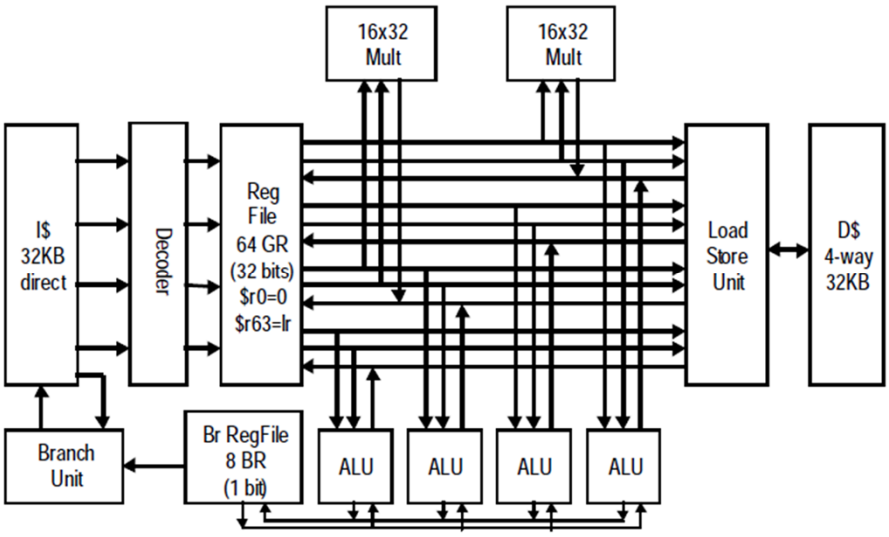
Design-Space Exploration

ASCI Spring School 2017 Lab

Joost Hoozemans
Jeroen van Straten

VLIW Processor

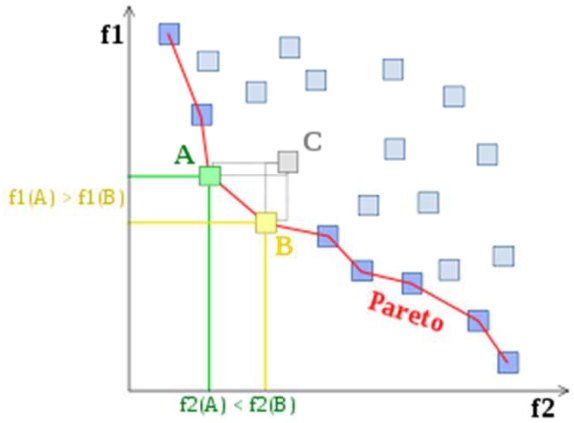


Design-Space Exploration

- Find the optimal configuration given a set of benchmarks
- "Optimal" : design choices
- Metrics: Performance, Energy, Area

There usually is no "optimal solution", but it is important to balance your metrics; How much are you investing in terms of area and power utilization, and how much performance are you gaining?

Design-Space Exploration



Goals

- Generate & evaluate design points (configurations)
  - Using the tools and some scripting
  - Prune design-space using common sense design choices
- Identify most promising candidates
  - Pareto points
- Perform multi-objective optimization (Trade-off)
  - Highest performance
  - Smallest area
- Discuss your findings in a report

Design-Space Exploration

- Find the optimal configuration given a set of benchmarks
- "Optimal" : design choices
- Metrics: Performance, Energy, Area

There usually is no "optimal solution", but it is important to balance your metrics; How much are you investing in terms of area and power utilization, and how much performance are you gaining?

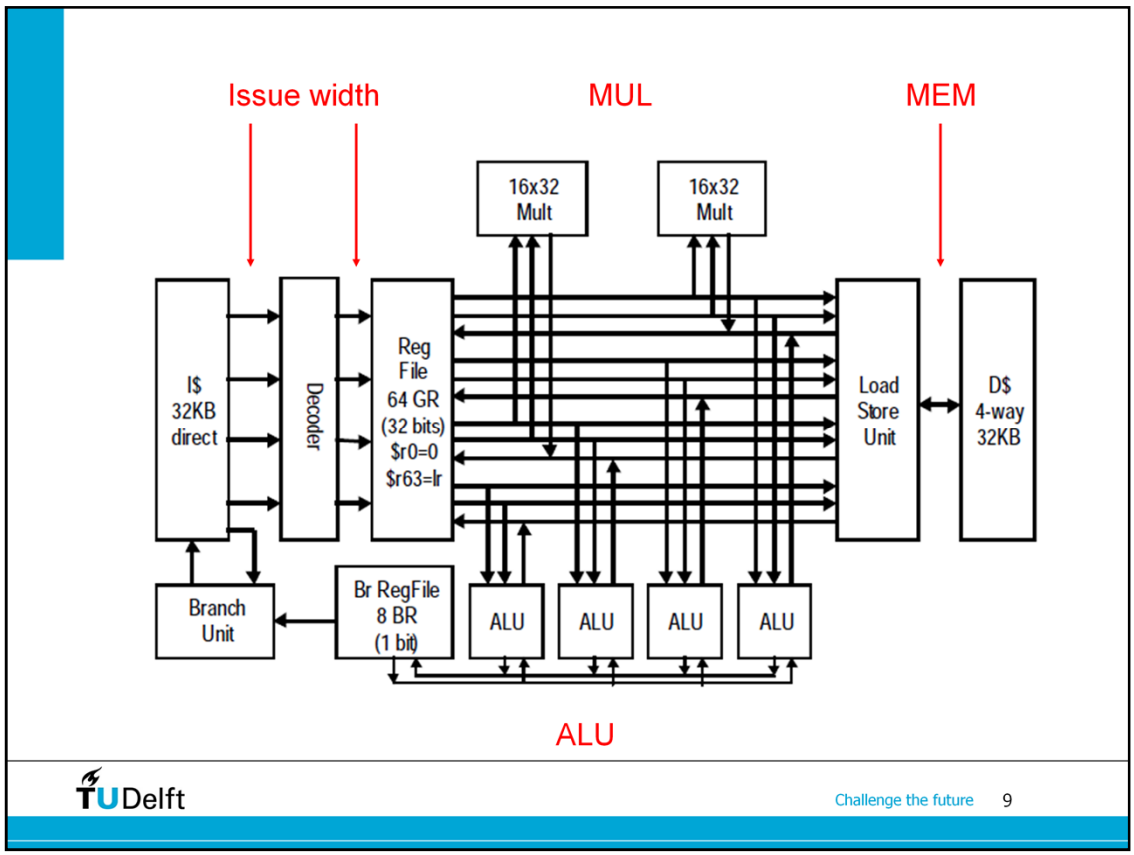
Design-Space Exploration

- Find the optimal configuration given a set of benchmarks
- "Optimal" : design choices
- Metrics: Performance, Energy, Area
- Platform: VEX simulator

There usually is no "optimal solution", but it is important to balance your metrics; How much are you investing in terms of area and power utilization, and how much performance are you gaining?

