Advanced Computer Architectures

github.com/martinopiaggi/polimi-notes

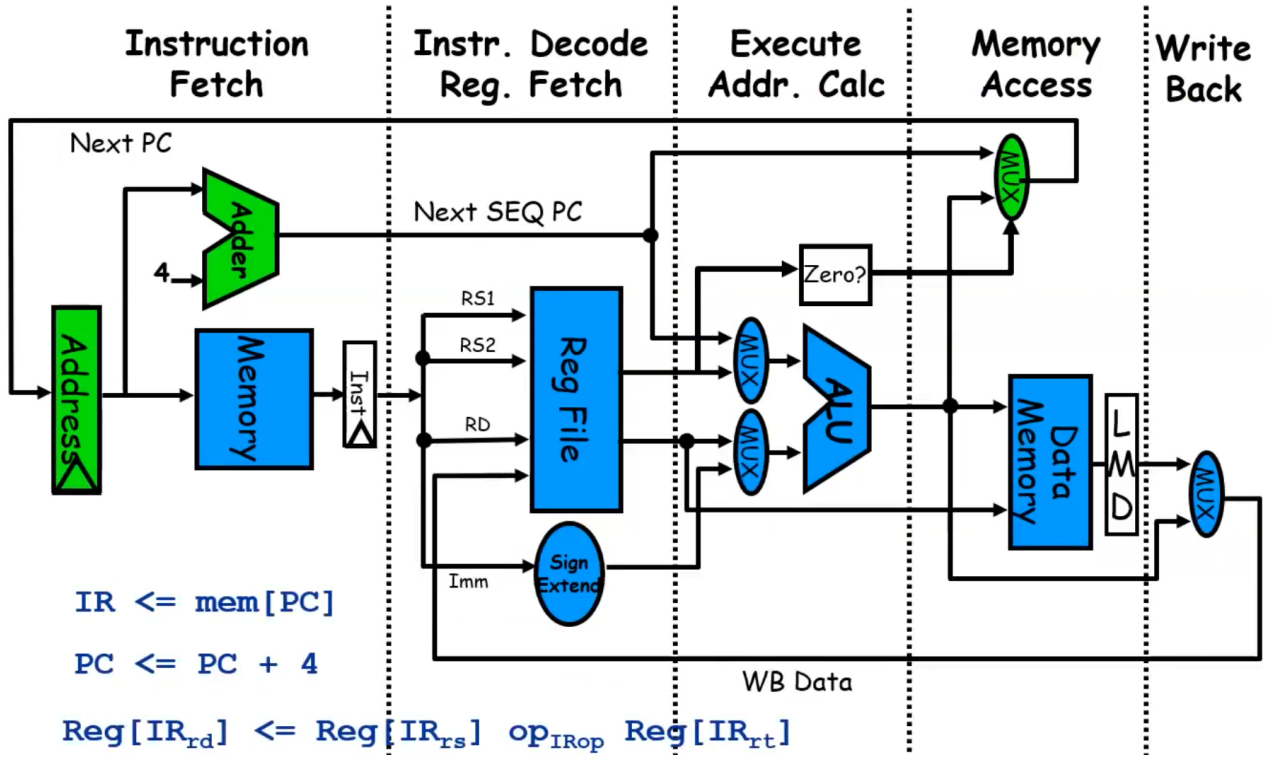
2022-2023

Contents

1 Dependencies and Hazards

1.1 Mips Recalls

Here is how the MIPS data path appears.



A multi-stage pipeline is made by:

- **Fetch** sends the Program Counter content to the memory in order to access the memory location where the instruction is. Then, the `PC` is updated (`PC + 4`, each instruction take `32` bits).
- **Decode**, determines operations and reads the registers needed for the computation.
- **Execution**, arithmetic logic operations occur
- **Memory**, accesses memory or cache hierarchy for LOAD or STORE instructions.
- **Write back**, result of computation is written to register file.

1.2 Hazards

- **Structural Hazards**: Attempt to use the same resource from different instructions simultaneously. Example: Single memory for instructions and data
- **Data Hazards**: Attempt to use a result before it is ready. Example: Instruction depending on a result of a previous instruction still in the pipeline
- **Control Hazards**: Attempt to make a decision on the next instruction to execute before the condition is evaluated Example: Conditional branch execution. Caused by delay between the fetching of instructions and decisions about changes in control flow (branches, jumps, exceptions)

1.2.1 Dependencies

Hazards are pipeline properties while dependences are program properties.

Dependences among instructions are crucial for determining the amount of parallelism in a program: if 2 instructions are dependent, they cannot be executed in parallel and must be executed in order or at least partially overlapped. 3 different types of dependences:

- **Name Dependences**: Name dependence occurs when 2 instructions use the same register or memory location (called name), but there is no flow of data between the instructions associated with that name. These kind of dependencies are "easily" fixed with register renaming.
- **Data Dependences** (or **True** Data Dependences)
- **Control Dependences**

Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls Ideal pipeline CPI: measure of the maximum performance attainable by the implementation

2 Performance Evaluation and Metrics

$$IC = \#\text{ instructions}$$

$$CPI = \frac{\text{CC}}{\text{IC}} = \frac{\text{clock cycle}}{\#\text{ instructions}}$$

$$MIPS = \frac{\text{clock frequency}}{CPI{*}10^6}$$

Ahmdal's law , this doesn't tell anything on the performance of the CPU still depends on its frequency.

$$\text{SPEED UP} = SU = \frac{1}{1 - (\text{fraction enhanced}) + (\frac{\text{fraction enhanced}}{SU_{\text{fraction enhanced}}})}$$

To sum up, Amdahl's law states that, for a fixed problem, the upper limit of speedup is determined by the serial fraction of the code -> strong scaling

Gustafson law revises it:

$$SU_{GUS} = \frac{1}{\text{sequential} + \frac{\text{parallel}}{N}}$$

X is n times faster than Y means:

$$\frac{\text{Performance}(X)}{\text{Performance}(Y)} = \frac{\text{Exe}(Y)}{\text{Exe}(X)}$$

The CPU time can be computed like this:

$$\text{CPU time } = \left(\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i\right) \times \text{ Clock cycle time}$$

so, X is n times faster than Y means:

$$\frac{EXE_{CPU_1}}{EXE_{CPU_2}} = \left(\frac{IC_1 * CPI_1}{F_1}\right) * \left(\frac{F_2}{IC_2 * CPI_2}\right)$$

Floating-point operations per second (FLOPS) is **a measure of compute performance used to quantify the number of floating-point operations a core, machine, or system is capable of in a one second**.

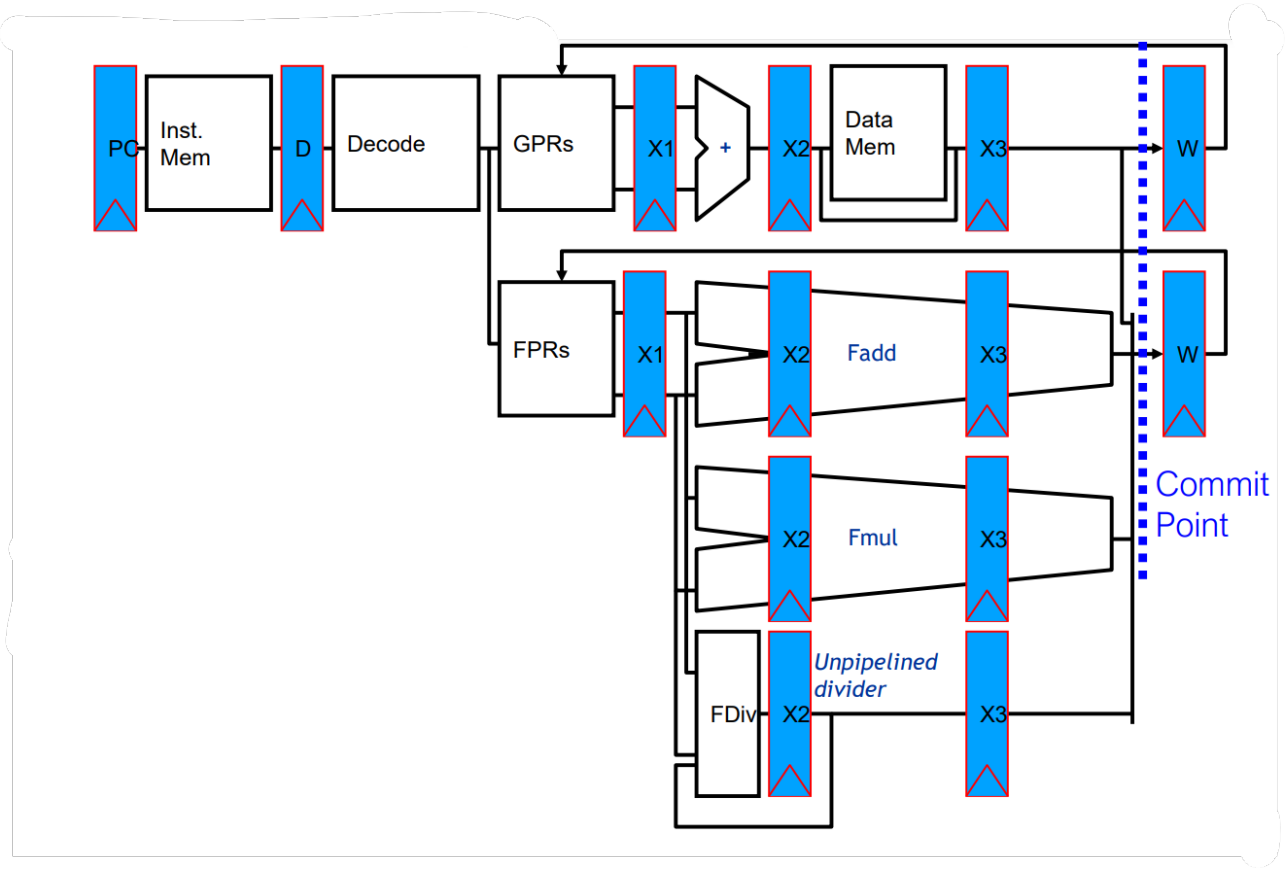
$$\text{Pipeline Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Pipe Stall Cycles per Instruction due to Branches}}$$

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

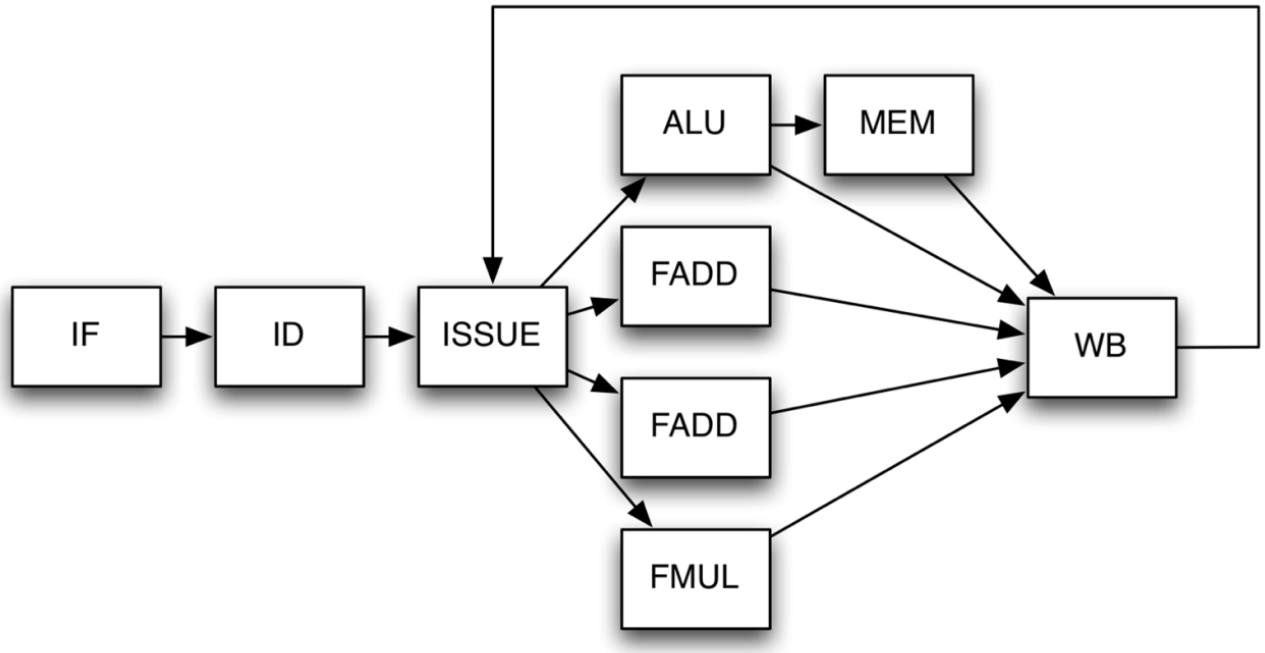
3 Complex Pipeline

Moving towards to a more complex pipeline: pipelining becomes complex when we want high performance in the presence of:

- Long latency or partially pipelined **floating-point units**
- **Multiple** function and memory **units**
- Memory systems with **variable access time**
- Precise **exception** (divisions by zero, underflow and overflow at hardware level)



General purpose registers (**GPRs**) contain information being manipulated by the user program currently running. Floating-point registers (**FPRs**) hold numeric values associated with some exponent.



3.1 Complex Pipeline main features

- All functional units are **pipelined**
- Instructions are fetched, decoded and issued in order (always in this course)
- **Issue** stage can be seen as an infinite buffer.
- Generally it's assumed that the WB unit has a single write port
- RF can write first half of the CC and Read in the second half. (so IS aligned with WB)
- Only one instruction can be issued at a time, and in the case multiple instructions are ready, the oldest one will go first

3.1.1 What to remember in the Complex Pipeline

- **Decode stalling** if it is causing a **WAR** or **WAW** hazard.
- **Issue stalling** if **RAW**
- **EX stalling** iff **structural** hazard on writing port

Example:

Instruction	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
`LD F1, 0(R1)`	F	D	IS	E1	E2	E3	WB					
`LD F2, 0(R1)`		F	D	IS	E1	E2	E3	WB				
`ADDD F1, F1, F2`			F	sD	sD	D	sIS	IS	E1	E2	E3	WB
`ADD R1, R1, 8`				sF	sF	F	D	sIS	IS	E	WB	

4 Branch Prediction Techniques

To make the best guess at branch direction, speculation is used.

4.1 Static Branch Prediction Techniques

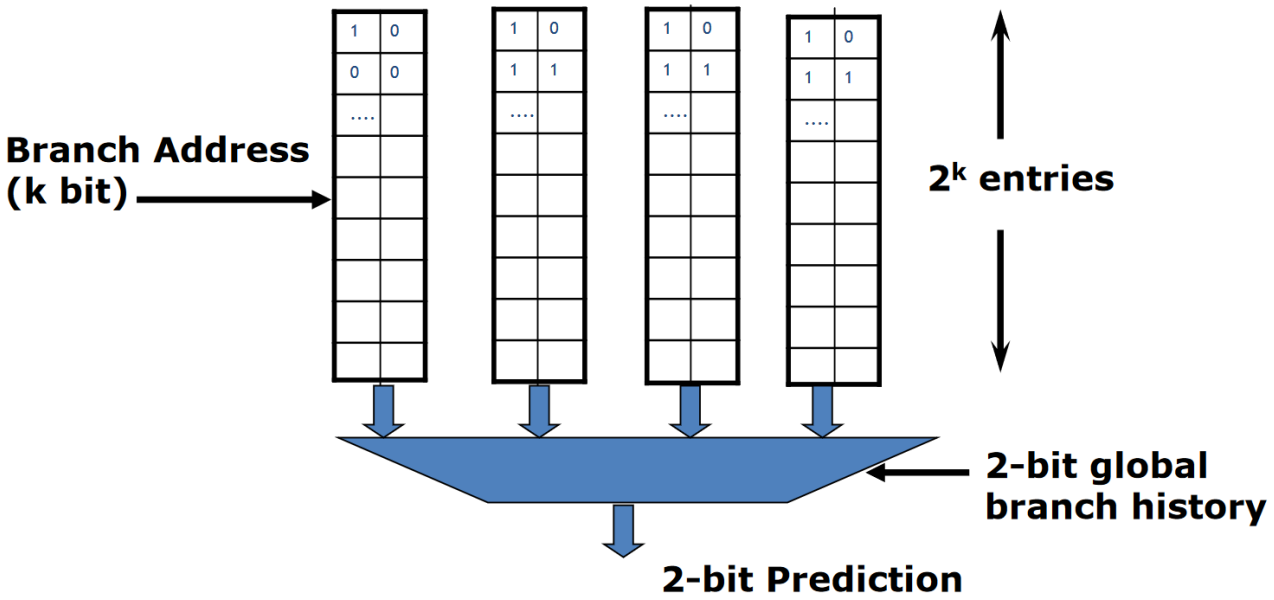
With Static Branch Prediction the actions taken by the CPU at each branch are fixed at **compile** time. This technique is used in processors where the expectation is that the branch behavior is highly predictable at compile time. Five static branch prediction techniques:

- Branch Always Not Taken (**Predicted-Not-Taken**)
- Branch Always Taken (**Predicted-Taken**)
- Backward Taken Forward Not Taken (**BTFNT**): examples are all the branches at the end of loops, it's assumed the backward-going branches are always taken.
- **Profile-Driven Prediction**: The branch prediction is based on profiling information collected from earlier runs. The method can use compiler hints.
- **Delayed Branch**:The compiler statically schedules an independent instruction in the branch delay slot. If we assume a branch delay of one-cycle (as for MIPS), we have one-delay slot. Three different ways:
  - **From before**: an instruction is taken (which it's known to not affects the data flow) from before the branch instruction and it's executed not before but after the branch instruction.
  - **From target**: an instruction is taken from the target of the branch. Usually used when the branch is taken with high probability, such as loop branches (backward branches).
  - **From fall-through**:This strategy is preferred when the branch is not taken with high probability, such as forward branches.

In general, the compilers are able to fill about 50% of delayed branch slots with valid and useful instructions, the remaining slots are filled with NOPs. The main limitations on delayed branch scheduling arise from: the restrictions on the instructions that can be scheduled in the delay slot.

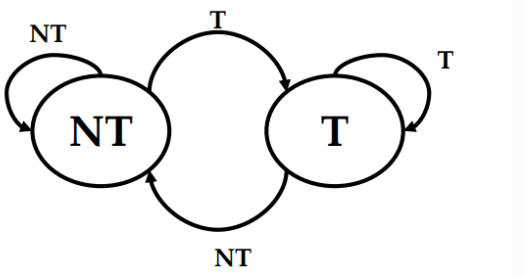
4.2 Dynamic Branch Prediction Techniques

Dynamic branch prediction helps to predict the outcome of a branch instruction dynamically adapting to the program, using the past behavior. The idea is to use a **Branch History Table**. A **BHT** is a "cache" which contains 1 (or 2) bits for each entry (which is an hash of the branch address (indeed since it is an hash function, collisions are possible)) to indicate if the branch was recently taken.

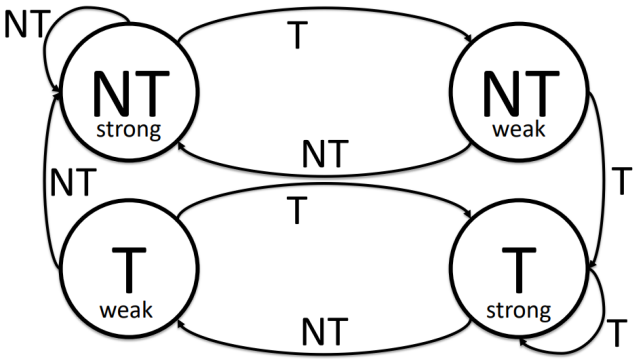


4.3 Branch History Table

A simple version of BHT uses a 1-bit saturating counter:



A 2-bit saturating counter is a state machine with four states.

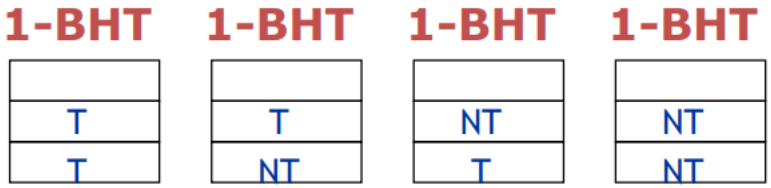


Generally if there are nested loops (like 99% of code) the 2-BHT is a better option since it's proved that in average gives better performance. In this case, the holy rule is:

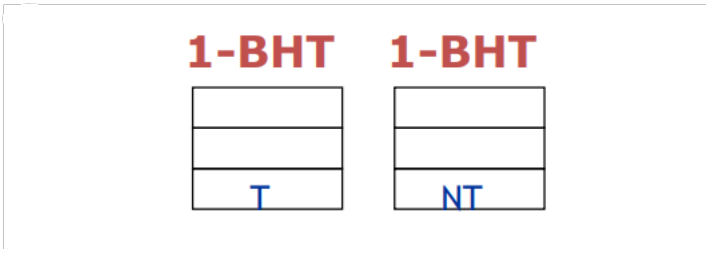
> "Best of 1-BHT is worst than worst-case of 2-BHT"

However, in **no**-nested loops situations, 1-BHT is not always outperformed by the 2-BHT!

We can both consider any problem with a collision between the branch addresses or not. If 2 branch addresses do not collide:



Otherwise the possible initializations are:



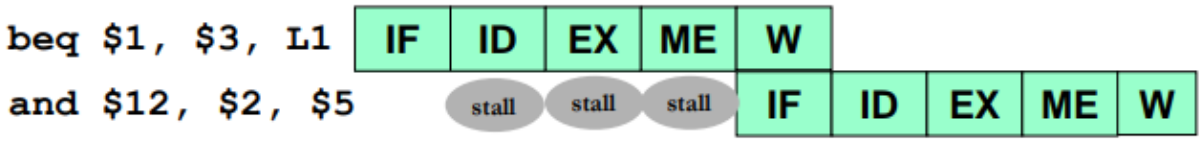
A further expansion of this idea is that **recent branches may exhibit correlated behavior**. The n-bit-BHT

technique only considers the branch's history, while an (m,n) **Correlating Branch** predictor takes into account the behavior of the previous m branches. Both a Branch History Register (BHR) and a Pattern History Table (PHT) can be used, with the PHT lookup selecting the most similar pattern to the recent branch execution flow and predicting based on that BHR.

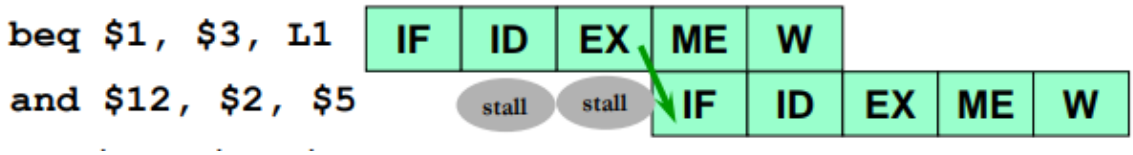
4.4 Early Evaluation

The MIPS architecture studied so far can't solve a branch hazard without introducing stalls. Branch prediction predicts the outcome of a branch, but knowing the true outcome **asap** is even more useful for better performance.

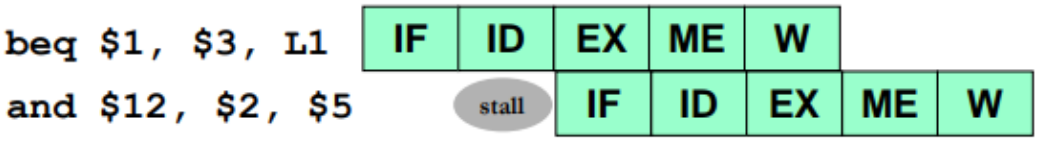
- **Without forwarding** the branch outcome is available only during MEM stage, so **3 stalls needed**.



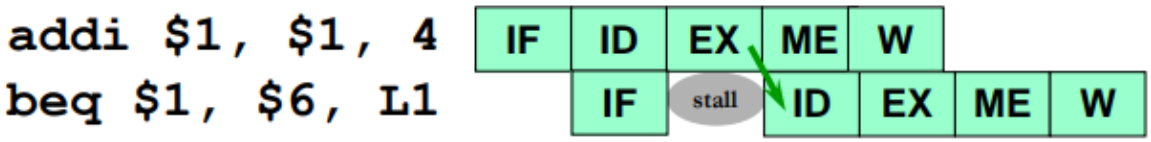
- **With forwarding** we can take the branch outcome calculated at the end of EX stage and forward it to the IF requiring 2 stalls.



- **With early evaluation**, we can calculate the branch outcome in the ID stage by adding additional hardware resources. The outcome of the branch condition is going to be available in the ID stage and only 1 stall is needed.



But remembers that Early Evaluation has a problem with the previous instruction.



And also if the previous instruction is a **load** instruction, **2 stall** cycles are needed in order to allow MEM/EX forwarding the result of the load.

5 Exceptions Handling

5.1 Interrupts

Interrupts are signals that alter the normal control flow of a program at runtime. An **interrupt** must be handled by a dedicated **interrupt handler**, and the architecture should be able to resume the normal execution of the program. Causes:

- **Asynchronous** external events:
  - I/O device reply
  - timer expiration
  - HW failure
- **Synchronous** internal event:
  - opcode error
  - arithmetic overflow
  - bad memory access
  - traps: traps are a mechanism used by computers to transfer control to the OS or to kernel routines in response to various events like system calls.

Interrupts can be characterized in various ways:

- **Async** interrupts are caused by external sources and can be handled at the end of the execution of the current instruction, making them easier to manage than **sync** ones.
- **User request** interrupts are predictable: they are treated as exceptions because they use the same mechanism. **Coerced interrupts** (forced) are caused by some HW event not under control of the program.
- **User maskable** vs **user nonmaskable**
- **Within instructions** interrupts are usually synchronous since the instruction triggers the exception. The instruction must be stopped and restarted.
- **Resume** vs **terminate** interrupts

5.2 Super exception table

Exception type	Synchronous vs. asynchronous	User request vs. coerced	Oser 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					
Breakpoint Integer arithmetic overflow	Synchronous	User request	User maskable	Between	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Mischaligned 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

6 VLIW

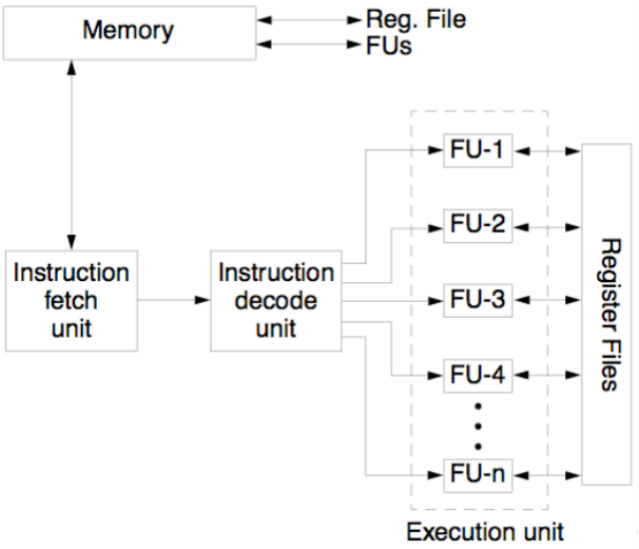
6.1 Static Scheduling intro

Two strategies to support ILP: Static and Dynamic scheduling. VLIW is an example of **Static Scheduling**: it relies on software for identifying potential parallelism. There are **limits of static scheduling**:

- **Unpredictable Branches**: the compiler can't know ahead of time which branch will be taken.
- **Variable Memory Latency**: compiler can't predict which memory blocks will be in cache at any given time. So it cannot optimize the instructions that rely on the data in memory.
- **Code Size Explosion**: Static scheduling can also result in a code size explosion. As we schedule instructions, we may need to insert more instructions to handle dependencies and ensure proper execution order.
- **Compiler Complexity**: static scheduling can also increase compiler complexity. The compiler must analyze the code for dependencies and optimize the instruction scheduling accordingly.

6.2 VLIW

In VLIW, there is a fetch stage where the bundle is fetched from memory, and a decode stage where the operations within the instruction are separated and sent to functional units (FUs). VLIW architecture uses a singular instruction bundle called Very Long Instruction, which results in the requirement of only one **Program Counter**. As conflicts are resolved by the compiler statically, the instruction is executed once everything required is already available.



6.2.1 What to remember in VLIW

- Both pipeline or unpipelined versions exist.

- WAR and WAW in the same clock if they are on different units.
- Both in-order and out-order issue exist.

6.2.2 VLIW: Pros and Cons

Pros:

- Simple HW: simpler hardware compared with superscalar processors. The hardware does not need sophisticated scheduling logic since it simply executes each operation according its position within the word without checking for dependencies among them.
- Easy to extend the #FUs: utilize all FUs in each cycle (in VLIW) as much as possible to reach better ILP and therefore higher parallel speedups.
- Good compilers can effectively detect parallelism

Cons:

- Huge number of registers to keep active the FUs
- Large data transport capacity between FUs and register files and between register files and memory
- High bandwidth between i-cache and fetch unit: the bus bandwidth should be capable of transferring an instruction that contains n operations, requiring a bandwidth of `n*(operation size)`.
- large code size since each instruction word contains many operations.
- binary compatibility

6.3 Static Scheduling methods

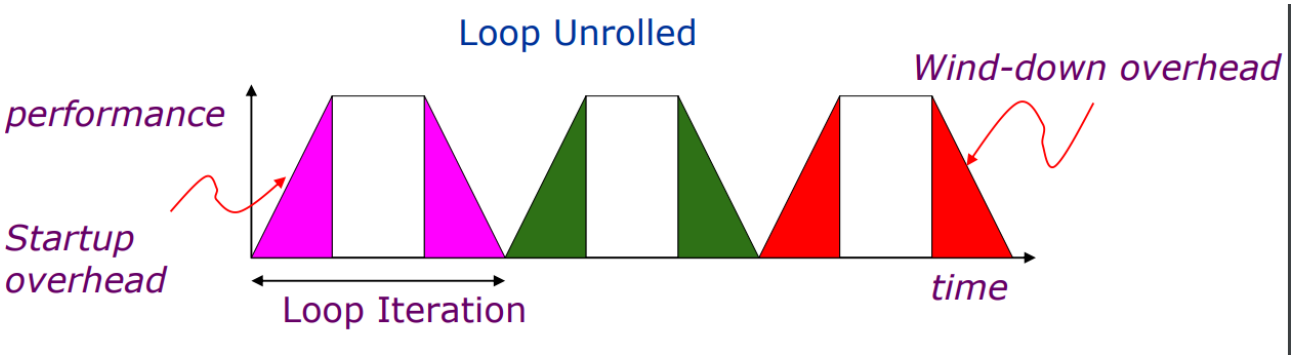
There are many different Static Scheduling methods that can be applied:

- **Simple code motion**: just moving code instructions during scheduling.
- **Loop unrolling & loop peeling**: Unrolling exposes more computation that can be scheduled to minimize stalls. If there aren't loop-carried dependencies we can "unroll" the iterations of any loop and perform them in a parallel way.
- **Software pipeline**
- **Trace scheduling**

All these techniques have lower power consumption compared to dynamic scheduling techniques.

6.3.1 Loop unrolling

Loop Unrolling performance can be enhanced by rescheduling. Anticipating the next loop iteration at a scheduling level helps in this. This is done by anticipating operations to free slots in the pipeline that can be used to start up the next iteration. **Compensation code** may need to be added: for example the compensation code to decrease a counter is added after it has been previously incremented.




```
loop:   ld f1, (r1)
        ld f2, 0(r2)
        fmul f1, f1, f1
        fadd f1, f1,f2
        st f1, 0(r3)
        addi r1, r1,4
        addi r2, r2,4
        addi r3, r3,4
        bne r3, r4, loop
```

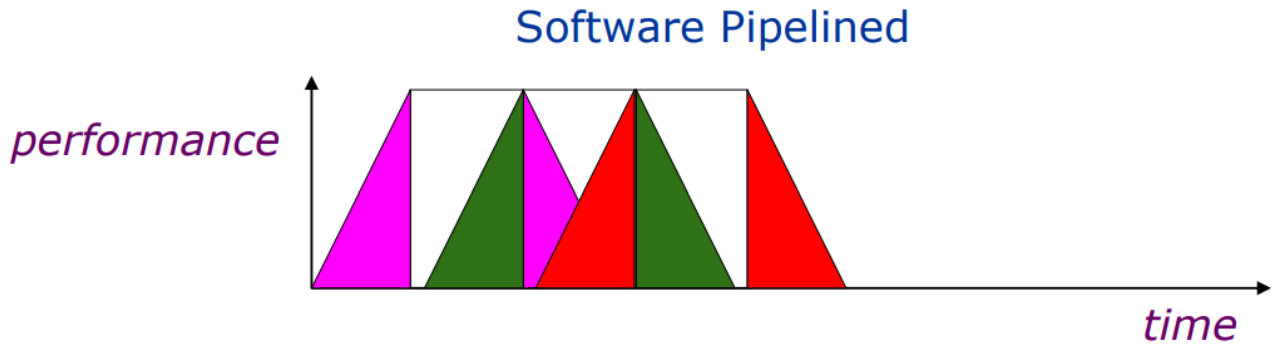
For example here, to optimize the code, we can unroll the loop by one iteration, performing two iterations of the original loop. It's not considered the prologue or epilogue.

```
loop:   ld f1, (r1)
        ld f3, 4(r1)
```

```
ld f2, 0(r2)
ld f4, 4(r2)
fmul f1, f1, f1
fmul f3, f3, f3
fadd f1, f1,f2
fadd f3, f3, f4
st f1, 0(r3)
st f3, 4(r3)
addi r1, r1,8
addi r2, r2,8
addi r3, r3,8
bne r3, r4, loop
```

6.3.2 Software pipeling

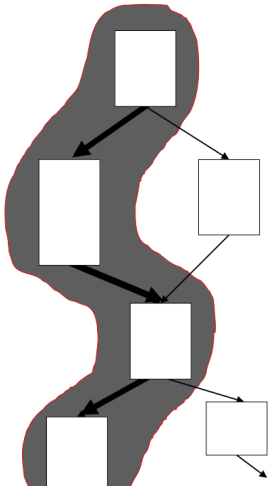
Software pipelining pushes even further this idea, permitting the loop iterations that are executed parallelly to also be pipelined.



The main difference between them is that loop unrolling replicates the entire code of the loop, while software pipelining can be thought of as "symbolic" loop unrolling. This means that loop unrolling needs to add some startup code before the loop as well as finish-up code at the end, whereas software pipelining does not.

6.3.3 Trace Scheduling

A trace is a sequence of basic blocks, it can be seen as a branch path. Trace scheduling aims to execute and profile the application to find the most probable traces to be executed. Each trace is then considered as a single one, allowing for heavy rescheduling of the entire operation set to improve performance.



Trace scheduling cannot proceed beyond a **loop barrier**. It's also important to say that TS can add **compensation code** (like loop unrolling) at the entry and exit of each trace to compensate for any effects that out-of-order execution may have had.

7 Scoreboard

7.1 Dynamic Scheduling intro

Two strategies to support ILP: Static and **Dynamic Scheduling**. Dynamic Scheduling depends on the hardware to locate parallelism. The hardware reorders the instruction execution to reduce pipeline stalls while maintaining data flow and exception behavior. Main advantages:

- It enables handling cases where dependences are unknown at compile time
- It simplifies the compiler complexity
- It allows compiled code to run efficiently on a different pipeline.

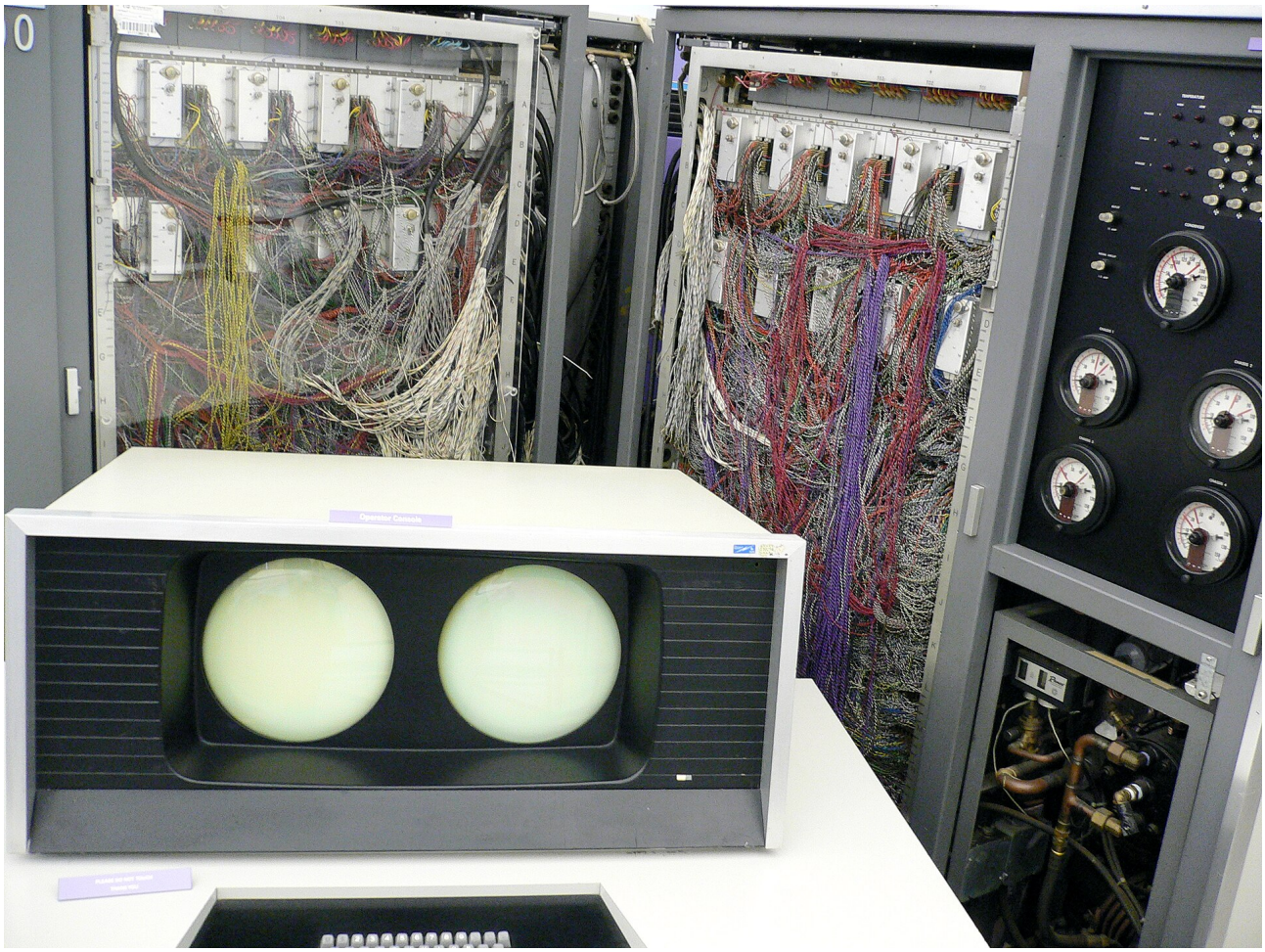
Those advantages are gained at a cost:

- More HW complexity
- Increased power consumption
- Could generate imprecise exception

"If an instruction is stalled in the IS stage, it implies that it has entered the issue stage and is causing a WAR problem. During exams, it's preferred to write"I" when an instruction enters the issue stage. The method of stalling IS is only used for RAW data dependencies. Stalling for every RAW in D can prevent the pipeline from executing other instructions, thus it's better to stall for WAR and WAW name dependencies only."

7.2 Scoreboard

The CDC 6600 introduced the first scoreboard in 1964.



7.2.1 Scoreboard main features

- complex pipeline with “ISSUE”
- In order issued
- out of order execution
- out of order committed (writebacked)
- no forwarding
- control is centralized into the **Scoreboard**

Scoreboard data structure tracks functional unit status and register result status:

- `busy` indicating unit status
- `Op` for operation type
- `Fi` for destination register
- `Fj` and `Fk` for source registers
- `Qj` and `Qk` for the FUs outputting the source registers
- `Rj` and `Rk` for flags indicating if the `Fj` and `Fk` registers are ready.

7.2.2 Four Logical Stages of Scoreboard Control

- Issue
- Read Operands

- Execution completion
- Writeback

7.2.3 What to remember in Scoreboard

- **Stalling ISSUE** stage is used to avoid **WAW** and structural hazards
- **Stalling Read Operands** stage is used to avoid **RAW** hazards. Results available the **cycle after the WB**.
- Exe is never stalled
- **Stalling WB** if a **WAR** is detected.

7.2.4 Simplified view

ISSUE	READ OPERAND	EXE COMPLETE	WB
Decode instruction Structural FUs check WAW checks	Read operands WAR if need to read RAW check	Operate on operands Notify Scoreboard on Completion	Finish exec WAR and **Struct check** (FUs will hold results) Can overlap issue/read\write 4 Structural Hazard;

Example:

	Issue	Read Op	Exec Co.	Write R.
LD F6 32+ R2	1	2	3	4
LD F2 45+ R3	5	6	7	8
MULTD F0 F4 F2	6	9	19	20
ADD F2 F8 F6	9	10	11	12
DIVD F12 F0 F6	10	21	31	32
SUBD F6 F8 F2	11	13	14	22

7.3 Scoreboard with renaming

7.3.1 Register renaming from Wikipedia:

```
r1 = m[1024]
r1 = r1 + 2
m[1032] = r1
r1 = m[2048]
r1 = r1 + 4
m[2056] = r1
```

The instructions in the final three lines are independent of the first three instructions, but the processor cannot finish `r1 = m[2048]` until the preceding `m[1032] = r1` is done. This restriction is eliminated by changing the names of some of the registers:

```
r1 = m[1024]
r1 = r1 + 2
m[1032] = r1
r2 = m[2048]
r2 = r2 + 4
m[2056] = r2
```

7.3.2 Simplified view

ISSUE	READ OPERAND	EXE COMPLETE	WB
Decode instruction allocate new physical register for result	Read operands	Operate on operands	Finish exec
Structural FUs; free physical registers check	RAW check	Notify Scoreboard on completion	Struct check (FUs will hold results); Can overlap issue/read&write

- **register renaming** is a technique that abstracts logical registers from physical registers. Every logical register has a set of physical registers associated with it.
- Register Renaming allows to avoid WAR and WAW hazards
- The concept explained in spaghettata mode: "I'm a FU and I'm waiting for an operand which is used atm by another FU. I can **abstract** over the register and use something like a **pointer**"

8 Tomasulo

In Tomasulo the control unit is decentralized by distributing buffers among functional units, which makes it more efficient. Reservation stations play a key role in this approach by allowing for register renaming.

8.1 Tomasulo main features

- in-order issue
- out-of-order execution
- out-of-order completion
- register renaming based on reservation stations to avoid war and waw hazards
- results dispatched through the common data bus CDB

Tomasulo nomenclature of paramers:

- `Vj` and `Vk` for source values
- `Qj` and `Qk` for the RSs

8.2 Three Logical Stages of Tomasulo

- **Issue**: it sends operands to the reservation station and performs register renaming.
- **Execution**: executes when operands on CDB are available
- **Write result**: When result is available, write it to the CDB

8.2.1 What to remember in Tomasulo

- **Stalling ISSUE** stage is used to avoid **RAWs** and **structural** hazards caused by the absence of RSs or parallel write on **CDB
- **Stalling WB** if a **structural** hazard over **CDB** is detected: not possible to access the CDB at the same clock cycle with 2+ different instructions.

8.3 Simplified view

ISSUE	EXECUTION	WRITE
Get Instruction from Queue and Rename Registers	Execute and Watch CDB	Write on CDB
Structural RSs check WAW and WAR solved by Renaming (!!!in-order-issue!!!)	Check for Struct on FUs RAW delaying Struct check on CDB	(FUs will hold results unless CDB free) RSs/FUs marked free

Example with:

- RS1, RS2 + 1 LDU (3cc)
- RS3, RS4 + 1 MUL (11cc)
- RS5 + 1 ADD (5cc)
- Consider structural hazards for RS, FU and CDB.

Instruction	ISSUE	START EXE	WB
I1: LD F6 32+ R2	1	2	5
I2: LD F2 45+ R3	2	6	9
I3: MULTD F0 F4 F3	3	4	15
14: ADD F8 F2 F6	4	10	16
15: DIVD F12 F8 F0	5	17	28
16: SUBD F8 F6 F2	17	18	23

- **waw** and **war** are automagically solved by renaming!!!! we keep only the RAWs !!

Other example:

Instruction	ISSUE	START EXE	WB	Hazards Type	RSi	Unit
I1: LD F1, 0(R1)	1	2	5		RS1	LDU1
I2: FADD F2, F2, F3	2	3	6		RS4	FPU1
I3: ADDI R3, R3, 8	3	4	7	Struct CDB	RS7	ALU1
I4: LD F4, 0(R2)	4	5	8		RS2	LDU2
15: FADD F5, F4, F2	5	9	1 2	RAW F4, RAW F2	RS5	FPU1
I6: FMULT F6, F1, F4	6	9	1 3	RAW F4, Struct CDB	RS6	FPU2
17: ADDI R5, R5, 1	7	8	9		RS8	ALU1
I8: LD R6, 0(R4)	8	9	1 4	Struct CDB	RS1	LDU1
I9: SD F6, 0(R5)	9	1 4	1 7	RAW F6	RS2	LDU2
I10: SD F5, 0(R6)	1 0	1 5	1 8	RAW F5, RAW R6	RS3	LDU1

8.4 Tomasulo with ROB

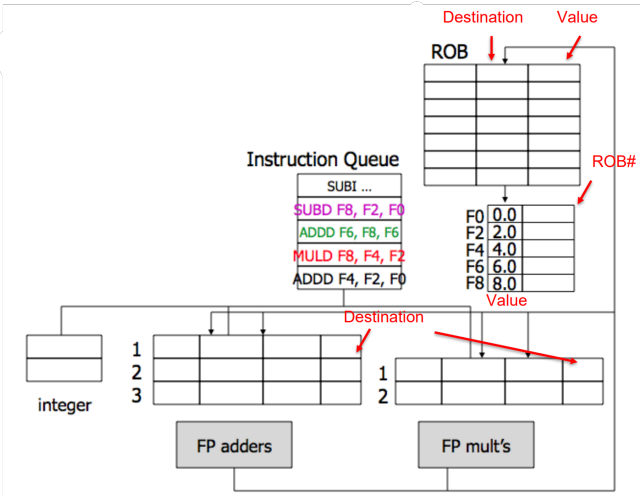
8.4.1 HW-based speculation

HW-based Speculation combines what we have seen so far in this course + Speculation:

- Dynamic Branch Prediction
- Dynamic Scheduling
- Speculation

Speculation consists to issue and execute instructions dependent on a branch before the branch outcome is known to allow instructions to execute out-of-order but to force them to commit in-order.

8.4.2 ROB



Out of order means always "buffer". In this case we use a "**Reorder Buffer**" which holds instructions in FIFO order. Each entry holds program counter, instruction type, destination register specifier and value if any, and exception status (info often compressed to save hardware). The reorder buffer is added over Tomasulo Algorithm:

1. Issue:
   - Get instruction from FP Op Queue
   - If reservation station and reorder buffer slot are free, issue instruction and send operands and reorder buffer number for destination.
   - This stage is sometimes called "dispatch".
2. Execution:
   - Operate on operands (EX)
   - When both operands are ready, then execute.
   - If not ready, watch CDB for the result.
   - When both are in reservation station, execute.
   - Checks RAW
   - This stage is sometimes called "issue".
3. Write result:
   - Finish execution (WB)
   - Write on Common Data Bus to all awaiting FUs and reorder buffer.
   - Mark reservation station available.
4. **Commit**:
   - Update register with reorder result
   - When instruction at the head of the reorder buffer and result present, update register with the result (or store to memory).
   - Remove instruction from reorder buffer.
   - Mispredicted branch flushes reorder buffer.

"Instead of processing instructions strictly in the sequential program order, the CPU executes them as soon as all required resources are available. While the execution unit of the current operation is occupied, other execution units can run ahead."

With Tomasulo we are still bounded by branches ! Tomasulo permits out of order execution of instruction that **surely** are executed. ReOrder Buffer permits to handle branches using speculation, since the fundamental concept is to have a buffer where we can temporally store already executed instruction that are committed depending on the outcome of the branch. So, to recap Tomasulo with ROB is possible to make **dynamic loop unrolling** since it's easy:

- Undo speculated instructions on mispredicted branches
- Manage exceptions precisely

8.4.3 Simplified view of TOMASULO with ROB

ISSUE	EXECUTION	WRITE	COMMIT
Get Instruction from Queue and Rename Registers Add Instruction to ROB	Execute and Watch CDB	Write on CDB Write on ROB	Update register with result (or store to memory) remove Instr from ROB
Structural RSs check Structural ROB check WAW and WAR solved by Renaming (!!!in-order- issue!!!)	Check for Struct on FUs RAW delaying Struct check on CDB	(FUs will hold results unless CDB free) RSs/FUs marked free	In-order commit Mispredicted branch flushes ROB

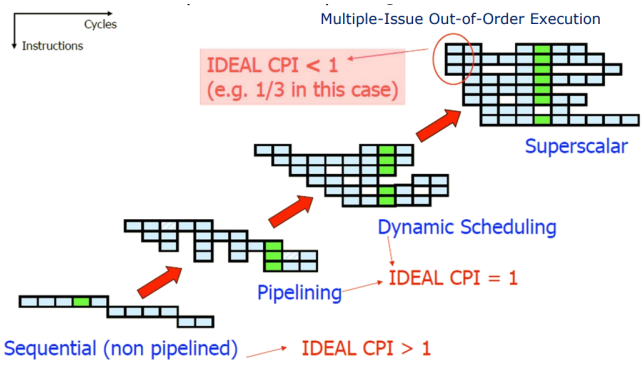
- waw and war are automagically solved by renaming!!!! we keep only the RAWs !!
- when an instruction is in the commit stage, other instructions can be executed
- in-order commit using the "index" of ROB
- remember to check **struct hazards** also for the **ROB**

9 Multithreading Architectures

9.1 Superscalar architectures

A scalar architecture is a single issue architecture while superscalar architectures allow for multiple instructions to be executed per clock cycle. Remember that ILP $\neq$ parallel processing:

- ILP improves throughput by pipelining operations
- Parallel processing is non-user-transparent way of executing programs



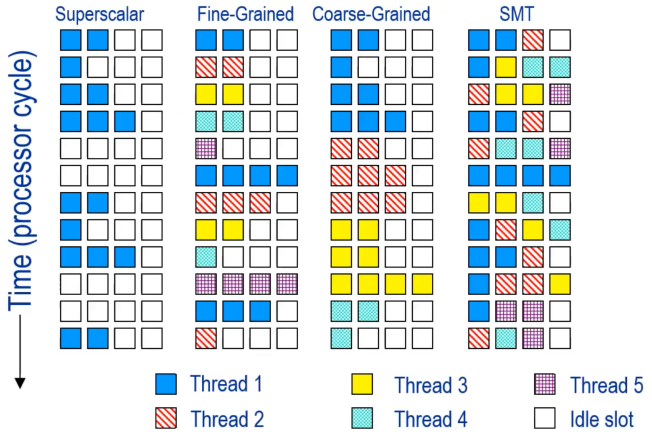
Superscalar architectures have limitations. ILP has an upper bound:

- Dynamic scheduling is expensive and difficult to design and verify
- Register renaming has its limits
- Jump and branch predictions are not always accurate
- Memory latency is a huge issue.
- Hazards prevent too many instructions from being issued simultaneously.

9.2 Temporal Multithreading

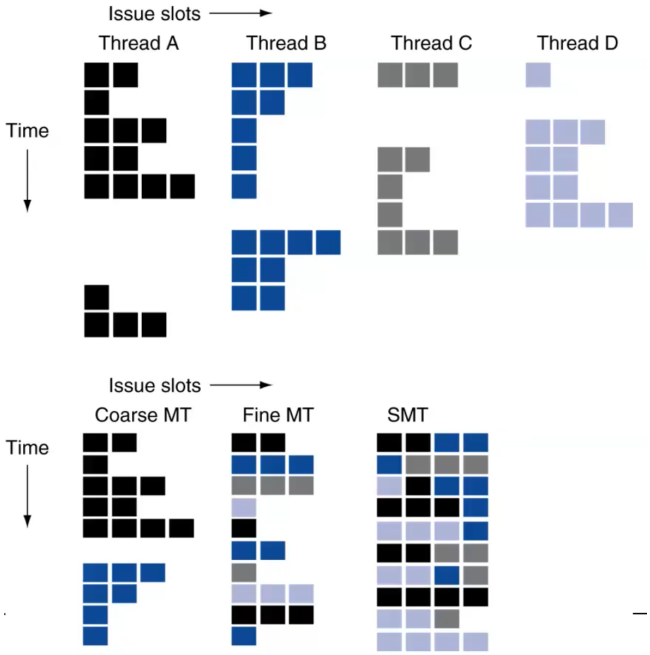
Using Temporal Multithreading is possible to further increase parallelism by alternately switching between tasks. Temporal Multi-threading has a high cost due to each thread having its own context and there are different techniques to achieve it:

- **Fine-grained** multi-threading involves switching between different threads at a very fine level of granularity, such as after every instruction. This allows for maximum utilization of resources but can also lead to increased overhead due to **frequent context switching**.
- **Coarse-grained** multi-threading involves switching between different threads at a coarser level of granularity, such as after completing a group of instructions from one thread before moving on to another thread. This reduces the overhead associated with context switching but may not fully utilize available resources.
- **Simultaneous multi-threading** (SMT) is similar to fine-grained multi-threading but goes further by allowing multiple instructions from different threads to be executed simultaneously within each clock cycle.



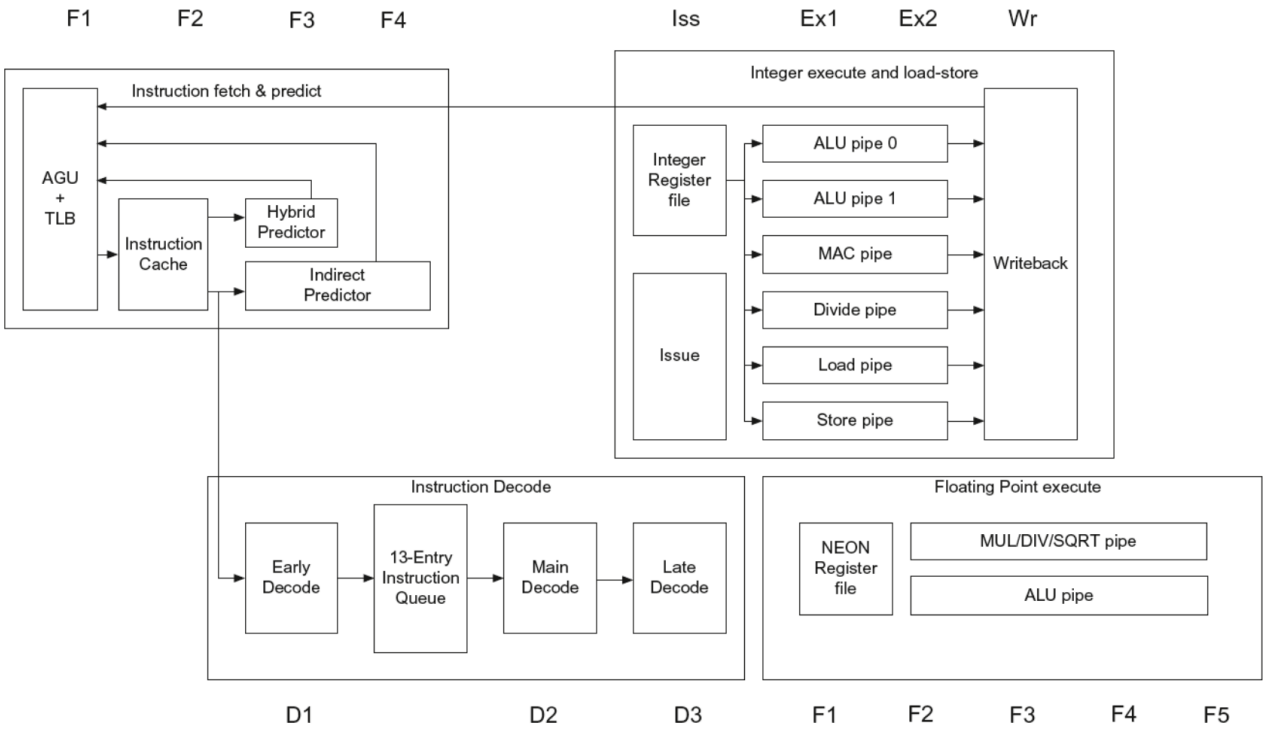
There are drawbacks in each approach:

- **Fine-grain** multithreading has empty issue windows, causing idle time.
- **Coarse-grained** multithreading architecture is not commonly used, since it requires flushing the pipeline and can result in starvation.
- **SMT** requires more complex hardware than either fine- or coarse-grained multithreading but can provide higher levels of parallelism and better resource utilization.



9.3 ARM Cortex-a53 pipeline example

What actually happens in real world? Basically different systems are combined to achieve the most performance: for example cpu alone is not enough for tasks such as playing games in 4K or streaming on Twitch: GPUs are a common addition to systems. This is called **heterogeneous architecture** since it combines different types of processors. An example of heterogeneous architecture is the big.LITTLE architecture. Cortex-a53 (also the Raspberry Pi 3 CPU) can be used alone as an energy-efficient alternative to the Cortex-A57, or in a big.LITTLE configuration alongside a more powerful microarchitecture. The big.LITTLE configuration architecture is multicore and heterogeneous with a shared ISA, combining battery-efficient (LITTLE) and power-hungry (big) cores. Typically, only one side will be active, but all cores can access the same memory.



10 Parallel Architectures

Parallel Programming

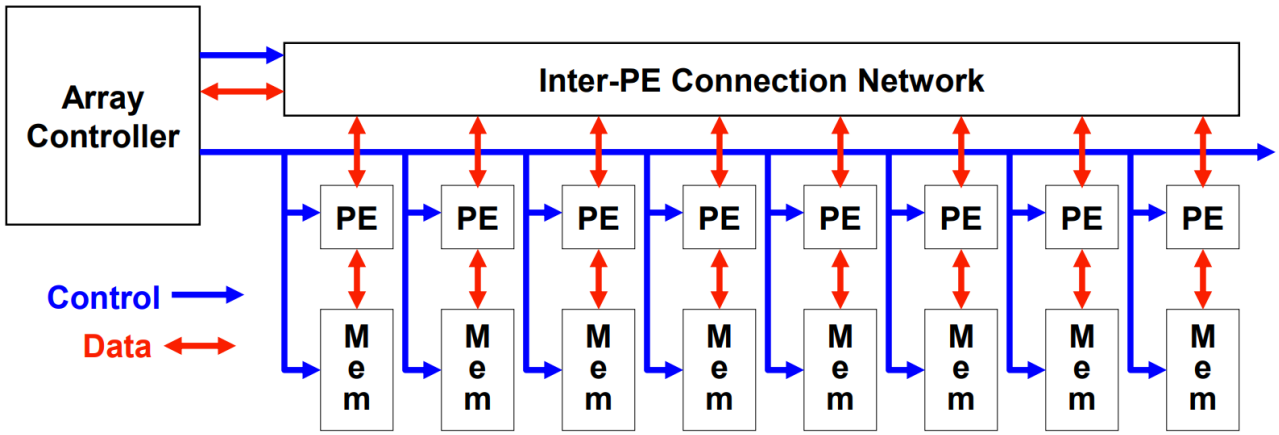
10.1 Flynn taxonomy recall

- SISD - Single Instruction Single Data : Uniprocessor systems
- MISD - Multiple Instruction Single Data: no practical configuration and no commercial systems
- SIMD - Single Instruction Multiple Data
- MIMD - Multiple Instruction Multiple Data: Scalable, fault tolerant, off-the-shelf micros

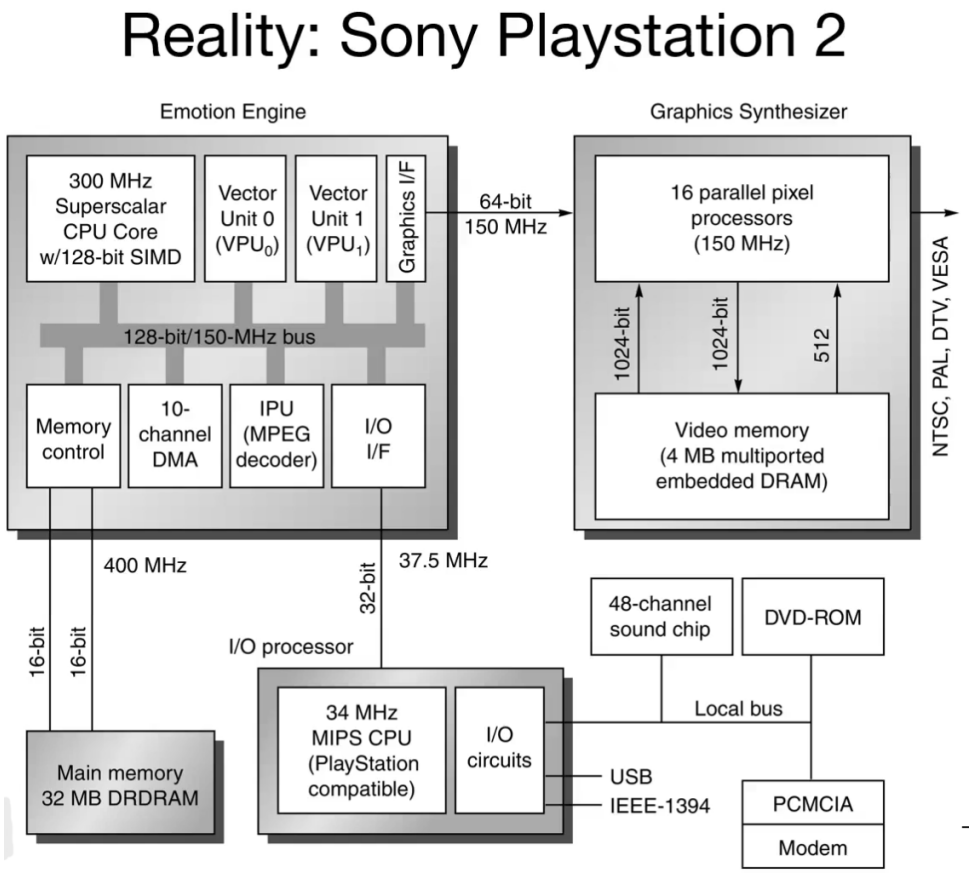
SISD architectures have hit this upper bound, and alternative architectures must be explored for increased performance: SIMD, MIMD, MISD.

10.2 SIMD

Vector computation (SIMD) requires each processing element to have its own memory space and necessitates vectorized architecture on multiple levels including registers, buses, and I/O ports. Vectorization can save time by summing up an entire loop in a single vector instruction, if iterations are independent.



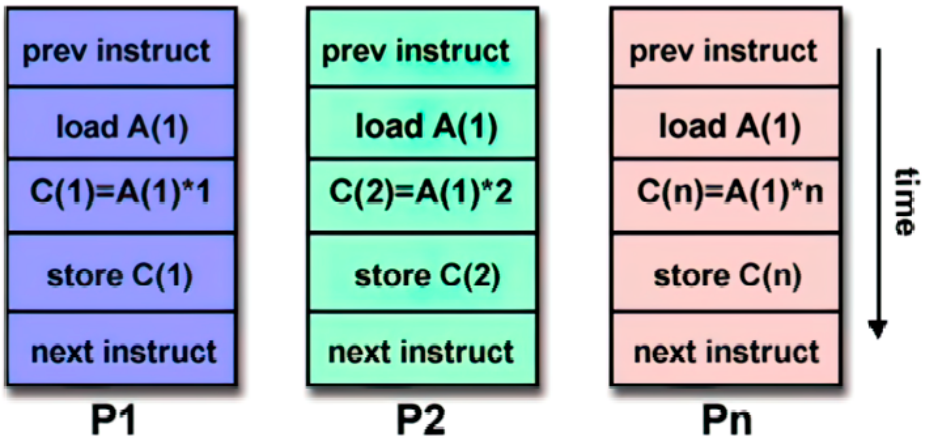
A notable example of complex vectorized architecture is Sony Playstation 2, which combines a graphics synthesizer similar to a modern GPU with a CPU that features a superscalar architecture with two vector units, a sort of SIMD extension to the cpu.



10.3 MIMD

Nowadays, the most common type of parallel computer are MIMD. MIMD architectures allow for multiple instructions to execute at the same time on different data, which can lead to super different computations on the same chip. MIMD processors can be arrays of processors that work asynchronously on different data, including

homogeneous and heterogeneous architectures, as well as clusters of independent machines. Highlights:



10.3.1 The console war between Xbox 360 and Playstation 3

The console war between Microsoft's Xbox 360 and Sony's Playstation 3 influenced the development of their respective architectures. Both systems used IBM's Powercore as the main processor, but the approaches differed:

- Sony used the **Cell** heterogeneous multicore processor, which contained a single 64-bit Power core and 8 specialized SIMD coprocessors. This enabled additional vector processing without changing the main CPU's ISA.
- Microsoft's **Xenon** homogenous multicore processor was composed of **three identical CPUs**, each containing its own SIMD extension, which modified the ISA.

Completely different approaches, one more oriented with a SIMD approach, the other more oriented with MIMD since three different CPUs.

> "If you program the software having in mind how the architecture is designed you will always gain better performance"

In a period of resurgence of applications /such as graphics, machine vision, speech recognition, machine learning and similar stuff, which requires large numerical computations that are often trivially data parallel SIMD-based architectures are most efficient way to execute these algorithms.

Nvidia ceo said:

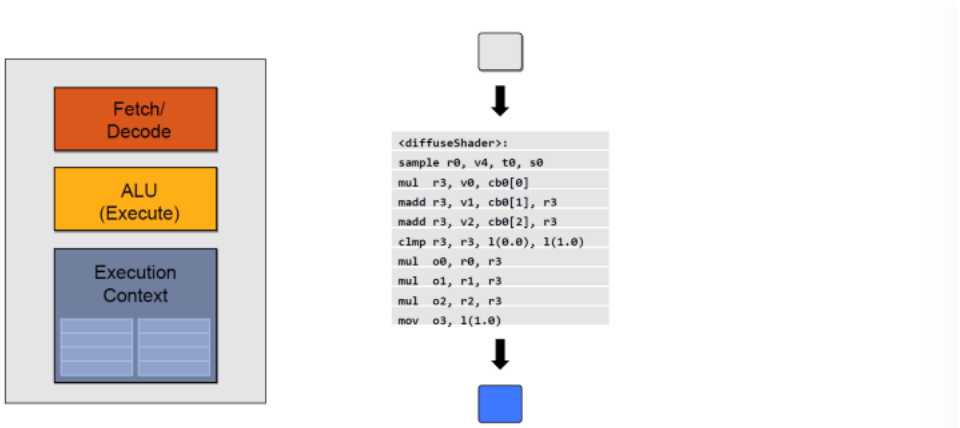
> "It's not just a matter of waiting for more transistor but designing the architecture, every part of the stack, from the hardware to the software to get more performance."

10.4 GPUs

GPU in computers:

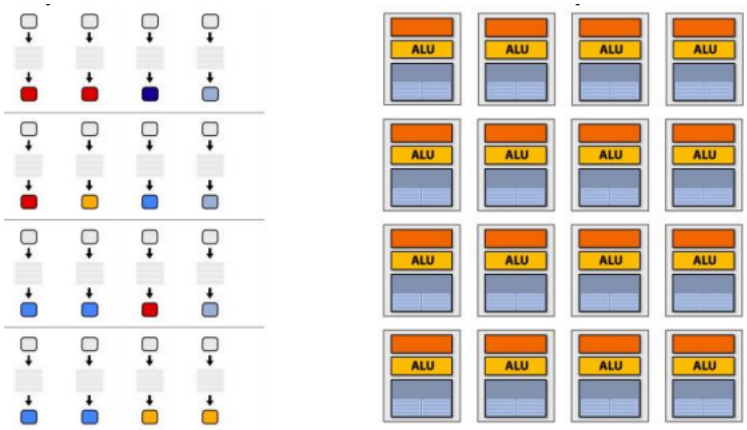
- In the 90s, high computational power in video games was needed, GPUs born
- GPUs also accelerate general-purpose computing functions with similar characteristics to graphics computing
- Extensive data parallelism
- Few control instructions
- Many math operations
- Few synchronization

A GPU simple core is very simple.

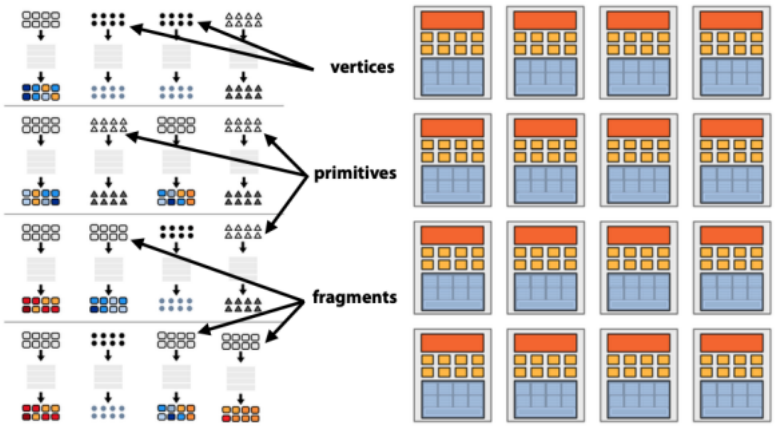


GPU is a throughput-oriented architecture. This means that the latency of the single stream is increased but the overall throughput is increased as well. As a result, the latency of the overall task execution is decreased. The interleaving of streams is managed by an HW scheduler and large register files are used to store execution contexts.

10.4.1 GPU pipeline



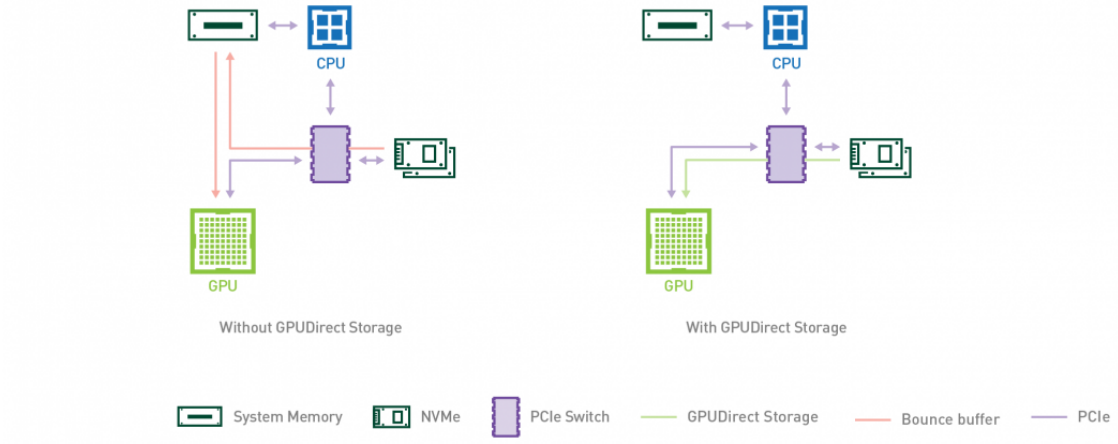
Computer graphics is parallel by nature and fragments, vertices and pixels are processed independently, meaning the function is executed for each fragment.



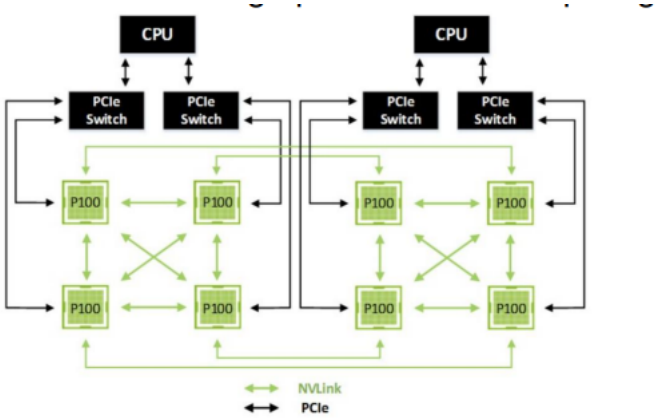
Nvidia CUDA architecture uses scalable array of multithreaded Streaming Multiprocessors (SMs). A CUDA program distributes kernel grid blocks to SMs which have execution capacity. Multiple thread blocks can execute on a SM concurrently.

The "warp" concept, introduced with the Fermi architecture, executes threads in groups of 32 threads. Similarly to Fermi architecture, Kepler's one incorporates register scoreboarding for long-latency operations and inter-warp scheduling decisions, similar to Fermi.

The CPU/GPU transmission bandwidth is a critical factor in this process. GPUDirect technology enables direct access to GPU memory from other devices, resulting in faster data transfer without involving the CPU's memory.



In a similar way, Nvidia developed a technology to increase data transfer speed between multi-GPU architectures used in high performance computing.



10.4.2 Tensor Cores

Half precision floating point instructions have been introduced, which are useful for **deep learning**. From NVIDIA Volta (2017) **Tensor Cores** were introduced, cores specifically designed for:

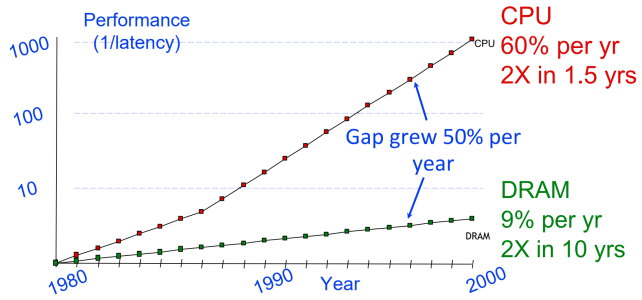
- Matrix multiplication and accumulation
- Very specific for DNN applications

In 2020, the Nvidia Ampere architecture introduced changes to Tensor Cores allowing for boosted tensor operations through **matrix sparsity** exploitation. Ampere introduces the concept of *fine-grained structured sparsity* which basically skips the computation of zero values.The primary alternative to *fine-grained sparsity* is through the organization of matrix entries/network weights in groups, such as vectors or blocks. This *coarse-grained sparsity* allows regular access pattern and locality, making the computation amenable for GPUs.

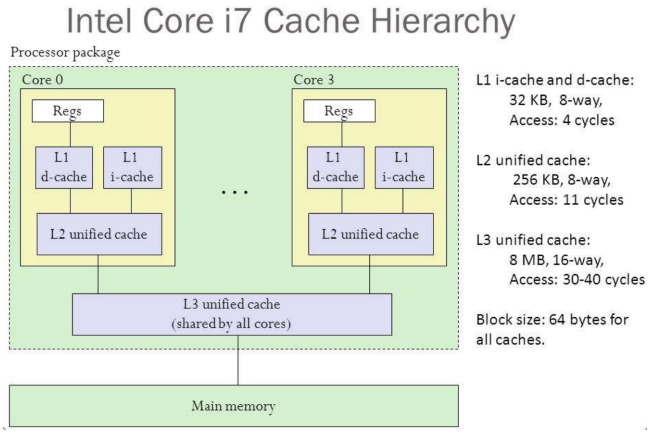
Nvidia Ampere architecture in 2020 introduced changes to Tensor Cores, enabling boosted tensor operations by exploiting **matrix sparsity**. *Fine-grained structured sparsity* skips the computation of zero values for efficiency. Another interesting introduction of Ampere was the **virtualization** of GPUs, particularly interesting for **cloud's service providers** where a single GPU can be partitioned in 7 separate virtual GPUs.

11 Memory Hierarchy

The origin of a lot of problems:



Developing a **memory hierarchy** is crucial in multi-core processors to improve memory access speed and create an illusion of a large, fast, and cheap memory.



11.1 Principle of Locality

The **Principle of Locality** states that programs access only a small portion of the address space at any given time. There are two predictable properties of memory references:

- **temporal locality** means that if a location is referenced, it is likely to be referenced again in the near future. This is observed in loops or reuse.
- **spatial locality** states that if a location is referenced it is likely that locations near it will be referenced in the near future. This is observed in straight line code or array access.

11.2 Cache

Block placement in cache can be addressed in different way:

- **Direct-Mapped Cache**: a block in the main memory can be placed only in one specific location in the cache. The block's address in the cache is determined by the index field of the memory address (in the next table is the tag).

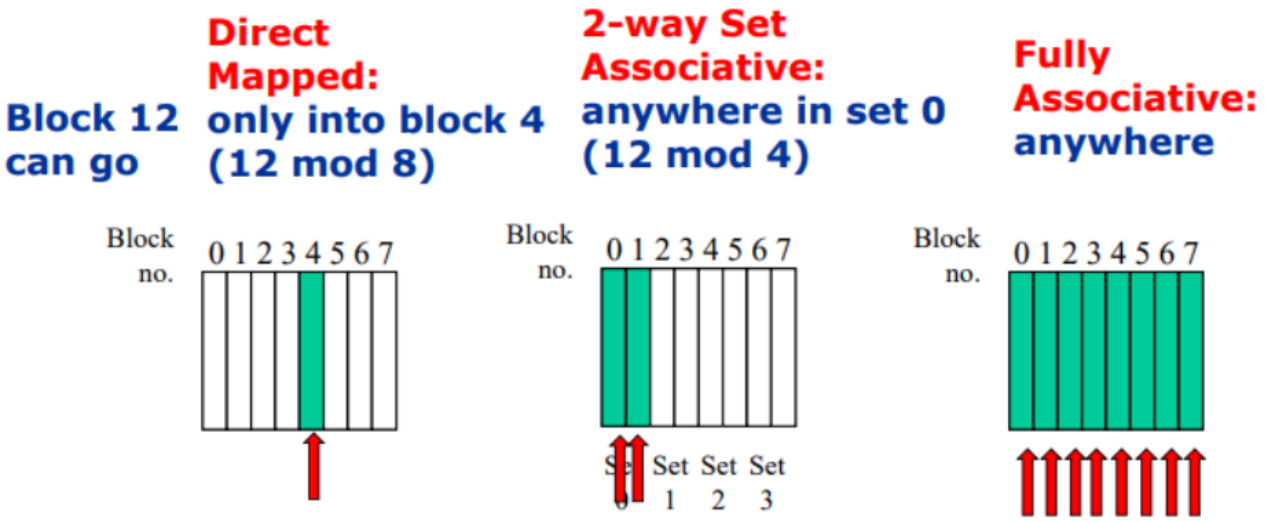
Block Address		Block Offset	
Tag	Index	Word OFFSET	Byte OFFSET

- **Fully Associative Cache**: a block in the main memory can be placed **anywhere** in the cache. The block's address in the cache is determined by a **tag** field that uniquely identifies the block.

Tag (n-4 bit)	Word offset (2 bit)	Block Offset (2 bit)

- **N-Way Set-Associative Cache**: it's a mix of the previous techniques. A block in the main memory can be placed anywhere inside a specific set. The block's address in the cache is determined by both the **set index** and the **tag**.

Block Address		Block Offset	
Tag	Index	Word OFFSET	Byte OFFSET



11.2.1 Different policies to manage cache

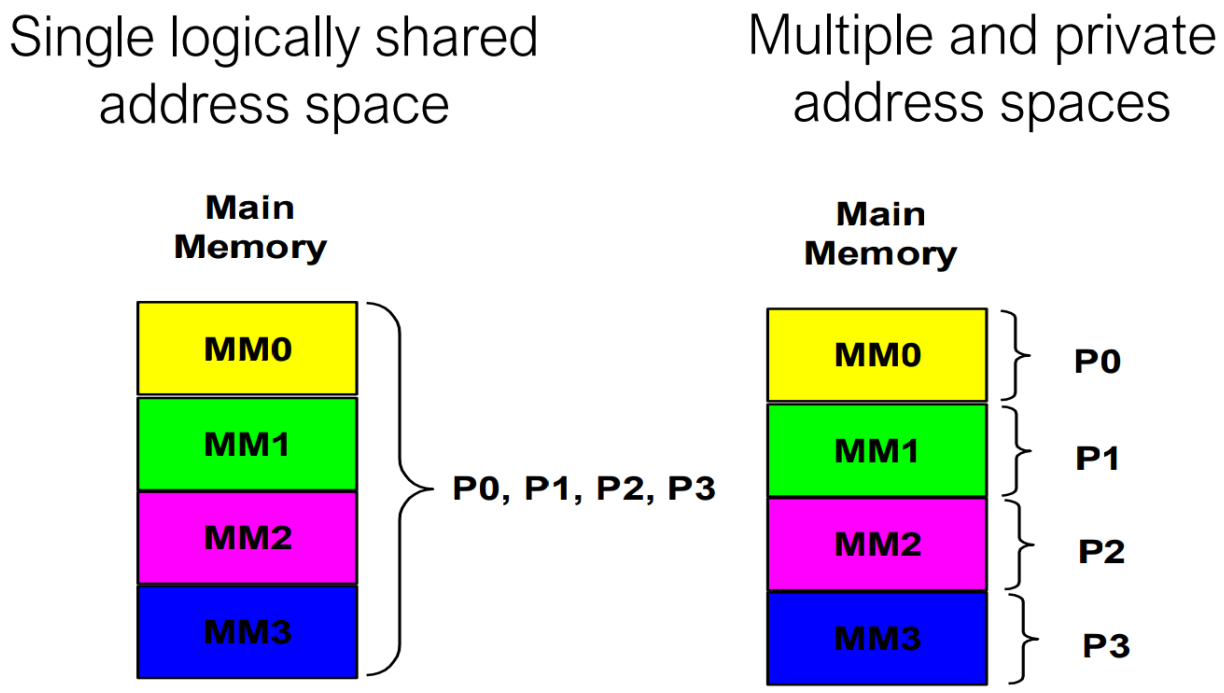
Common combinations are write-through with no write-allocate and write-back with write-allocate.

	Write-Through	Write-Back
Policy	Data written to cache block also written to lower- level memory	Write data only to the cache Update lower leve when a block falls out of the cache
Debug	Easy	Hard
Do read misses produce writes?	No	Yes
Do repeated writes make it to	Yes	No

11.3 Memory Address Space Model

Two types of logical memory architectures:

- **Single logically shared address space**: single logically shared address space. Any processor can reference any memory location. Multiple processors can refer to same physical address in memory.
  - Implicit communication (loads/stores)
  - Low overhead when cached
  - Complex to build in way that scales well
  - Requires synchronization operations
  - Hard to control data placement within caching system
- **Multiple logically private address spaces**: multiple and private address spaces. Processors communicate through send/receive primitives (**message passing**). Each processor's memory is disjoint and cannot be addressed by other processors.
  - Easier to control data placement (no automatic caching)
  - Overhead can be quite high
  - More complex to program
  - Introduces question of reception technique (interrupts/polling)



11.3.1 Physical Memory Organization

The concepts of addressing space (single/multiple) and the physical memory organization (**centralized/distributed**) are orthogonal to each other. Both the centralized memory and the distributed memory models can be used to implement a shared-memory architecture. Last phrase but in bold and repeated:

> **Shared memory does not mean that there is a single centralized memory.**

11.4 Cache Coherence

11.4.1 Bus-Based: Symmetric Shared Memory

The most popular memory architecture is **bus-based**: each core has its own cache memory and there is the bus, which connects processors with each other and to the shared memory. Since each core has its own cache memory, it's possible to encounter cache coherence problems between multiple memories. **Cache coherency protocols** ensure that all caches have the **same values** for the same address.

11.4.2 Cache Coherency Protocols in Multiprocessors

Two classes of protocols:

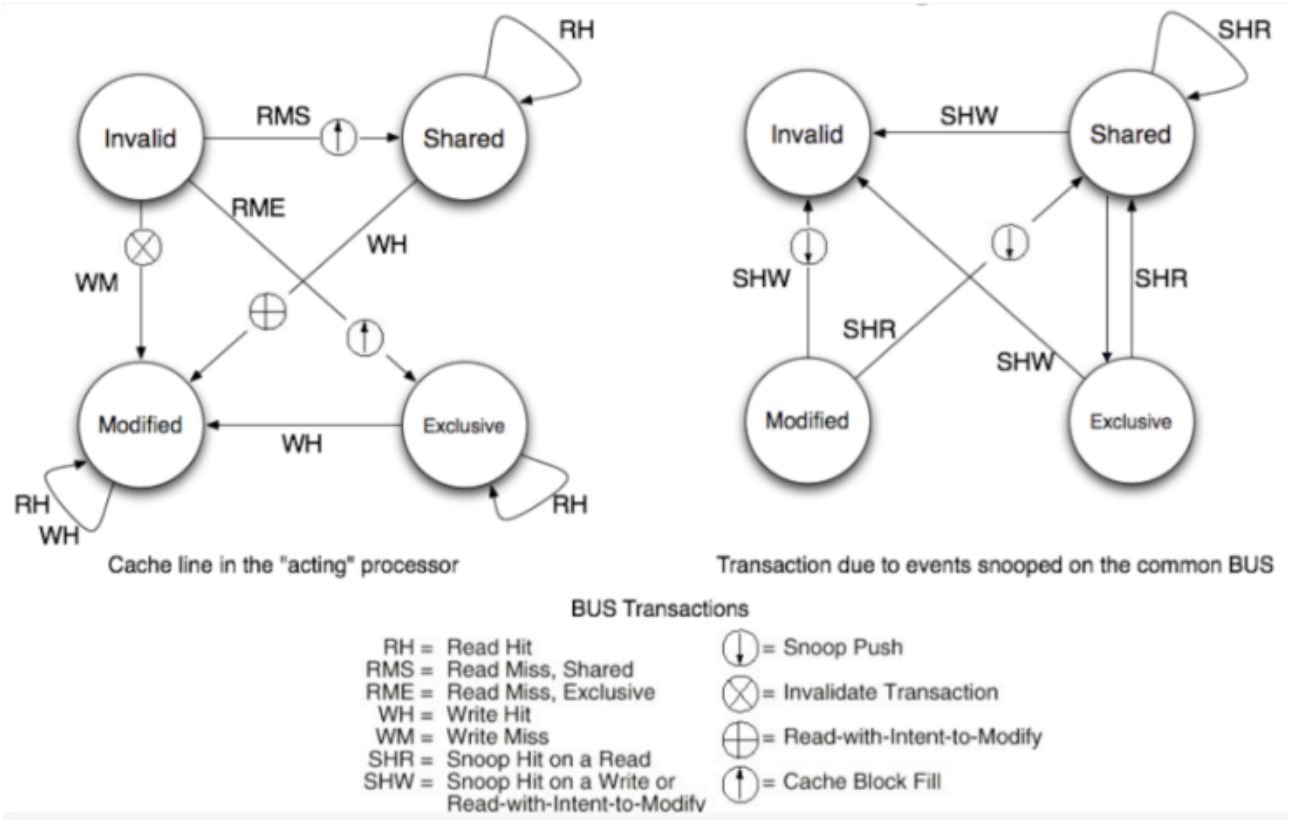
- **Snooping Protocols**: all cache controllers monitor (**snoop**) on the bus to determine whether or not they have a copy of the block requested on the bus and respond accordingly.
  - **Write-Invalidate Protocol**: The writing processor issues an invalidation signal over the bus to cause all copies in other caches to be invalidated before changing its local copy. This scheme allows multiple readers but only a single writer.
  - Write-Update or Write-Broadcast Protocol:
- Directory-Based Protocols

11.4.3 MESI

MESI protocol is a write-invalidate protocol which enhances
MSI Invalidate Protocol: it adds to the MSI protocol the exclusive state to indicate clean block in only one cache.

11.4.3.1 Cache blocks logical states:

- **Modified** : the block is dirty and cannot be shared; cache has only copy, its writeable.
- (NEW STATE) **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 Add exclusive state to distinguish exclusive (writable) and owned (written)



11.4.3.2 What to remember

- In both S and E, the memory has an up-to-date version of the data.
- Write to a S block implies the invalidation of the other copies of the block in caches and the memory will not be up to date.
- 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.

11.4.3.3 Simplified view

Golden rule:

"Keep the block has much as possible and don't update the main memory if not strictly necessary"

	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/yes) *	Maybe
A write on this line...	Access the BUS	Access the BUS	Access the BUS and Update the cache	Direct access to the BUS

* actually it depends on how the state machine is implemented. From the fifth and sixth edition of the reference course book conventions changed. If the protocol keeps track only the invalid signals it is not always implied that a Shared block has other copies in other caches. If the protocol keeps track also what the other operations are doing it's safe to say that shared implies always others copies.

11.4.3.4 Mesi example

After Operation	P1 block state	P2 block state	Memory at block0 up to date	Memory at block1 up to date
P1:read1	Exclusive1	Invalid0	Yes	Yes
P2:read0	Exclusive1	Exclusive0	Yes	Yes
P1:read0	Shared0	Shared0	Yes	Yes
P2:writeblock0	Invalid0	Modified0	No	Yes
P1:read1	Exclusive1	Modified0	No	Yes
P2:read1	Shared1	Shared1	Yes	Yes
P1:read1	Shared1	Shared1	Yes	Yes
P2:writeblock1	Invalid1	Modified1	Yes	No
P1:read0	Exclusive0	Modified1	Yes	No
P2:read1	Exclusive0	Modified1	Yes	No
P2:writeblock1	Exclusive0	Modified1	Yes	No
P1:read1	Shared1	Shared1	Yes	Yes
P2:read1	Shared1	Shared1	Yes	Yes