80% of executed instructions in slots **useful** for computations.
  - Around 50% of slots usefully **filled**.

- Using deeper pipelines and multiple instruction issue, more slots are needed.
  - Abandoned in favor of more dynamic approaches.

Performance and fixed prediction

- Number of stalls depends on:
  - Branch frequency.
  - Branch penalty.

- Penalty cycles per branch:

$$cycles_{branch} = frequency_{branch} \times penalty_{branch}$$

- Speedup:

$$S = \frac{depth_{pipeline}}{1 + frequency_{branch} \times penalty_{branch}}$$

Practical case

- **MIPS R4000** (deeper pipeline).
  - 3 stages before knowing branch target.
  - 1 additional stage to evaluate condition.
  - Assuming no data stalls in comparisons.
  - **Branch frequency**:
    - **Unconditional branching**: 4%.
    - **Conditional branching, not-taken**: 6%
    - **Conditional branching, taken**: 10%

branch scheme	Penalty		
	unconditional	not-taken	taken
Flush pipeline	2	3	3
Predict taken	2	3	2
Predict not-taken	2	0	3

Solution

branch scheme	Branch			Total
	unconditional	not-taken	taken	
Frequency	4%	6%	10%	20%
Flush pipeline	$0.04 \times 2 = 0.08$	$0.06 \times 3 = 0.18$	$0.10 \times 3 = 0.30$	0.56
Predict taken	$0.04 \times 2 = 0.08$	$0.06 \times 3 = 0.18$	$0.10 \times 2 = 0.20$	0.46
Predict not-taken	$0.04 \times 2 = 0.08$	$0.06 \times 0 = 0.00$	$0.10 \times 3 = 0.30$	0.38

Contribution over ideal CPI

Speedup of predicting **taken** over flushing pipeline

$$S = \frac{1 + 0.56}{1 + 0.46} = 1.068$$

Speedup of predicting **not taken** over flushing pipeline

$$S = \frac{1 + 0.56}{1 + 0.38} = 1.130$$

Branching run-time alternatives

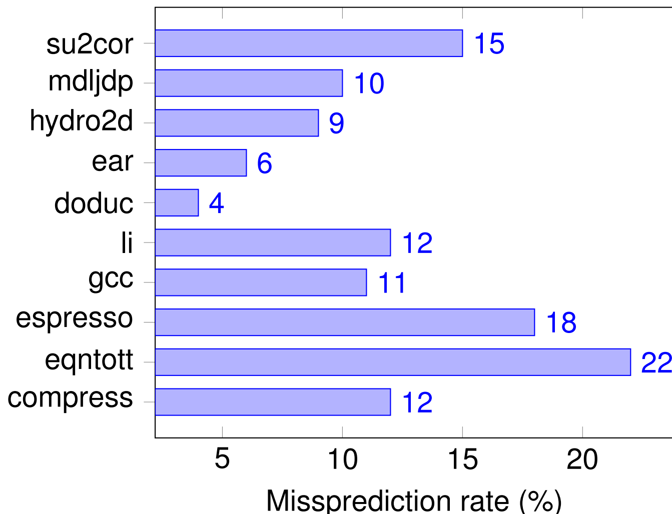
- Each conditional branch is **strongly biased**.
  - Either is taken most of the time,
  - or it is not taken most of the time.

- **Prediction based on execution profile**:
  - Run once to collect statistics.
  - Use the collected information to modify code and take advantage of information.

Predictions with execution profile

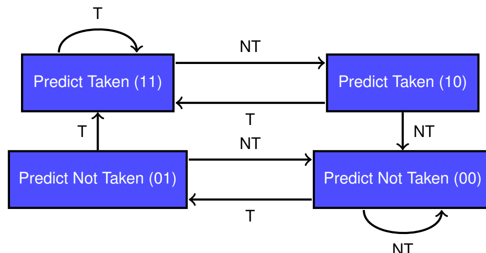
- SPEC92: Branch frequency 3% to 24%
- **Floating point**:
  - **Missprediction rate**.
    - **Average**: 9%.
    - **Standard deviation**: 4%.
- **Integer**:
  - **Missprediction rate**.
    - **Average**: 15%.
    - **Standard deviation**: 5%.