VEX : VLIW Example

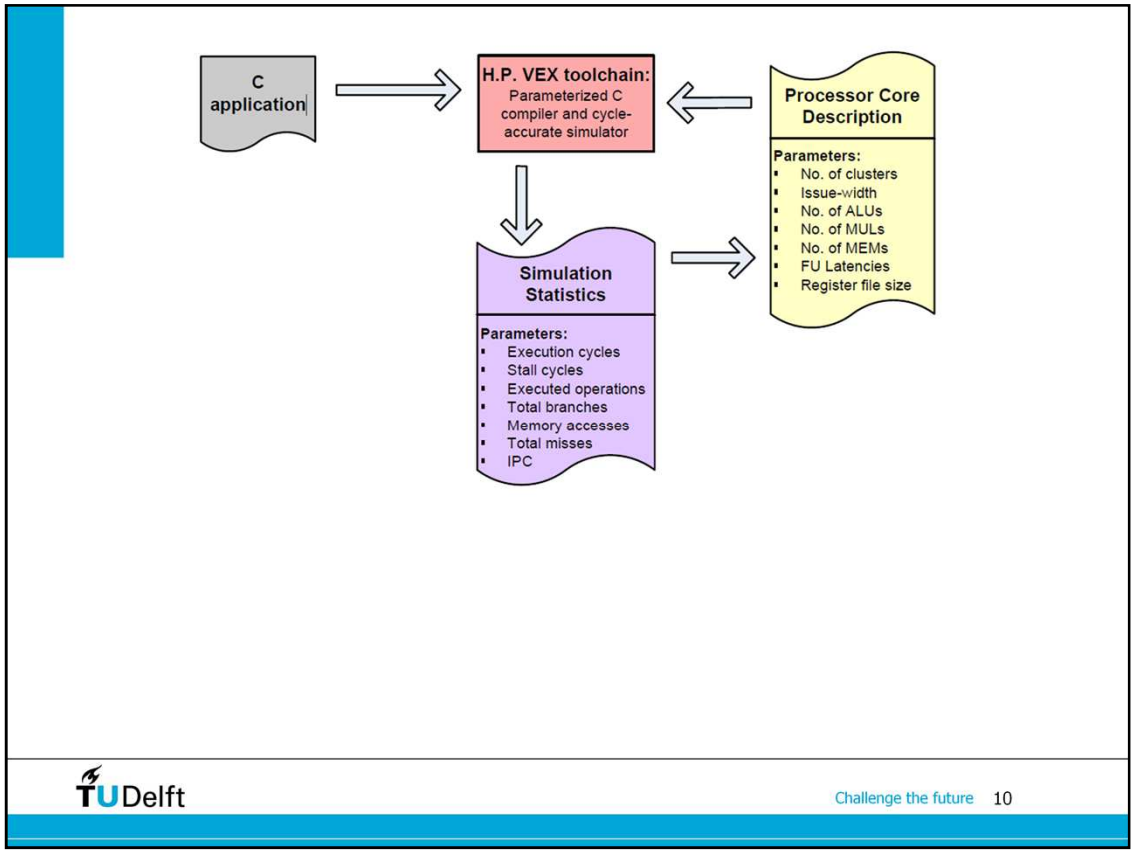
- Family of VLIW processors
- Parameters
  - Issue width (max nr. of operations/cycle)
  - Functional units (ALU, Multipliers)
  - Clustering
  - Number of registers
  - Cache sizes
- Compiled simulator
  - Generates a simulation of your core running a program



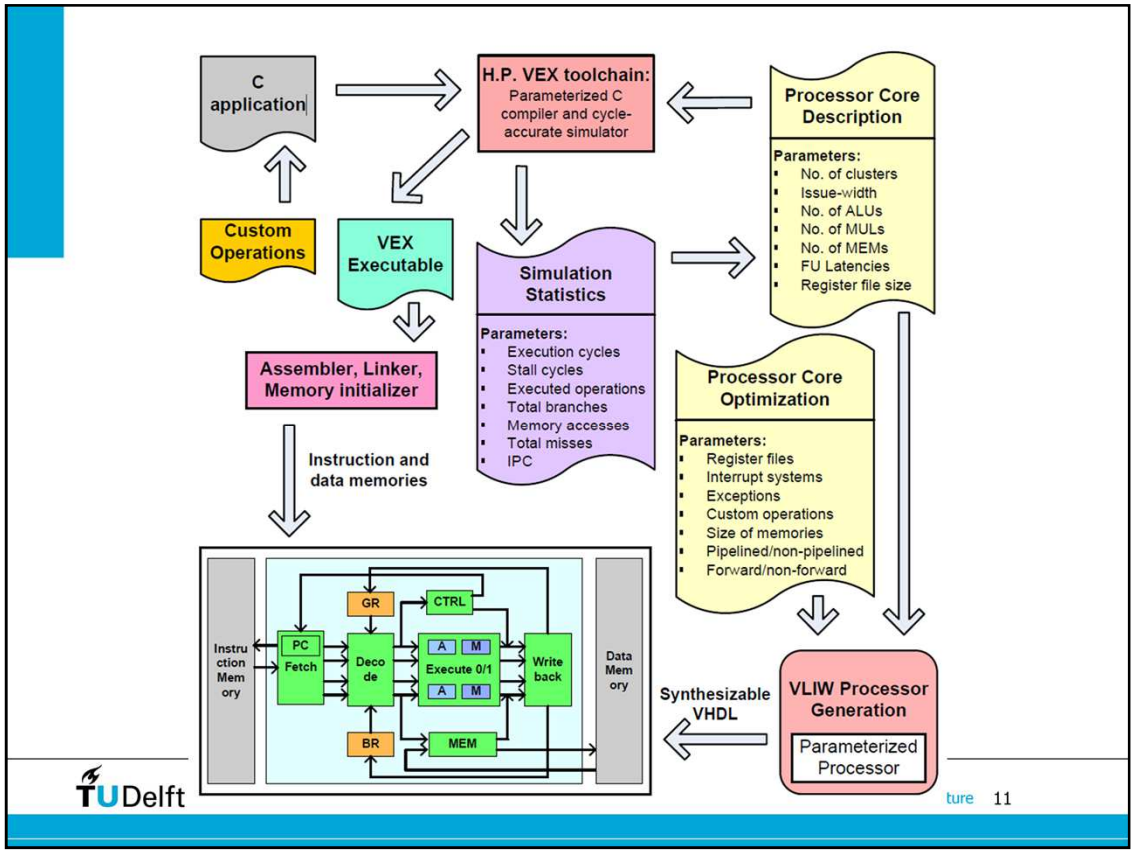
I will add some more information about VLIWs and the VEX configuration parameters. As you will understand, most of this information is somewhat simplified.

This is a diagram of the default VEX core, taken from the vex.pdf documentation. A more simple diagram will follow.

The issue width determines how many operations the core can read from its instruction cache and send to its units. Notice that a lot of complexity will be added when increasing it. The instruction cache must become wider as it needs to load more operations per cycle. The decoding logic will become larger. The register file needs to be able to read and write more operands per cycle, as every operation can read 2 operands and write back 1 result. This leads to a superlinear increase in complexity, so the cost of the register file will increase by more than a factor of 2 when doubling the issue width! Furthermore there are more connections needed to connect all the functional units.



This is the workflow. We will only use the VEX tools. You will need to evaluate configuration parameters for some benchmark programs and choose what you think is the best processor configuration.



This is a more realistic workflow including a parametrized hardware implementation (that can give more realistic area utilization). TUDelft has a VHDL implementation of the VEX ISA. It is parametrized using most of the same configuration parameters as VEX. Using the parameters found using the VEX tools, you wouldbe able to synthesize a processor. However, this is too complex for the current lab.

Programs: Powerstone

- Simple benchmark programs
- Different application domains (embedded, image processing, general-purpose)

Processor core description: Machine configuration file

- Example in workspace
- REG: Resources (functional units)
- DEL: Delays (pipeline)
- REG: Number of Registers

```
# 2 issue vex 1 cluster

RES: IssueWidth 2
RES: Alu.0 2
RES: Memory.0 1
RES: Mpy.0 2
RES: CopySrc.0 2
RES: CopyDst.0 2

DEL: AluR.0 0
DEL: Alu.0 0
DEL: CmpBr.0 1
DEL: CmpGr.0 0
DEL: Select.0 0
DEL: Multiply.0 1
DEL: Load.0 1
DEL: LoadLr.0 2
DEL: Store.0 0
DEL: Pft.0 0
DEL: CpGrBr.0 1
DEL: CpBrGr.0 0
DEL: CpGrLr.0 1
DEL: CpLrGr.0 0
DEL: Spill.0 0
DEL: Restore.0 1
DEL: RestoreLr.0 2

REG: $r0 64
REG: $b0 8
```


Please do not change the delays. The other values you may change. The number of general purpose registers ($r) is 64 at maximum.

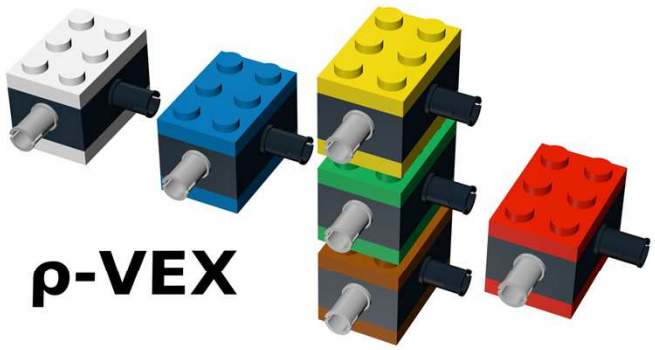
Lab materials

- Lab manual
  - And this presentation – more slides with hints
- The HP VEX toolchain
- Script to facilitate:
  - Generation of machine configurations
  - Area and energy estimation
  - Build & run benchmarks
- Download link: on ASCI webpage

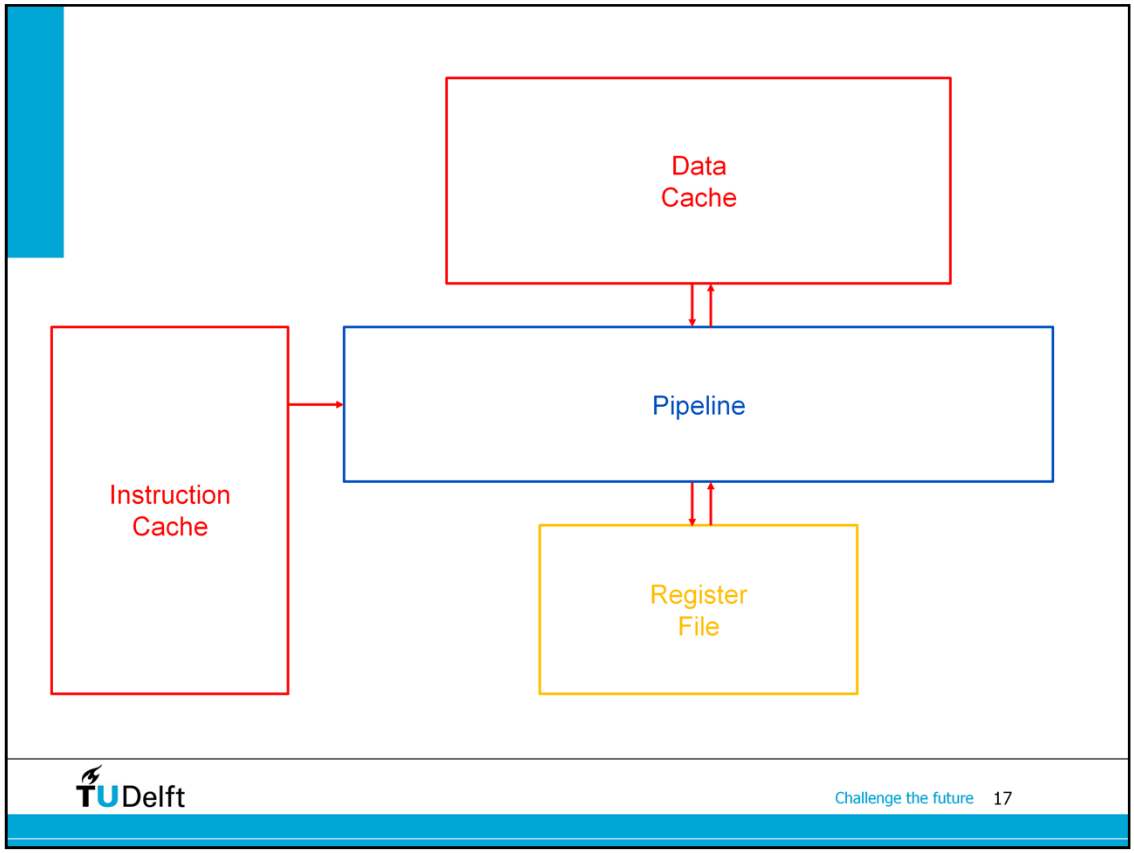
Lab guides

Jeroen van Straten
j.vanstraten-1@tudelft.nl

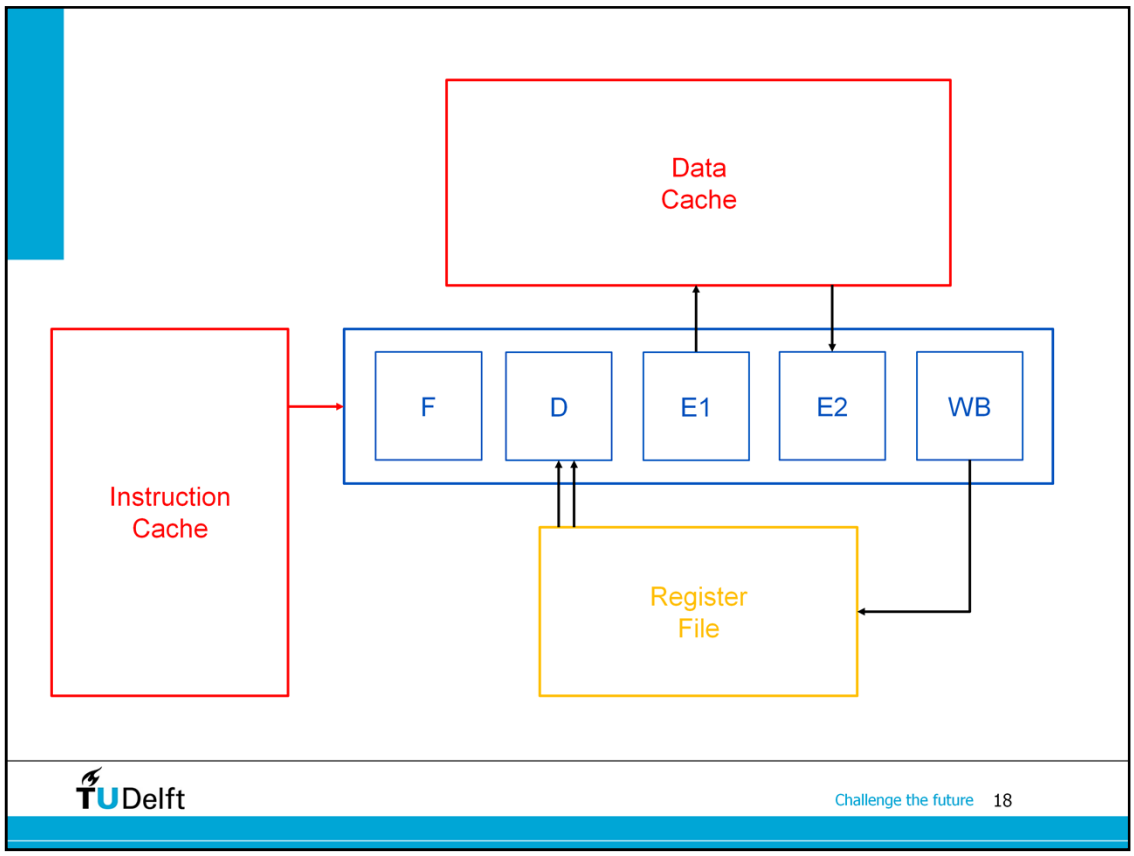
Joost Hoozemans
j.j.hoozemans@tudelft.nl



http://rvex.ewi.tudelft.nl

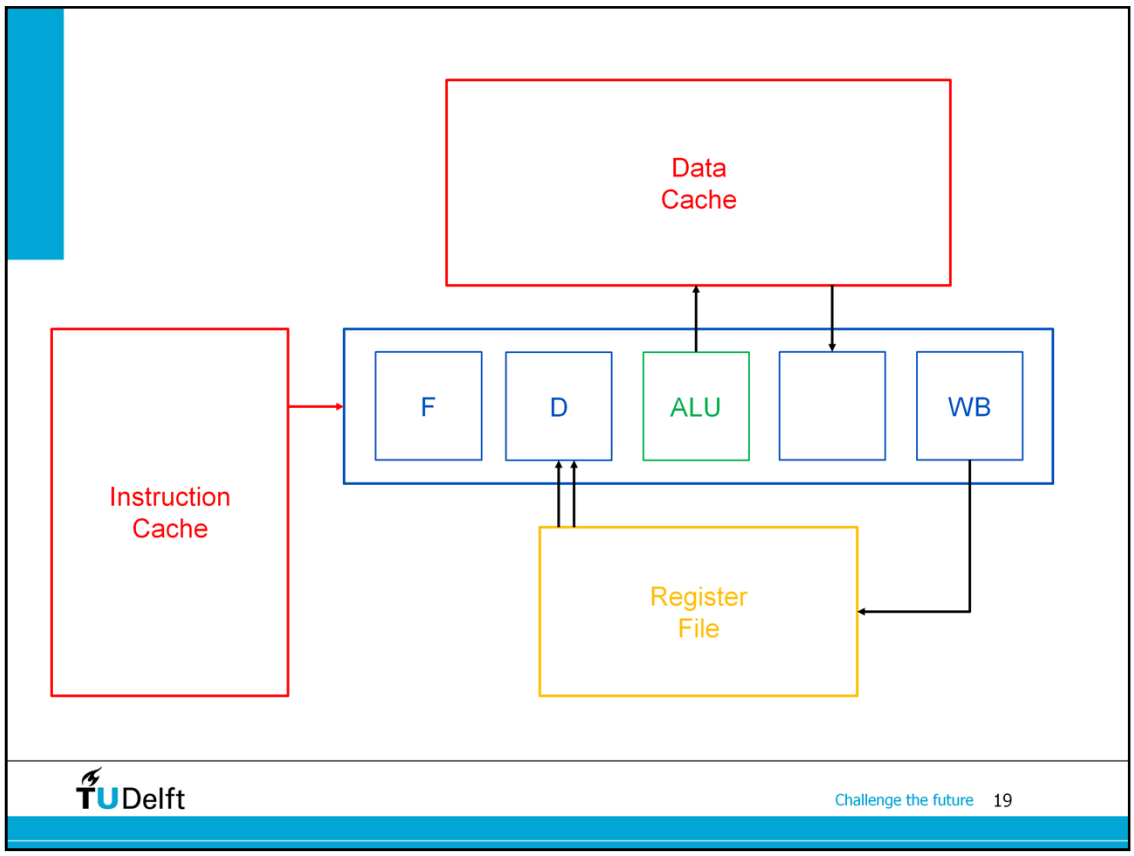


This is a very simple diagram of a scalar processor.

