ELEC-H-473
Th03: Execution hazards: problems & solutions

Dragomir Milojevic
Université libre de Bruxelles

2020-2021

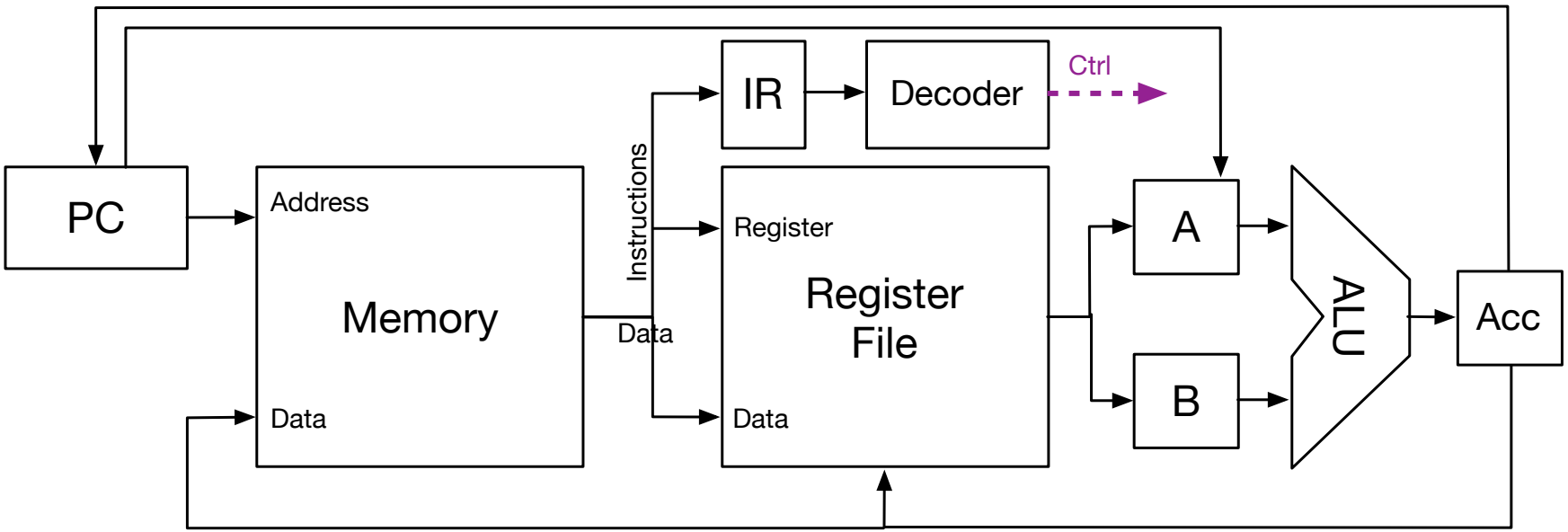
Plan for today

1. Pipelined, super-scalar architectures
2. Execution hazards – overview
3. Name dependencies
4. Data dependencies
5. Instruction dependencies
6. Control dependencies
7. Loop unrolling

1. Pipelined, super-scalar architectures

Previously

- Basic computer architecture is a **combination** of Von Neumann and Harvard architecture to maximize **data & instruction throughput** between the CPU and the memory
- We saw the simplified model of a complete computer architecture:



- Any (modern) CPU/computer will look like this (this is of course a helicopter view, we will be refining this gradually ...)

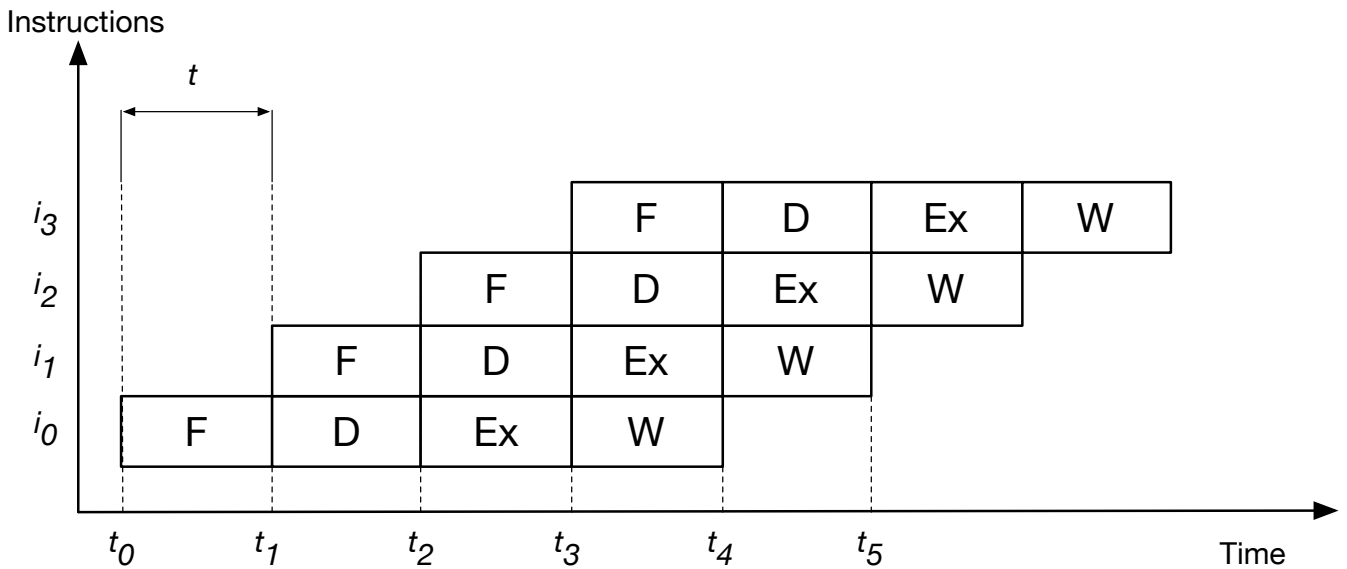
Main system architecture

- Instruction execution model introduces different views:
  - Programmer's view – F, D, Ex, W
  - IC designer's view – tied to logic circuit critical path $\rightarrow$ that will define maximum circuit operating F
- Pipelined execution is implemented to trade-off:
  - **delay reduction** (i.e. circuit speed) – frequency increase due to shorter critical path

    vs.
  - **latency** – pipeline execution will delay the output of the instruction for the number of cycles equal / proportional to pipe-line depth
- Pipeline depth is a HW design choice, with two extreme options:
  - CPUs with few pipeline stages generally associated to RISC
  - CPUs with many pipeline stages generally associated to CISC

IPC of a pipelined architecture

- IPC = Instruction(s) Per Cycle: for a single ALU and even if different stages are executed in parallel, at it's best, the pipeline can achieve IPC=1 (adding more ALUs could make IPC>1)
- That is 1 instruction that completes at **each system clock cycle**
- The above is possible only when pipeline is full and initialized (here from t_4 & on) and execution pace goes smoothly ...

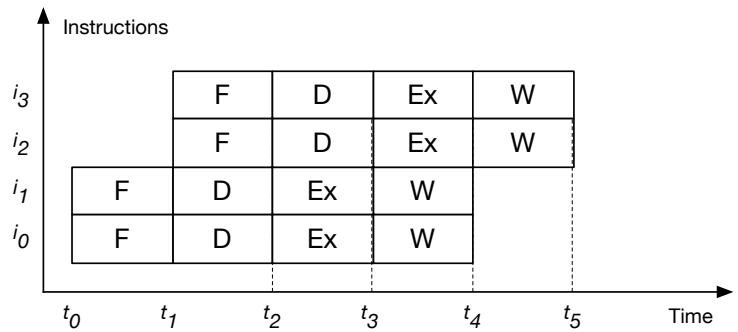


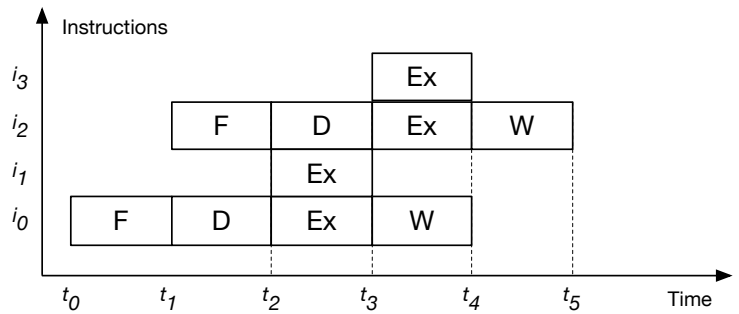
Super-scalar architectures

- In our simple architecture we have assumed **only one execution unit**; so only one instruction could be executed in one clock cycle
- ALUs are **not that expensive in HW** when compared to other components in the core/CPU (register file, all other stages, etc.)
- Why not using multiple ALUs to enable higher instruction throughput – super-scalar architectures
- If we have multiple ALUs **and the program has data independent instructions**, we could have few executions in the same clock cycle, and thus IPC>1
  - Note that not only computation is considered in super-scalar architectures, memory access too, so when IPC>1 we can have few computations and/or R/W operations
- Question is how to supply data to these ALUs and make them as busy as possible? (i.e. how to fetch and decode more instructions per single clock cycle?)

Instruction execution in super-scalar architectures

- One approach is to duplicate complete pipeline stage, i.e. we duplicate the whole pipeline (left figure)
  - ▷ This is what multi-core systems do ... This obviously will cost area & maybe it is not necessary to do it at single core level
- Another approach would be to use the existing F/D stage to fetch & decode more than one instruction at each clock cycle to feed the pipeline back-end (right figure below)





- Today CPUs use the combination of two: at core level one F/D stage generates multiple instructions for Ex stage; multiple cores are implemented in a single CPU

Acceleration of a super-scalar architecture

- For N instructions, n pipeline stages and m execution units, the total cycle count for a super-scalar architecture and acceleration with respect to a normal pipeline architecture is:

$$C_{ssc} = n + \frac{N-m}{m} \rightarrow Acc = \frac{C_{ssc}}{C_{pipe}}$$

- For $N \gg n, m$ (N is big compared to n and m):

$$Acc = \lim_{N\to\infty} \frac{C_{ssc}}{C_{pipe}} = \lim_{N\to\infty} \frac{n + \frac{N-m}{m}}{N+n-1} = \frac{mn+N-m}{m(N+n-1)} = \frac{1}{m}$$

- Acceleration of a super-scalar architecture is proportional to the number of execution units!
  - Compared to Single Issue Base Line the acceleration is: $n \times m$

2. Execution hazards – overview

Pipeline acceleration

- System speed-up proportional to the number n of pipeline stages is possible only iff:
  - all pipeline stages take the same, fixed, predictable amount of time – this is far from being easy to achieve in HW & SW for obvious reasons
  - there is no dependency between consecutive instructions – this could be done in SW (programmer of compiler), but also not that simple (notion of instruction scheduling that we will see later on)
  - we have lot of instructions – this is possible in heavy loops
- First two assumptions are EXTREMELY important !!!
- If this happens in the real world, then we are: either very lucky ... or we understand computer architecture so well, that we were able to write a perfect program !

In practice this is (VERY) hard to achieve, even for a very good programmer that knows HW and SW very well

In the real world

- What happens if we can not maintain ideal pipeline pace?

  - Do you remember C. Chaplin's movie "Modern Times"? (a must see BTW ...)
  - Our hero is too tired & distracted to follow other workers in the pipeline
  - He misses his turns, pipeline chain is broken ...

    and Mr. Ford doesn't become a millionaire !

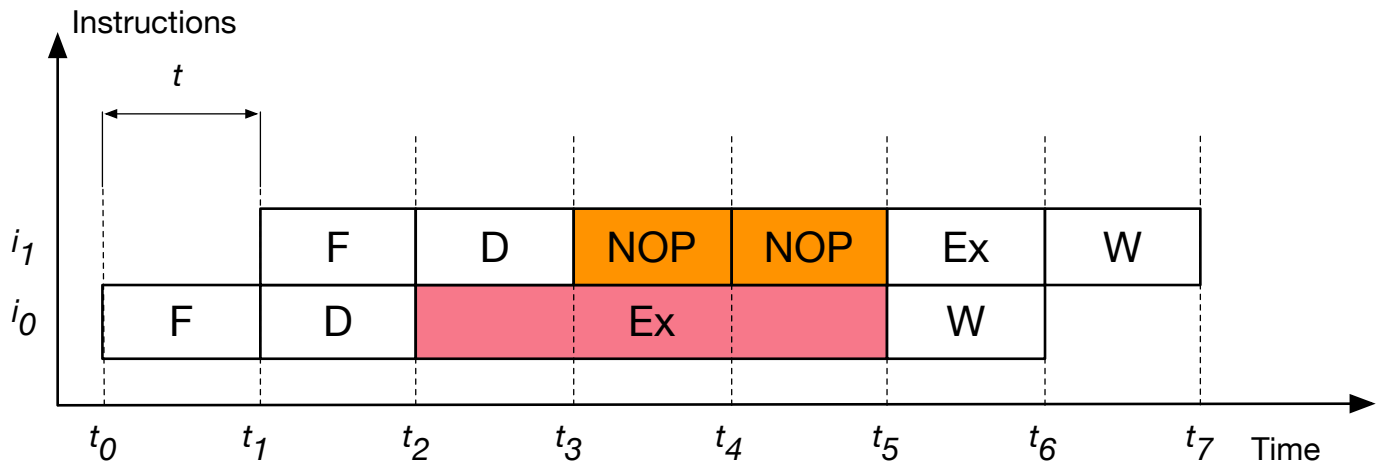
- In this occurs in a computer architecture we call this: pipeline stall, broken pipeline or bubbling (bubble "enters the pipeline")
- If this occurs often (think of loops and this is where computers are good), the SW performance could be seriously degraded, so high-performance programs will try to minimize these as much as they can