Predictions with execution profile



Dynamic prediction: BHT

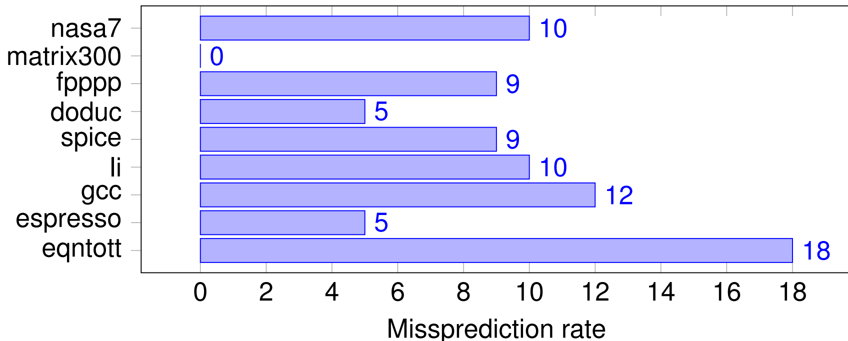
- **Branch History Table**:
  - **Index**: Lower bits of address (**PC**).
  - **Value**: 1 bit (branch taken or not taken last time).



- **Improvements**: Use more bits to improve precision.

BHT: Precision

- Missprediction rate:
  - Wrong prediction in branch outcome.
  - History of different branch in table entry.
- BHT results of 2 bits and 4K entries:



Dynamic branch prediction

- Why does branch prediction work?
  - Algorithms exhibit regularities.
  - Data structures exhibit regularities.
- Is dynamic prediction better than static prediction?
  - It looks like.
  - There is a small number of important branches in programs with dynamic behavior.

1 Introduction to pipelining

2 Hazards

3 Multi-cycle operations

4 Conclusion

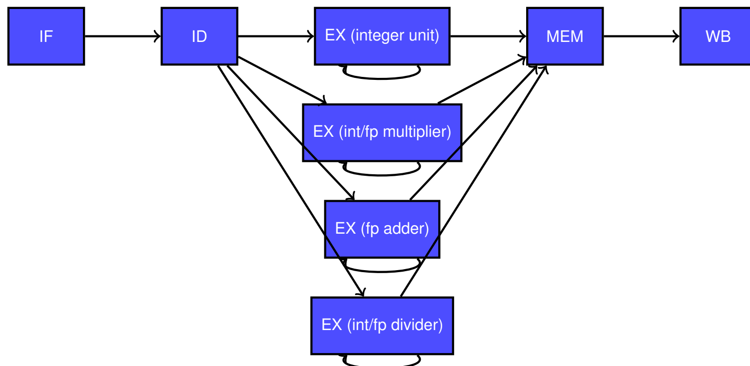
Floating point operations

- One-cycle floating point operations?
  - Having an extremely long clock cycle.
    - Impact on global performance.
  - Very complex FPU control logic.
    - Huge amount of resources for FP logic.

- **Alternative**: Floating point pipelining.
  - Execution stage may be repeated several times.
  - Multiple functional units in EX.
    - **Example**: Integer unit, FP and integer multiplier, FP adder, FP and integer divider.

Floating point pipeline

- EX stage now has a duration of more than 1 clock cycle.



Latency and initiation interval

- **Latency**: Number of cycles between instruction producing the result and instruction using the result.
- **Initiation interval**: Number of cycles between issue of two instructions using the same functional units

Operation	Latency	Initiation interval
Integer ALU	0	1
Loads	1	1
FP addition	3	1
FP multiplication	6	1
FP division	24	25

1 Introduction to pipelining

2 Hazards

3 Multi-cycle operations

4 Conclusion

Summary

- A pipelined architecture requires higher memory bandwidth.
- Pipeline hazards cause stalls.
  - Performance degradation.
- Stalls due to data hazards may be mitigated with compiler techniques.
- Stalls due to control hazards many be reduced with:
  - Compile time alternatives.
  - Run-time alternatives.
- Multi-cycle operations allow for shorter clock-cycles.

References

- **Computer Architecture. A Quantitative Approach**
  5th Ed.
  Hennessy and Patterson.
  Sections C.1, C.2 y C.5

- **Recommended exercises**:
  - C.1, C.2, C.3, C.4 y C.5.

Introduction to instruction level parallelism

Computer Architecture

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Francisco Javier García Blas

ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid