DEPARTMENT OF
NETWORKED SYSTEMS
AND SERVICES

COMPUTER ARCHITECTURES

Wide Instruction Pipelines

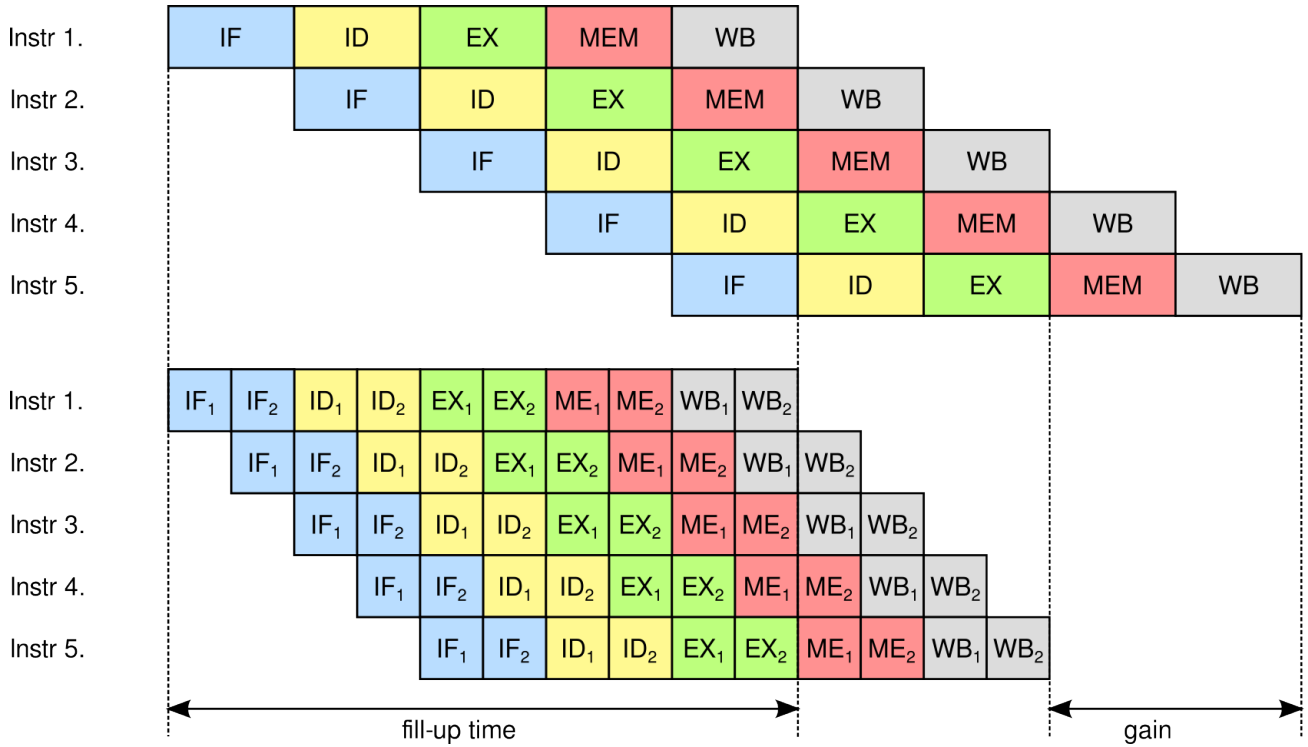
Gábor Horváth, ghorvath@hit.bme.hu

Budapest,
2023. 05. 16.

HOW TO MAKE THE CPU FASTER

Option 1: Make the pipeline *deeper*

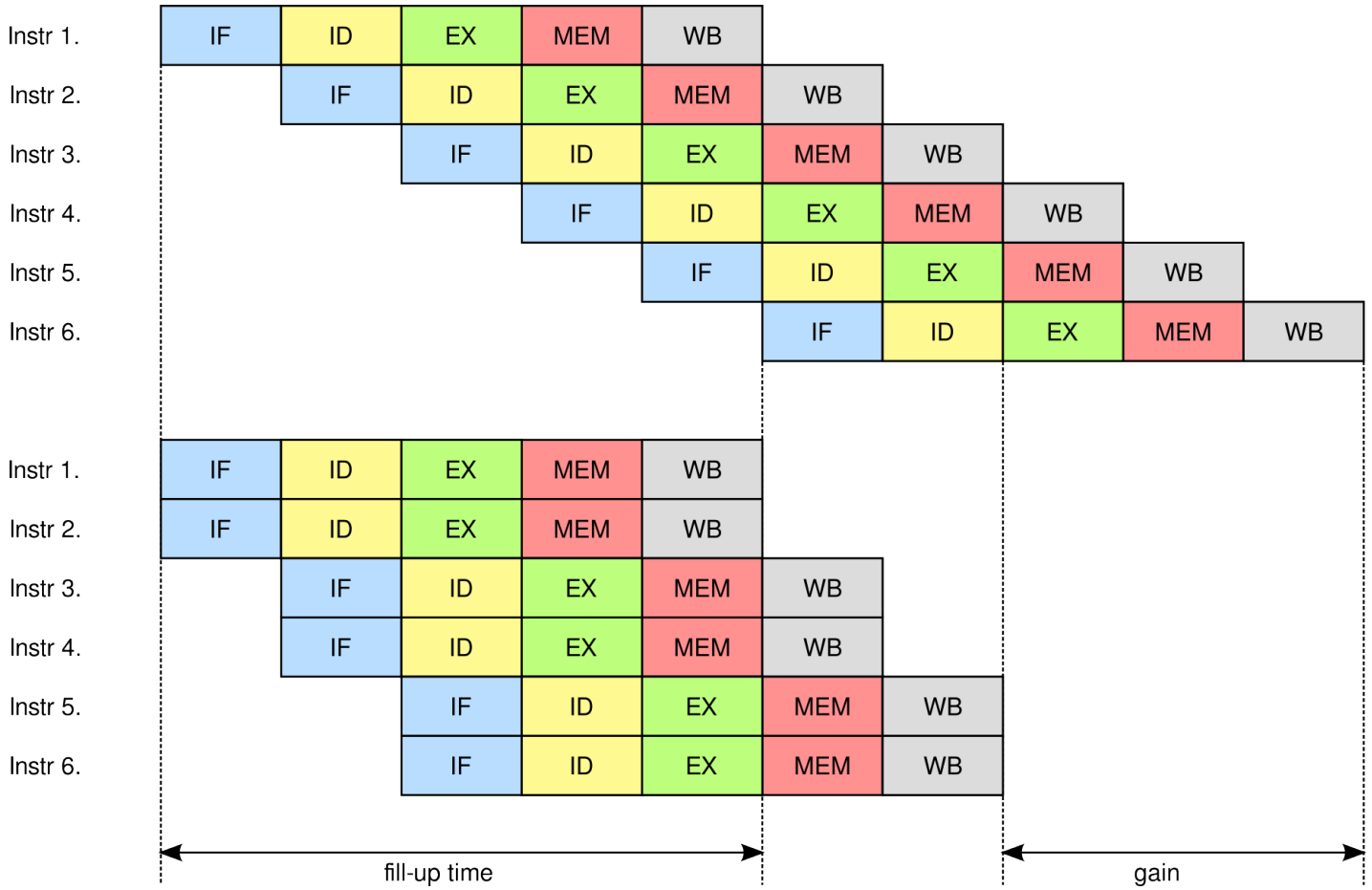
- Let us increase the clock frequency by a factor of 2
- To do so each stage needs to be cut to two sub-phases
- The throughput of the pipeline increased by a factor of 2!



HOW TO MAKE THE CPU FASTER

Option 2: Make the pipeline *wider*

- Each stage can work on several instructions at the same time



EVALUATION

- Depth: k times, width: m times
  → leads to a speedup of factor m*k in theory
- Practical issues:
  - If m*k is large: many instructions are being processed at the same time → there are lots of data hazards → pipeline stalls are necessary to resolve them → the efficiency drops
  - If the pipeline is too wide → too many forwarding paths' → complexity, power consumption, price, etc.
  - If the pipeline is too deep: the penalty of the speculative decisions will grow
  - The clock frequency can not be arbitrary large
    - The read/write latency of the pipeline registers is a physical limit
- Typical values:
  - Depth: 5-30
  - Width: 1-6

EXAMPLES

	Depth	Width
Pentium	5	1
Pentium Pro	14	1-3 (1 complex + 2 simple)
Pentium 4 Prescott	31	3
Intel Core	14	4
Intel Core i7 Nehalem	16	4
Intel Core i7 Kaby Lake	14-19	4
Intel Atom	16	2
Alpha 21264	7	4
ARM Cortex A55	8-10	2
ARM Cortex A75	11-12	3
APPLE A10	?	6
POWER9	12-16	6

MAKING THE PIPELINE WIDER

- Key of the efficiency:
  - The program must contain enough number of independent instructions

- Who collects independent instructions?
  - The compiler:
    - **Static scheduling**
    - VLIW/EPIC architectures
  - The CPU itself, real-time:
    - **Dynamic scheduling**
    - Superscalar processors
      - In-order: the instructions are executed in the order given by the program
      - Out-of-order

INSTRUCTION SCHEDULING

- There are several possibilities between the purely static and purely dynamic scheduling

	Collect instructions to execute in parallel	Assigning instructions to functional units	Determining when an instruction is executed
Superscalar	Hardware	Hardware	Hardware
EPIC	Compiler	Hardware	Hardware
Dynamic VLIW	Compiler	Compiler	Hardware
VLIW	Compiler	Compiler	Compiler

Wide pipelines with dynamic scheduling

SUPERSCALAR PROCESSORS

- The processor can fetch several instructions in a cycle
- The processor is able to...
  - ...collect instructions for simultaneous execution
  - ...assign instructions to execution units
  - ...perform dependency analysis (and schedule instructions)
- If the execution of *m* instructions can start in the same cycle:

  → ***m-way superscalar processor***

- Two solutions:
  - In-order:
    - Execution order: given by the program
  - Out-of-order:
    - Execution order: optimized, to run the program faster
    - Yet the semantics of the program is respected

IN-ORDER VS. OUT-OF-ORDER

- Example:

```
i1: R1 ← R2 + R3
i2: R4 ← R1 - R5
i3: R7 ← R8 - R9
i4: R0 ← R2 - R3
```

- In case of a two-way superscalar processor:

In-order:

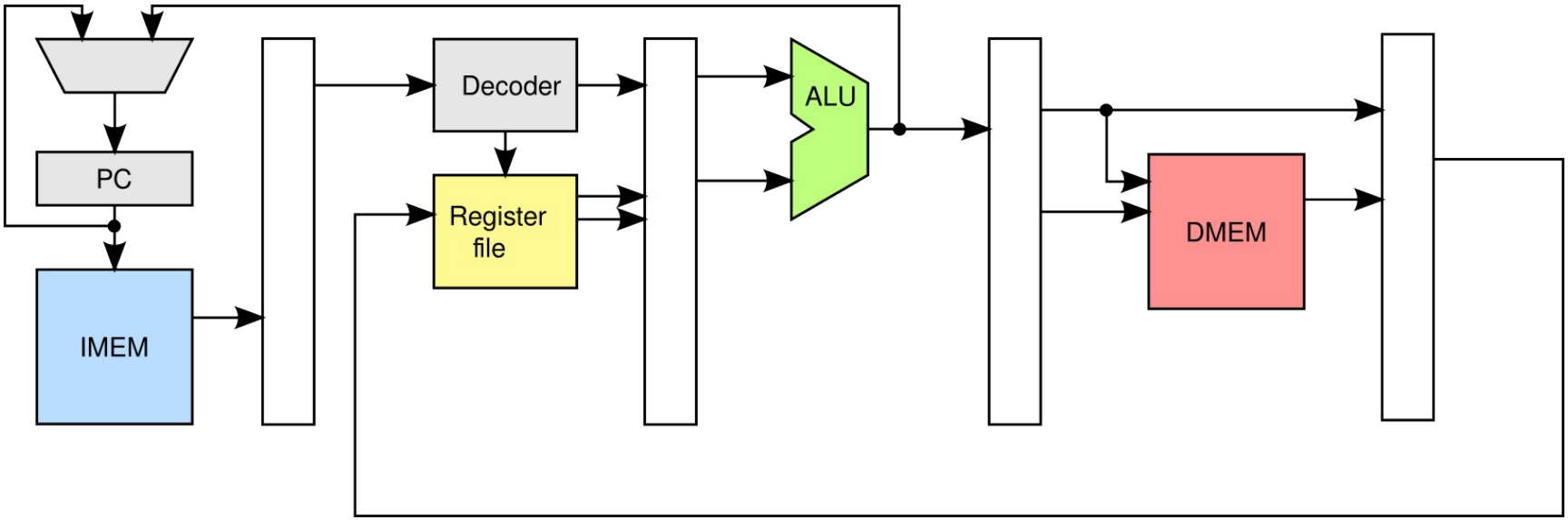
Clock cycle	Instructions
1:	i1
2:	i2, i3
3:	i4

Out-of-order:

Clock cycle	Instructions
1:	i1, i3
2:	i2, i4

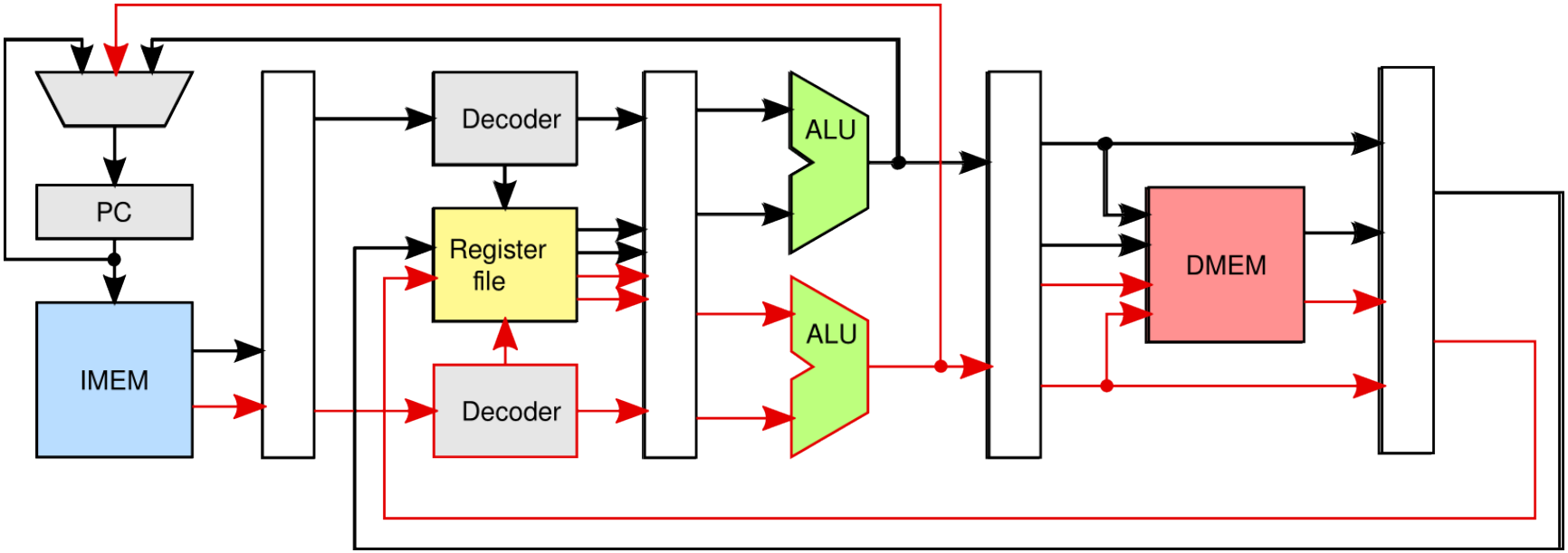
IN-ORDER SUPERSCALAR PIPELINE

”Ordinary”, non-superscalar CPU



IN-ORDER SUPERSCALAR PIPELINE

Two-way in-order superscalar CPU

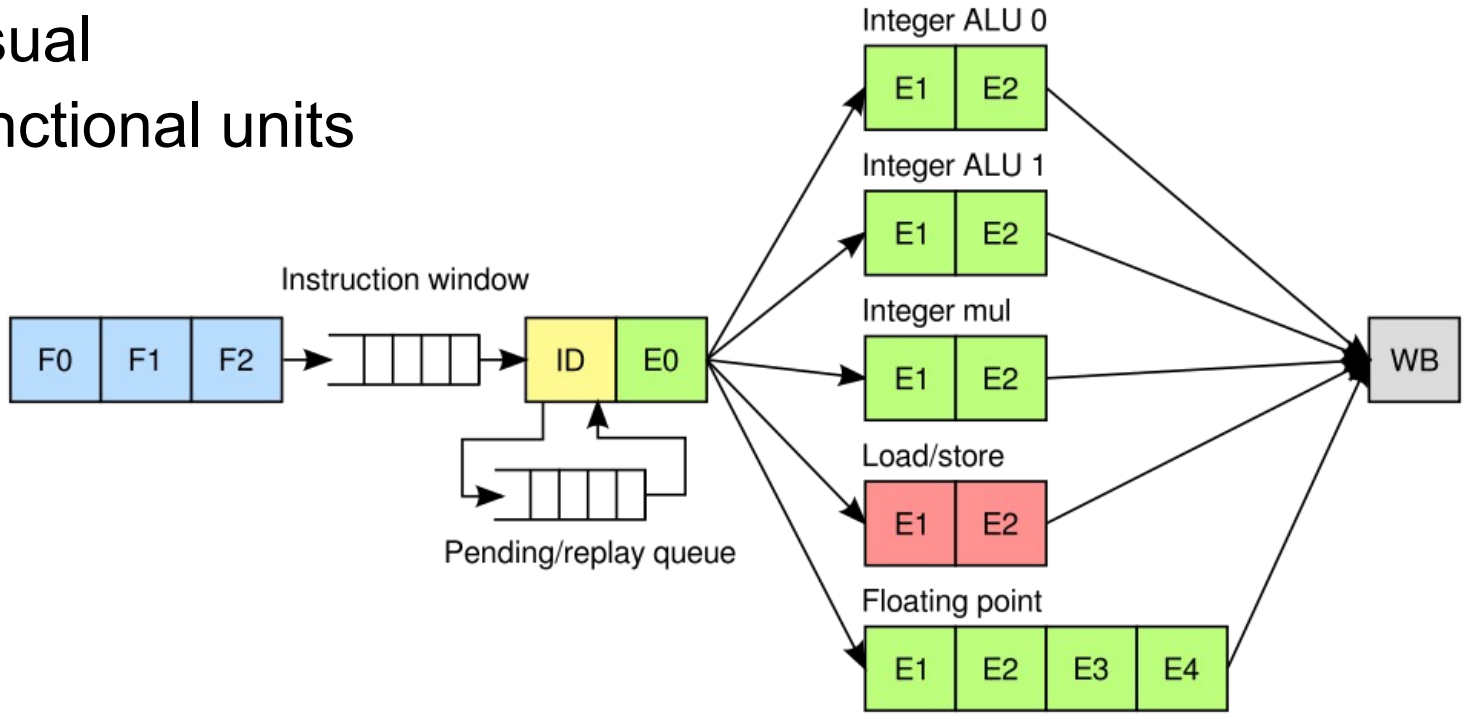


IN-ORDER SUPERSCALAR PIPELINE

- Not essentially different, compared to simple pipelines
- Needs (2-way in-order superscalar):
  - Two decoders
  - Two ALUs
  - A register file with two ports
  - Instruction and data cache with two ports
- Not too difficult to implement
- Problem:
  - The pipeline can not be arbitrarily wide, since the number of forwarding paths increases with a quadratic rate

THE PIPELINE OF ARM CORTEX-A53

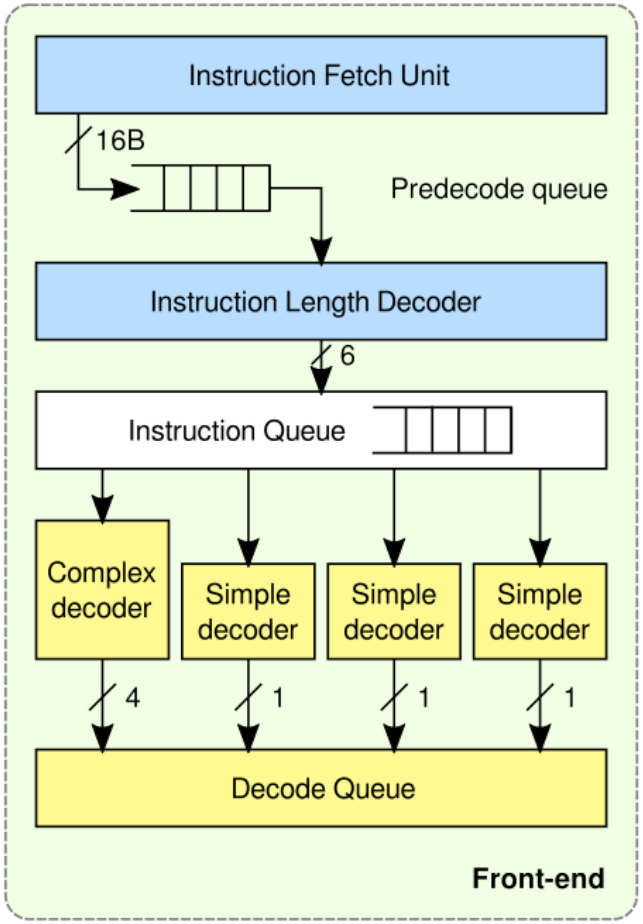
- 2-way in-order superscalar pipeline with 8 stages
- IF: 3 phases
  - F0: increments PC, for jumps calculates target address
    F1: gives address to instruction cache
    F2: puts the instruction into the queue
- ID: as usual
- EX: 5 functional units

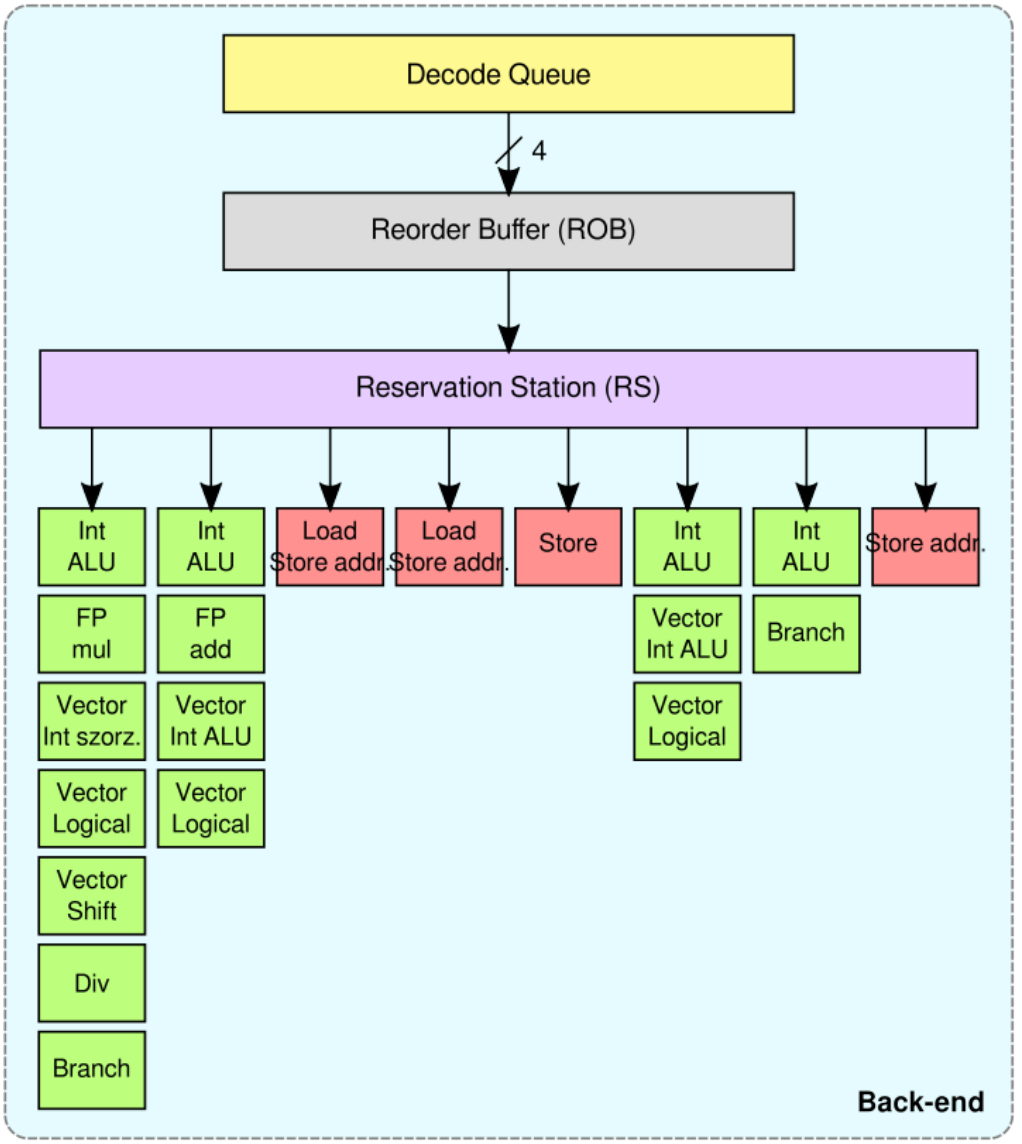


OUT-OF-ORDER SUPERSCALAR PROCESSORS

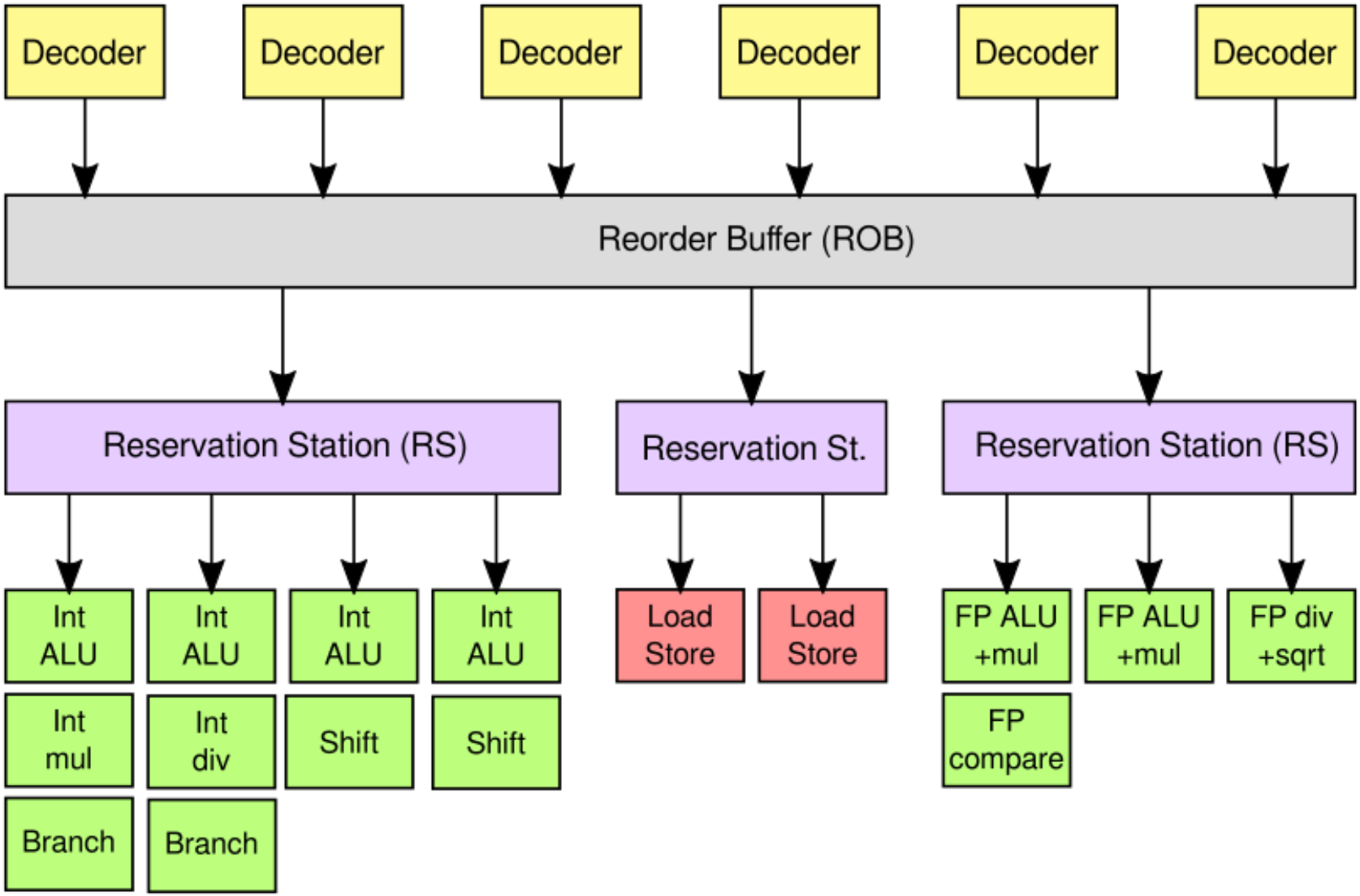
- The out-of-order execution can be easily extended
- In each cycle
  - More than one instruction is fetched and decoded
  - The execution of more than one instruction can be started
- The out-of-order execution principle is the same as we saw

THE PIPELINE OF INTEL HASWELL





THE PIPELINE OF APPLE CYCLONE



Wide pipelines with static scheduling

DYNAMIC SCHEDULING

- What is wrong with dynamic scheduling?
  - The CPU is able to make decisions based on the instructions fetched so far only (those sitting in the reservation station)
  - Complexity
    - A complex algorithm is used that affects speed and cost

STATIC SCHEDULING

- What is wrong with dynamic scheduling?
  - The CPU can make decisions only based on the instructions fetched so far (those that are already in the reservation station)
  - Makes the CPU complex

- The compiler can do the same job! It can detect which instructions can be executed in parallel at compile time

  → ***Static scheduling***

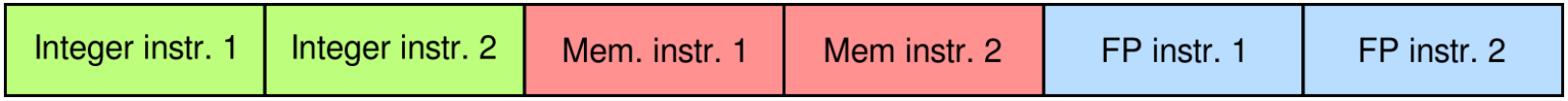
- Why is it better if the compiler does it instead of the CPU?
  - Speed is not an issue at compile time

    → a compiler can use a more sophisticated algorithm that is slower
  - The compiler can see *all the instructions* of the program

    → it can collect independent instructions from a much wider set

THE VLIW ARCHITECTURE

- VLIW = Very Long Instruction Word
- 1 pipeline, but it operates on ***instruction groups***



- Role of the groups:
  - To mark independent instructions
  - To assign functional units to the instructions
- Unused positions in the group contain a NOP (No operation) instruction
- Shocking, but

  VLIW processors are not able to handle hazards!
- What happens if there is a data hazard, and a stall is required to resolve it?
  - Processor does not care
  - The compiler has to detect it
  - … and generate a group full of NOPs

THE VLIW ARCHITECTURE

- Example:
- Latencies: integer – 1, memory – 3, floating point – 4

```
i1: R3 ← MEM[R1+0]
i2: R4 ← MEM[R1+4]
i3: D1 ← MEM[R1+8]
i4: R5 ← R3 + R4
i5: R6 ← R3 - R4
i6: D2 ← D1 * D1
i7: MEM[R2+0] ← R5
i8: MEM[R2+4] ← R6
i9: MEM[R2+8] ← D2
```

	Int 1.	Int 2.	Mem 1.	Mem 2.	FP 1.	FP 2.
1.	NOP	NOP	i1	i2	NOP	NOP
2.	NOP	NOP	i3	NOP	NOP	NOP
3.	NOP	NOP	NOP	NOP	NOP	NOP
4.	i4	i5	NOP	NOP	NOP	NOP
5.	NOP	NOP	i7	i8	i6	NOP
6.	NOP	NOP	NOP	NOP	NOP	NOP
7.	NOP	NOP	NOP	NOP	NOP	NOP
8.	NOP	NOP	NOP	NOP	NOP	NOP
9.	NOP	NOP	i9	NOP	NOP	NOP

THE VLIW ARCHITECTURE

- We were not able to find a lot of independent instructions
- Several NOPs are inserted
  → low utilization

- Summary: Tasks of the compiler:
  - Forming groups from instructions
    - Trying to achieve the best possible utilization (minimizing the number of NOPs)
    - The instructions in the groups must be independent
  - Scheduling the execution of the instructions
    - It must detect hazards
    - And inject the necessary number of NOPs to resolve the hazards

THE VLIW ARCHITECTURE

- Typical sizes of instruction groups:
  - 3-4, up to 28 instructions in extreme cases

- Everything is done by the compiler

  → CPU has very little to do: it only executes the operations!
- Typical applications of VLIW processors:
  - Cheap embedded system requiring low energy consumption
  - Digital signal processors (DSP) (e.g. TMS320C6x – 8 instructions/group)
  - Graphics processors: a huge number of simple execution units are needed. AMD graphics cards used a VLIW3 or a VLIW4 architecture for a while

THE VLIW ARCHITECTURE

- Drawbacks:
  - The program runs only on the specific processor it has been compiled to

    → If a new generation of the same processor family appears
    - It can not have more functional units
    - It can not have lower latency functional units
  - **Implementing a transparent cache is a problem**
    - Since with caches the memory instructions do not have a constant latency any more (cache hit: fast, cache miss: slow)
    - Compiler does not know the memory latency at compile time

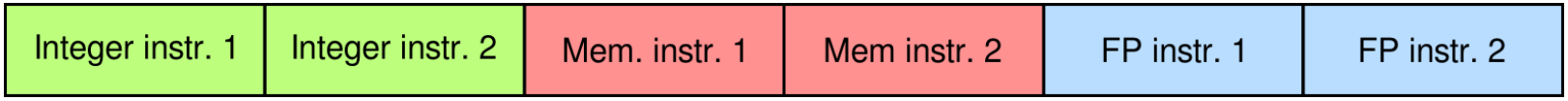
      → VLIW processors do not have caches
  - The size of the programs is very large due to the presence of the large number of NOPs

THE DYNAMIC VLIW ARCHITECTURE

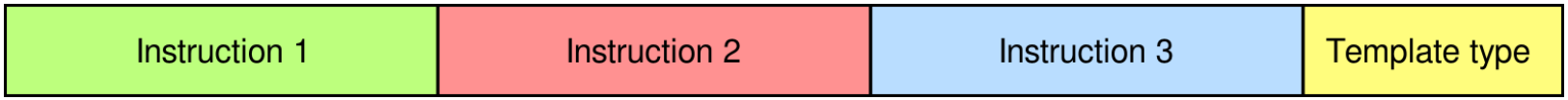
- Compared to the classical VLIW, in dynamic VLIW, we have:
  - The only task of the compiler is to collect independent instructions and form groups from them
  - Scheduling is done by the processor
    - The processor is able to detect hazards
    - And inserts the necessary amount of pipeline stalls automatically
  - Advantages:
    - Cache can be implemented
    - Functional units can be faster in the subsequent generations of the same VLIW processor family
      - The program does not need to be re-compiled to utilize the lower latencies

THE EPIC ARCHITECTURE

- In VLIW, the position in the group selects the functional unit that executes the instruction as well:



- In EPIC it does not. The only role of instruction groups is to encapsulate independent instructions:



- Supported template types are part of the ISA
  - Later CPU generations MUST support them and may add further ones (backward compatibility)

THE EPIC ARCHITECTURE

- It is possible to implement a superscalar EPIC processor
  - With a large number of functional units
  - The CPU is able to execute several instruction groups in parallel
- Instruction groups can be chained
  - Thus the compiler can indicate that it found not only 3, but 6, 9, etc. independent instructions
- → the program does not need to be re-compiled when the capabilities of the CPU extends