Bubbling in computer architecture

- Fundamental problem of bubbling is that in any practical computer architecture we will always have what is called execution hazards
- Hazards occur:
  - When instructions to be executed in the pipeline do not take the same amount of time to execute
    - Use of non-optimized arithmetic or any exotic operations ...
  - When something unpredictable occurs
    - You need to write to a IO device that is busy, your write can't complete
    - You need to fetch one operand from RF and the other from memory
  - When there is computational dependency between instructions
    - This is something that we will have by definition since computing with machines is about serialization of computation – ie. Turing machines
- We need to enable HW management of such situations, even if it is advised that as SW level we manage pipeline stall equally (difference between a good and poor program)

Pipeline stall – a simple way to handle hazards

- Instruction i_0 is fetched and decoded; already at that stage the control logic determines if there is a potential hazard for the next instruction i_1 (machines knows what it is going to do)
- If this is the case, the control logic will insert No OPeration – NOPs instructions (AKA as wait states) to slow down the execution of the next i_1 instruction; number of NOPs compensates difference in execution time, below the output of i_1 is delayed 2 cycles



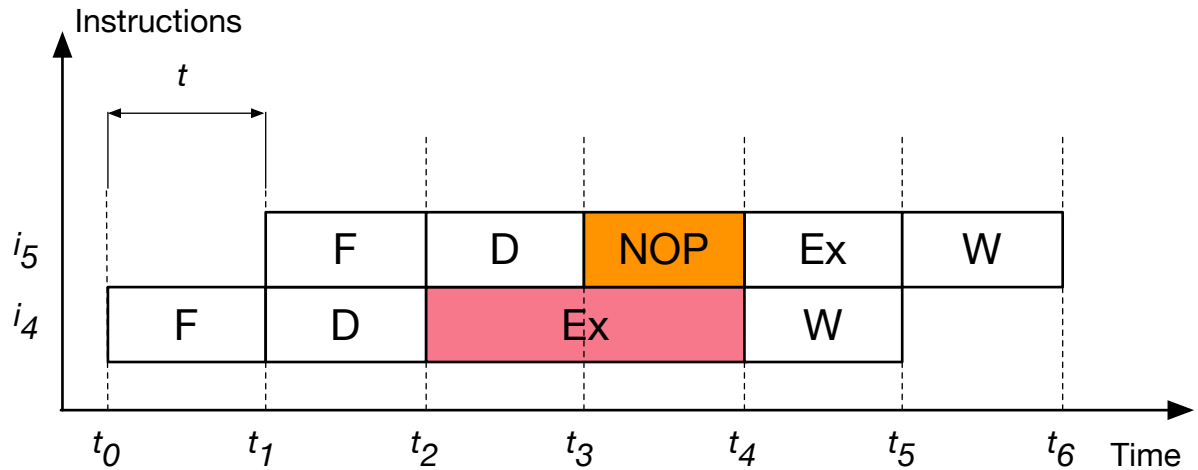
- If number of NOPs > 2, the previous stages will not be able to proceed either (bubble propagates to the input of the pipeline); next instruction executes only when the first fully completes

Bubble ex.: instructions with diff execution time

- Let's compute $D = (A/B) + C$ using:

```
1 mov  ax,a;
2 mov  bx,b;
3 mov  cx,c;
4 div  ax,bx; -- this one takes two cycles
5 add  ax,cx; -- this one will have to wait
```

- Suppose that `div` (`i4`) takes two cycles to execute and since it occurs before `add` (`i5`) that takes one cycle we insert one NOP
- Instruction `i5` waits for 1 cycle, no write in t_3, 1 cycle lost ... now imagine something more wild than 2 cycles more and how it could impact program performance



Impact of bubbling on acceleration

- For a single ALU, pipelined CPU, we could have IPC$=1$ **at most**, and this would mean one write at each clock cycle
- Because of the execution hazards we have pipeline stalls, and thus the real IPC is always going to be <1
- Obviously to improve the execution time we need to reduce pipeline stalls as much as possible
- Already at HW-level some things could be done and in order to understand few techniques (some of which are readily used in your favorite CPUs today)
- But let's first classify pipeline hazards:
  - Different types of dependencies
    - We speak of: Name, Data and/or Control dependencies &
  - Resource conflicts

3. Name dependencies

Name dependencies

- RegFile – a set of of registers that store operands; these are fast, multi-ported memories, expensive in Silicon, so we typically limit their number as much as possible to limit the system cost (& even system performance)
- If two or more instructions destinations targets the same register, but if there is no actual data flow between instructions we will have – register name dependency
- These dependencies are not **true (data) dependencies**, since there is no real computational sequence between instructions (reason why they are also called **false dependencies**)
- If this is the case, the register names can be changed, as long as they preserve the computation order – welcome to register renaming
- Who cares where the data is stored ... (computer of course, but not the programmer)

How to do register renaming?

- Different approaches depending on **when** exactly register allocation is going to be executed done
- Static register allocation – done **during programming or compile time**, not at run-time, so we have time to decide how to do it (good news, since more time better results we can get)
  - Programmer – when allocating registers for operands (when we write assembly code)
  - Compiler – when translating from high-level (C/C++) into assembly
  - Whoever does it, once registers are allocated they can not be changed; change is possible only if we re-compile (not handy)
- Dynamic register allocation – **at run-time**, during execution, live
  - Dedicated HW looks for register renaming opportunities
  - It dynamically adapts the renaming according to the needs (though we have much less time to decide ...)
  - This in general works quite well & is implemented in most CPUs these days

Dynamic register renaming – how it works?

- RegFile will contains the actual physical registers
- These registers are mapped to ISA, and they become logical registers (e.g. EAX, EBX, ECX etc.), i.e. as seen by the programmer
- In the HW there is a mapping of logical to physical register is done in 1-to-1 fashion, once for all (hardwired in HW) then for each ISA register there is **only one corresponding physical register**
- Or, we could do things differently:
  - Logical (ISA) registers are mapped to physical registers using some kind of dynamically allocated table called Register Alias Table – RAT
  - Program access physical registers of RegFile **through** the RAT
  - If the allocation (that is the mapping of logical to physical registers) is done smartly, register renaming could be done at run-time; can be fast since we need only a single read operation from RAT (a simple look-up, i.e. memory read)

Name dependency – example

Logical and physical registers are the same:

```
1 mov cx,[mem1]
2 add cx,bx
3 mov cx,[mem2]
4 add cx,dx
```

- Both additions target same destination register cx
- There is no possible overlap of instructions
- Poor possibility for pipelining (everything points to cx)

Logical and physical registers mapped through RAT, the actual execution of the code on the left could look like this:

```
1 mov cx,[mem1]
2 add cx,bx
3 mov jx,[mem2]
4 add jx,bx
```

- Second DST register modified to target another free slot (e.g jx)
- This is done live

Register allocation both compiler & HW

4. Data dependencies

Example of data dependency

- Compute: `eax` $\rightarrow$ `eax/ebx` and then `eax` $\rightarrow$ `eax+ecx`

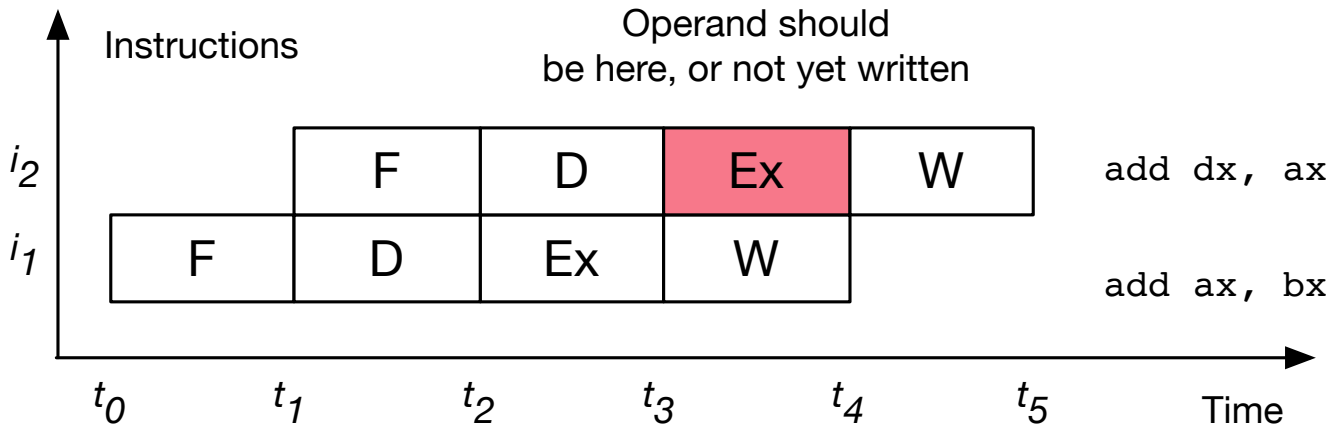
```
1 mov cx,a
2 mov bx,b
3 mov ax,c
4 div ax,bx ; this one is slow
5 add ax,cx ; this one is fast needs previous output
6 mov d,ax
```

- Suppose arithmetic division in line 4. is going to take more time then addition line 5. (as said this might not be true for high perf CPUs because they will have fast divider)
  - If these two instructions are to be pipelined: addition will have to wait until division ends first!
- It is possible to enumerate all possible types of data hazards:
  - Read-After-Write (RAW)
  - Write-After-Read (WAR)
  - Write-After-Write (WAW)

Different types of data hazards (1/2)

Read-After-Write (RAW) – (true data dependency) one instruction reads a location after an earlier instruction writes new data to it, but in the pipeline the write occurs after the read (so the instruction doing the read gets old data)

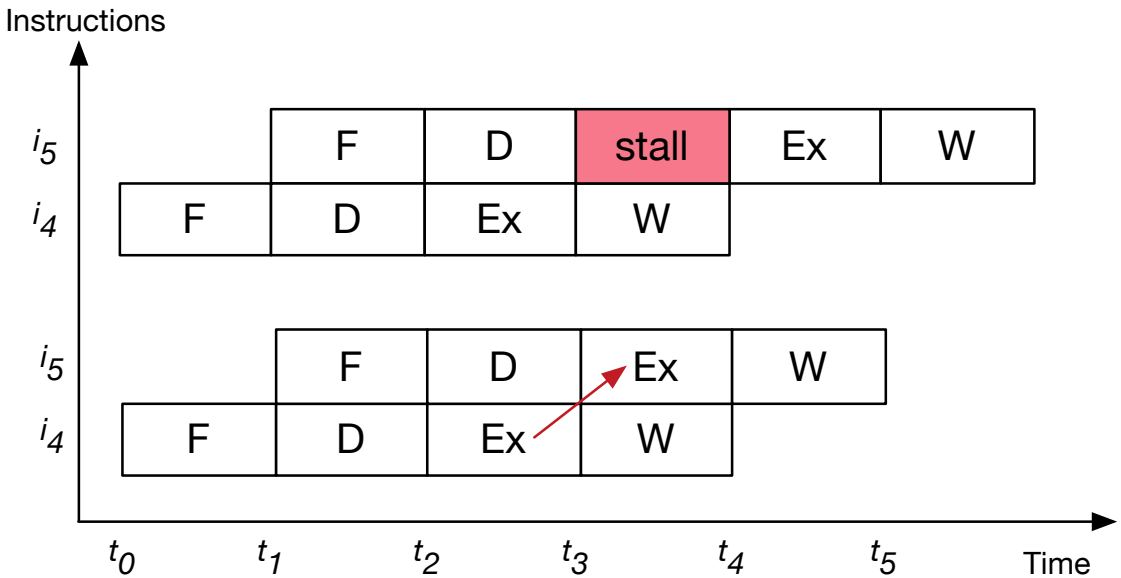
```
1 add     ax, cx;
2 add     bx, ax; ax needs to be written first
```



How CPU handles this?

Simple solution for Read-After-Write hazards

- In RAW case if i_5 uses result from i_4, it will be stalled even if i_4 takes only 1 cycle to execute
- This is because data needs to be written to RegFile before i_5 can use it (E stage of i_5 can proceed only after W of i_4, hence there is a **bubble**)
- Or computed data is already available after E of i_4: if this computed **data can be passed directly** to i_5 (bypass RegFile) we do not need to wait for W of $i_4 \rightarrow$ data forwarding



Different types of data hazards (2/2)

Write-After-Read (WAR) – (anti-dependence) if i_2 writes before i_1 uses it a write occurs after a read, but the pipeline causes write to happen first (this can happen if we have multiple ALUs and i_2 terminates earlier, good order needs to be preserved)

```
1 add ax, cx;
2 add cx, ax; if this is written
3             ; before previous instruction completed
```

Write-After-Write (WAW) – (output dependence) both instructions write to same destination; this can & will happen if we have multiple ALUs and i_2 terminates earlier

```
1 add ax, cx
2 add ax, bx
```

5. Instruction dependencies

Instruction scheduling 1/2

- Two instructions are said to be independent if they can execute simultaneously (i.e. in parallel)
- If we have multiple ALUs, then multiple independent instructions **could** be executed in the same clock cycle and IPC > 1 (Why do we say they "could"?)
- If instructions are independent they will not cause pipeline to stall
- From the program perspective this would mean that the best is to put independent instructions **as close as possible and as much as possible** together!
- Inversely, data dependent instructions should be placed just far enough so that the result is available when **exactly** needed
- In another words **there is a preferred execution order of instructions** that maximizes the instruction throughput, i.e. minimizes pipeline stalls
- Instructions scheduling is about finding this preferred order of instructions

Instruction scheduling 2/2

- Scheduling is a well known computer science problem, although most often studied on a completely different level of abstraction
- Usually we speak about task/thread scheduling in Operating Systems (OS)
- Many different scheduling techniques exist, the choice is made depending on what we want to do: please the user, or provide some guarantees on the execution time that could be hard (like in real-time systems)
- Difference between task and instruction scheduling is in the granularity & time available to compute the schedule (this is typically ms vs. ns time frame)
- Small amount of available time to compute schedule will have a strong impact on the way instructions are scheduled
- Can you tell what and why?

In-Order execution

- Proposed simplified architecture follows the execution sequence imposed by the program itself: this is called In-Order execution
- Instructions are executed in **exactly the same order** as they appear in the executable code; simple from HW perspective ...
- But if the executable code is poor, hazards will occur
- If we use assembly language than the programer is responsible for the instruction order
- If we use high-level programming language, this can be automated and the code quality will depend on the compiler quality
- Important: for in-order architectures, **correct scheduling is going to be ISA dependent**, one executable may execute differently on another ISA (or may not execute at all)
- Consequence – performance is architecture dependent

Out-of-Order executions vs. In-Order execution

- Another approach: make a **dedicated HW unit** that will perform Out-of-Order (OoO) execution (so, at run-time)
- Goal of OoO engine is to examine the code before the execution and **pack** instructions together to minimize potential pipeline stalls (i.e. maximize instruction throughput)
- Because it is a dedicated HW block OoO execution can adapt **dynamically** to the code that executes
  - Advantage: scheduling is compiler/user independent
  - Disadvantage: extra area (significant!) & minimal decision time, or scheduling problem is a very complex SW problem, so don't expect miracles (remember: Garbage-In, Garbage-Out)
- Even if OoO is fully automated, performance chasers will check eventual penalties (especially in loops)
- Most of the CPUs today are OoO, only small CPUs or μcontrollers remain in-order

Out-of-order (OoO) execution example

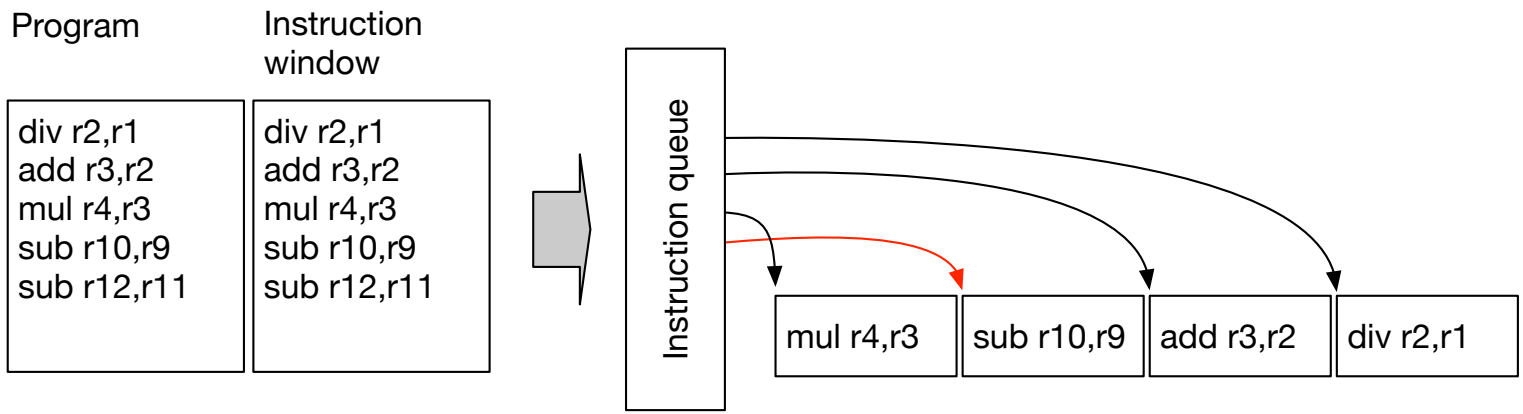
- First 3 instructions are data dependent, so no scheduling opportunities for `div`, `add`, `mul`; `div` takes 3 cycles to execute
- Last two `sub` operations are independent
- One of the `sub` could be executed earlier to hide the latency of `mul`

```
1 div r2,r1;
2 add r3,r2;    -- needs r2
3 mul r4,r3;    -- needs r3 --> serial computation
4 sub r8,r7;
5 sub r10,r9;
6 sub r12,r11; -- all these are independent
```

Cycle	1	2	3	4	5	6	7	8
Div R2, R1	Fetch	Decode	Execute			Write		
Add R3, R2		Fetch	Decode	Wait		Execute	Write	
Sub R8, R7			Fetch	Decode	Execute	Write		
Mul R4, R3				Fetch	Decode	Wait	Execute	Write
Sub R10, R9					Fetch	Decode	Execute	Write
Sub R12, R11						Fetch	Decode	Execute

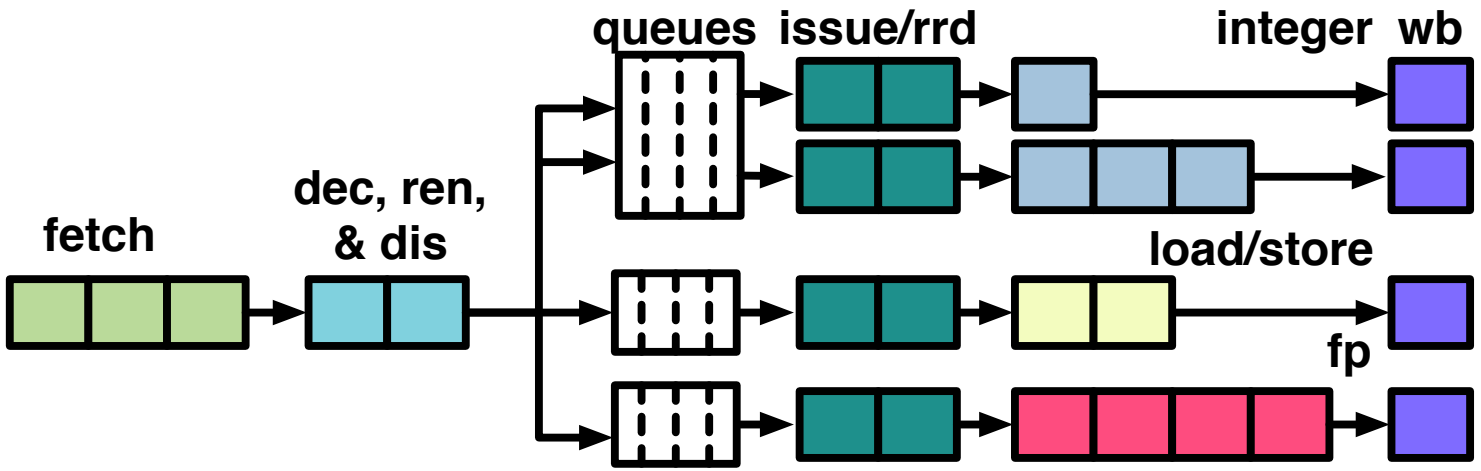
OoO: How it is done in practice?

- Instruction are fetched and dispatched into an instruction queue (instruction window) where they wait until their input operands become available
- Instructions are allowed to leave the queue before others; they are issued whenever an appropriate ALU is free
- Results are queued (in some memory), and only after all older instructions have their results written back to the register file, the end result will be retired from the output queue



OoO: Concrete example

- University of Berkeley Out Of Order Machine – open source, RISC V spec CPU that is configurable (you can choose what is inside), supports multiple ALUs, it is silicon proven with performance comparable to competitors
- Described using high-level language descriptions (Chisel)
- Generates actual HDL for circuit synthesis, place & route



6. Control dependencies

Branches

- Branches are essential in any software, because they provide mechanisms that allow to alter "normal" instruction execution flow
- At **branching point**, the instruction control flow can chose from 2 different instruction sequences that are mutually exclusive (one or the other, but never both since condition is a Boolean)
- Branches can be:
  - ▷ Unconditional – equivalent to `goto` statements
  - ▷ Conditional – Boolean evaluation of an algebraic/logical expression

What does the following code do:

```
1 mov bx,b;
2 label:
3     ...; -- doing something
4     sub bx,1;
5     jnz label;
```

Unconditional jumps are bad!

- **Uncontrolled** usage of unconditional jumps creates unreadable code, that is difficult to debug and possibly leading to inefficiencies during execution
- And yes there is no way to "control" jumps anyway ...
- Their usage lead towards what in literature is known as "spaghetti code" – this is not a joke, analogy is perfect, you can not figure out which part of the spaghetti goes where !

So what we should do?

- The right alternative is structured programming:

  A programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of subroutines, block structures, for and while loops – in contrast to using simple tests and jumps such as the go to statement which could lead to "spaghetti code" causing difficulty to both follow and maintain.

- Use functions & subroutines and NEVER JUMPS!
- Known for some time now, look what E.W.Dijkstra wrote in '68:

> Since a number of years I am familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. Later I discovered why the use of the go to statement has such disastrous effects and did I become convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except -perhaps- plain machine code).

Go-to statement considered harmful in Commun. ACM 11 (1968), 3: 147-148

Conditional jumps in C like

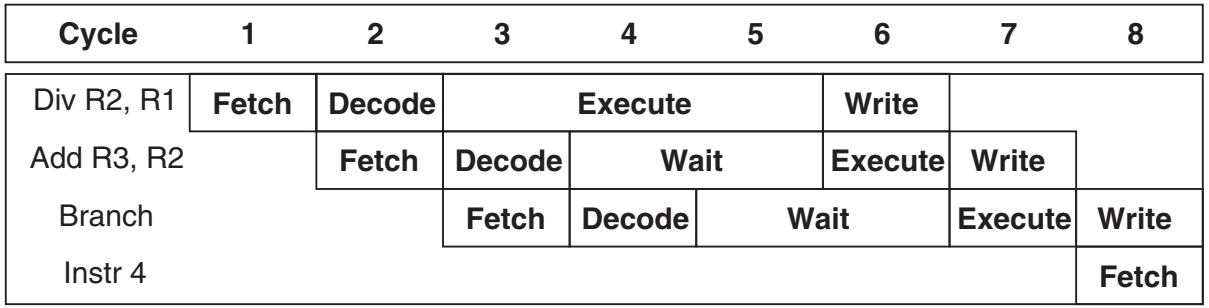
- Typical branching structure:

```
1 if (a==35) then {
2     // instruction sequence 1 NOT TAKEN
3  } else {
4     // instruction sequence 2 BRANCH TAKEN
5 }
```

- Depending on the Boolean value (true/false) of the condition (a==35), the branch could be:
  - Not taken (no branching) – we execute the instruction sequence 1; the next instruction is stored just right after (in the program memory) so **we do not have to change the Program Counter**
  - Taken – we execute instruction sequence 2; the next instruction is stored somewhere further in memory, we have to change the Program Counter!
- To take, or not (a branch) will have impact on system performance, so be careful with your test ... let's see

Branching and pipeline execution

- Branching has a very strong impact on pipeline execution since PC will change to something that is not simply incremental (next instruction), and hence can not be easily anticipated
- We first compute `if` test (using our ALU) and then look into the result of this, and only then we can decide where to go
- Next instruction can not be fetched before we know which of the two paths we will have to follow !



Example: here i_4 has to wait until cycle 8 to be fetched; caused by cumulated delay due to divide & **branch instructions**

Condition statistics

- For any `if` statement we could try to figure out: how often the condition is TRUE (or FALSE)?
- In the previous example condition a==35 could be: often TRUE; often FALSE; or it could be 50/50 ... or whatever statistics
- Knowing the statistics we could write the condition so that during execution branch **is not taken** most of the time!

```
1  // a is often different from 35
2  if (a!=35) then {
3      // instruction sequence 1
4  } else {
5     // instruction sequence 2
6  }
7  // a is often equal to 47
8  if (b==47) then {
9      // instruction sequence 3
10 } else {
11    // instruction sequence 4
12 }
```

What are the statistics for values a or b here?

How to deal with branching?

- Branches are essential part of any SW & they have strong impact on the pipeline performance: we need to handle them smartly
- In order to improve pipeline execution of branches all processors today include some kind of branch prediction
- What are the options:
  1. No prediction at all – keep things as they are ...
  2. Static prediction – decide at CPU design-time
  3. Dynamic prediction – decide at run-time, adaptable
- In the last one we **speculatively** execute a branch, even if it is not the right one ...
- Typically two things should be predicted:
  - If the branch will be taken or not (to decide if the PC needs to be changed)
  - Branching address where to go (how to change the PC)

a) No prediction at all

- We first compute Boolean condition (we have only one ALU remember) and then we look into the result of this computation
- Next instruction can not be fetched before we know which of the two paths we will have to follow!
- Simplest scheme to handle branches is to flush the pipeline, i.e. hold or delete any instructions after the branch, until the branch destination is known
  - Flushing pipeline is in general a bad idea; Can you quantify what exactly are we loosing?
- This is simple to make in hardware and to use in software
- But as you would expect this is not efficient, especially when the programer wrote a wrong test ... in a loop that will execute many, many times (so many, many pipeline flushes & lost CPU cycles)

b) Static branch prediction 1/2

- Static prediction — during CPU design time we decide which is the preferred branch will be **always** chosen by the HW (choice between taken or not take)
- When `if` is encountered, the pipeline **continues** the execution, as if there was no branch in the code
- Pipeline doesn't sees the branch, the condition is calculated as any other instruction in the program
- Even if the result of the branch condition is known later, the next instruction can be pushed into the pipe
- When the result of the test becomes available we are either:
  - OK – program took the branch that it should, pipeline is not broken (SW & HW are in phase)
  - FAIL – we took the wrong way! (and we loose cycles)

b) Static branch prediction 2/2

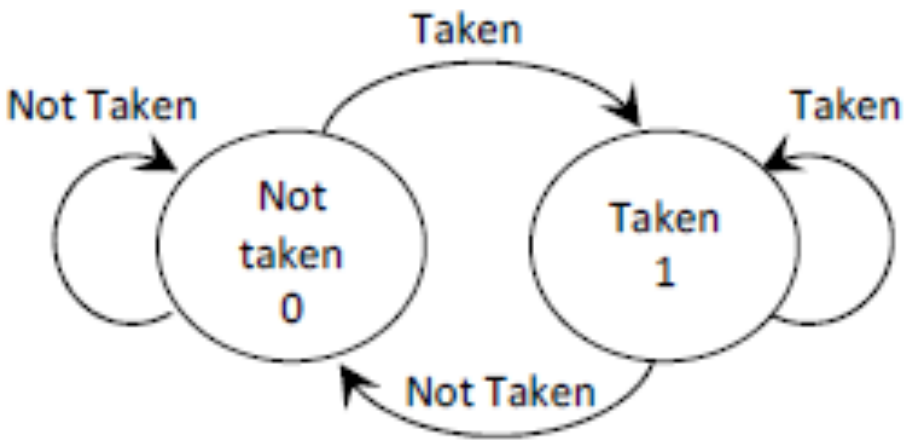
- If HW & SW are in phase, the result of the test can be discarded; it is like there was no branch in the SW
- If not, we need to **flush the pipeline**:
  - Turn already fetched instruction into NOP (no operation)
  - Re-calculate the new address for the Program Counter (PC)
  - Restart the instruction fetch of a instruction indicated by the new value of PC
- This is bad since it will cause a loss of many CPU cycles, the overall SW performance goes down
- If the branching strategy of your CPU is known, you can adapt your code accordingly when you write your `if`
- If not, you have 50% chances to guess it right (or not), and we do not like guessing in exact sciences, don't we?

c) Dynamic branch prediction (1/5)

- In static prediction, all decisions are made **before** the actual program execution (at CPU design time)
- This means that the prediction scheme **can not adapt** itself to the **program behavior** that could change over time or **different input data** (data sets could vary, e.g. a processed image could be mostly white or black)
- To allow dynamic adaptation to the SW most of the CPUs introduce dynamic branch prediction – use the past branch behavior to predict the future
- We use HW to dynamically predict the outcome of the test; prediction will depend on the behavior of the branch at run-time
- Prediction can be updated depending on the branching *history*, using more or less complex algorithm

c) Dynamic branch prediction (2/5)

- 1-bit branch prediction scheme uses only 2 states
- Branch prediction will say "not taken" if the previous branch has not been taken (it will say "taken" if the previous branch has been taken)
- In another words the prediction follows what happened last; the state change will occur after one miss-prediction



c) Dynamic branch prediction (3/5)

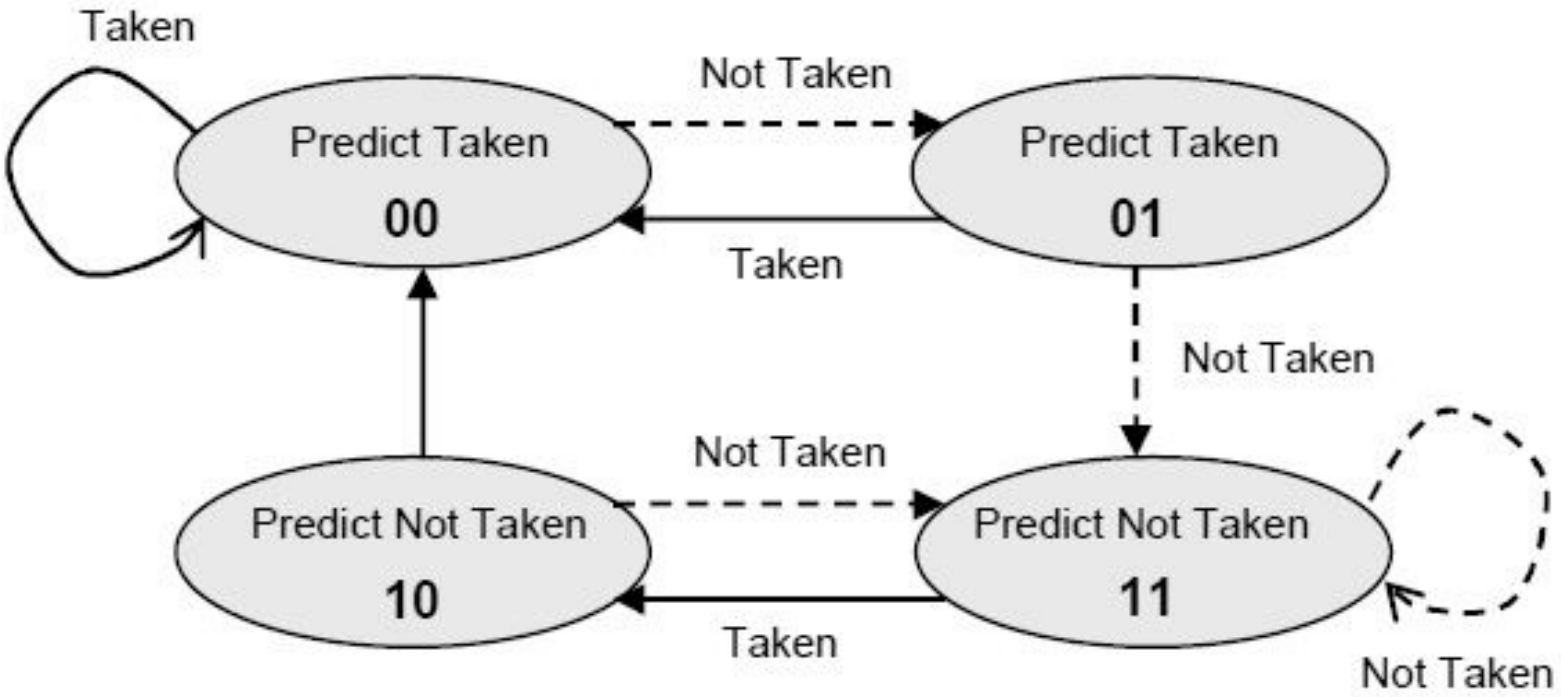
- Loops – they need a branch to decide if they should enter or not the loop body; taken (T) enters the body, Not Taken (NT) exits the loop and assume that when branch is executed for the first time we predict that it is not taken (NT)
- If the loop has at least one iteration, the NT prediction will be wrong and will turn to T until the last element in the loop where we will make a misprediction again
- If the loop is huge, two mispredictions are not a big deal
- Now let's analyze 1-bit branch prediction in a nested loop:

```
1  for (i=0; i<1000000; i++){
2    for (j=0; j<10;j++) {
3    // some computation
4    }
5  }
```

- Due to loop nesting and loop indexes (many outer, less inner loops) this could cause serious performance penalty due to many mispredictions

c) Dynamic branch prediction (4/5)

- 2-bit branch prediction scheme (4 states): you have to be wrong twice in a row before the prediction is changed



c) Dynamic branch prediction – downside (5/5)

- Spectre vulnerability – exploit branch prediction to perform side-attack!

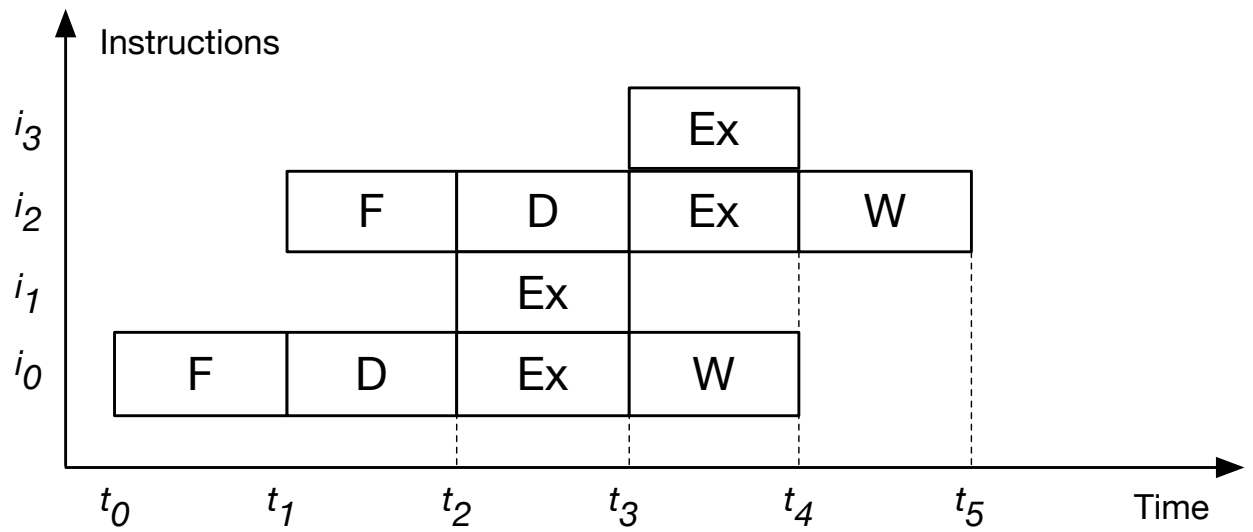
***Abstract*—Modern processors use branch prediction and speculative execution to maximize performance. For example, if the destination of a branch depends on a memory value that is in the process of being read, CPUs will try to guess the destination and attempt to execute ahead. When the memory value finally arrives, the CPU either discards or commits the speculative computation. Speculative logic is unfaithful in how it executes, can access the victim's memory and registers, and can perform operations with measurable side effects.**

Spectre attacks involve inducing a victim to speculatively perform operations that would not occur during correct program execution and which leak the victim's confidential information via a side channel to the adversary. This paper describes practical attacks that combine methodology from side channel attacks, fault attacks, and return-oriented programming that can read arbitrary memory from the victim's process. More broadly, the paper shows that speculative execution implementations violate the security assumptions underpinning numerous software security mechanisms, including operating system process separation, containerization, just-in-time (JIT) compilation, and countermeasures to cache timing and side-channel attacks. These attacks represent a serious threat to actual systems since vulnerable speculative execution capabilities are found in microprocessors from Intel, AMD, and ARM that are used in billions of devices.

Paul Kocher et al., "Spectre Attacks: Exploiting Speculative Execution"

Resource conflicts 1/2

- Super-scalar processor fetch and decode **few** instructions at a time to build the instruction queue (remember that in the OoO the instructions pulled out of the queue do not need to be in order)
- Different instructions are **then dispatched** to different ALUs
- Different ALUs in a super-scalar CPU do not have to be the same – often they have a common subset of functions + some specialized features (i.e. they are heterogeneous)



Resource conflicts 2/2

- Let's imagine that on top of normal functionality we have a dedicated HW multiplier in ALU1 and not in ALU2
- If two instructions are to be executed are `add` and `mul`, we are OK, both ALUs can be used at the same time (parallel processing) and we could have IPC$=2$
- If two instructions are `add`, OK too (we still have IPC$=2$)
- Problem arises if both instructions are `mul`
- If issued at the same time they will generate a resource conflict – two or more instructions try to use the same HW resource – one instruction will have to wait ...
- This will causes pipeline stall and we loose clock cycles as before
- In another words **functional specificity** of ALUs will generate extra constraints for instruction scheduler that will become more complex to extract ILP to increase IPC (no free lunch)

7. Loop unrolling

Why loop unrolling?

- Let's look into the following loop:

```
1 // heavy loop starts here
2 for(i=0;i<1000000;i++) {
3   A[i]=A[i]+B[i];
4   }
```

- If for some reason there is a **delay on** `A[i]=A[i]+B[i]` computation (data not ready, long operation, etc.), there will be a **pipeline stall** on every loop iteration!
- Bigger the loop, more time (& power) we loose for nothing (here the loop is $\times$1.000.000)
- At the end of each iteration we need to check (i.e. calculate) the condition `i<1000000`
- This is really useful only 1 out of 1.000.000 times ! (for 999.999 times the test is TRUE)

Loop unrolling: how and what does it do

- Consider the following loop transformation:

```
1 for(i=0;i<250000;i++) {
2   A[i+0*250.000]=A[i+0*250.000]+B[i+0*250.000];
3   A[i+1*250.000]=A[i+1*250.000]+B[i+1*250.000];
4   A[i+2*250.000]=A[i+2*250.000]+B[i+2*250.000];
5   A[i+3*250.000]=A[i+3*250.000]+B[i+3*250.000];
6 }
```

- Goal here is to increase the probability of possible overlapping instructions to help instructions scheduler to avoid pipeline stalls
- Four computations are now **independent**, so it is possible:
  - to do register renaming
  - to order instructions to minimize the pipeline stall
  - if the CPU is super-scalar, the system could use multiple execution units at the same time
  - we compute less conditions
  - we increment less