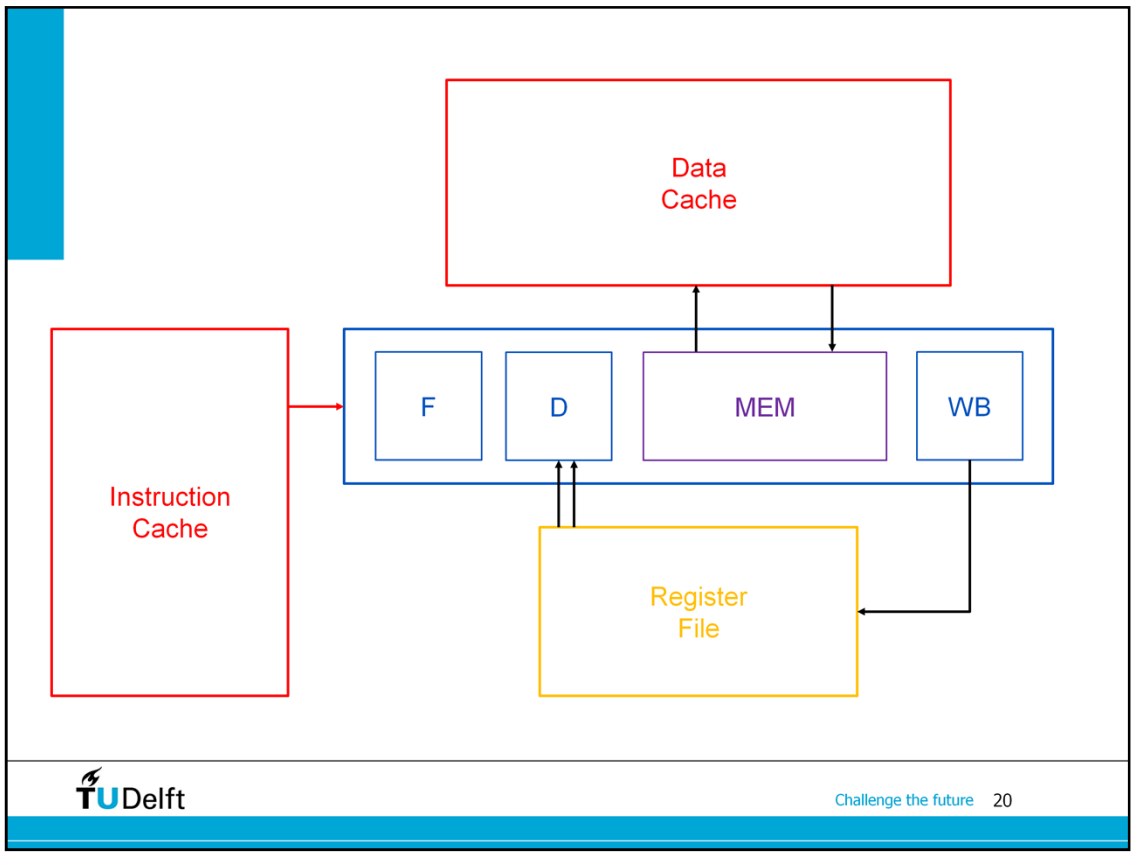


Let's suppose it has a classic 5-stage pipeline just like our TUDelft rVEX processor. Its pipeline stages are Fetch, Decode, Execute1, Execute 2, and Writeback.

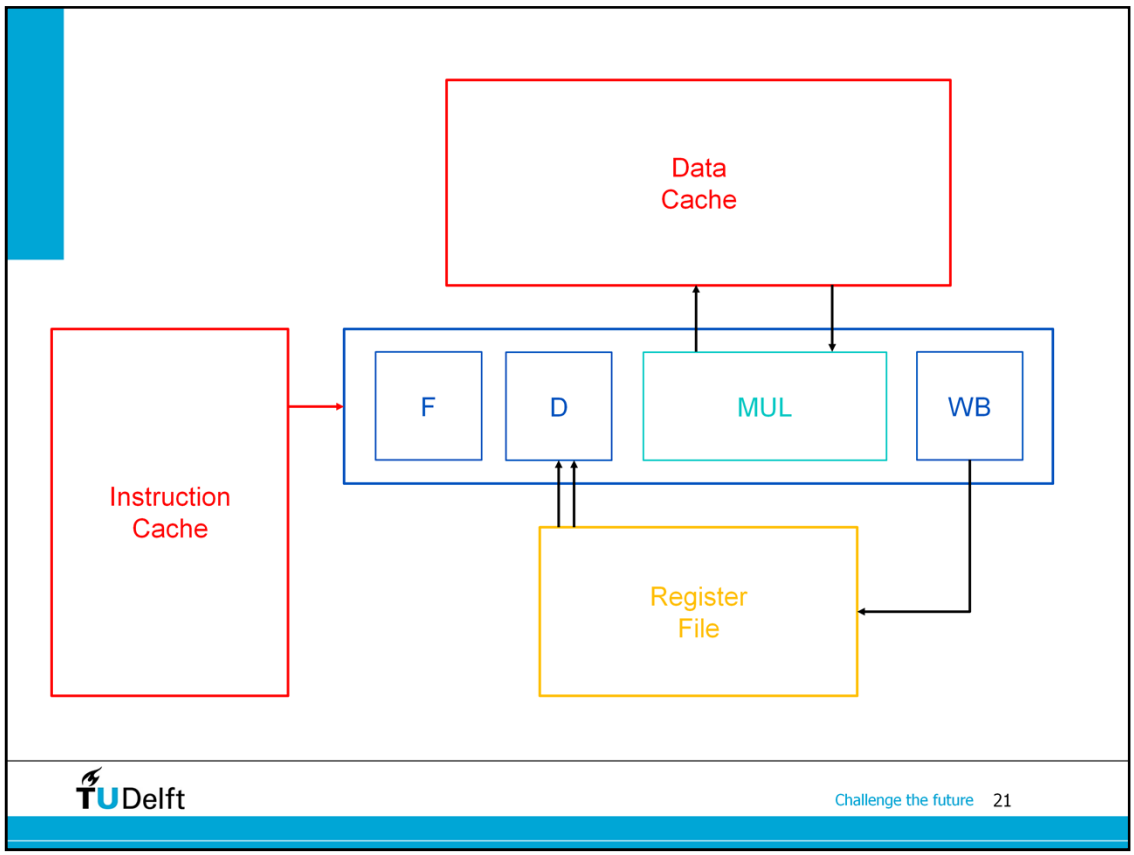
The Fetch stage reads the instruction from the instruction cache, the decode stage reads the operands from the register file, the execute stages perform a calculation or data cache access, and the write back stage writes the result of an operation to the register file.



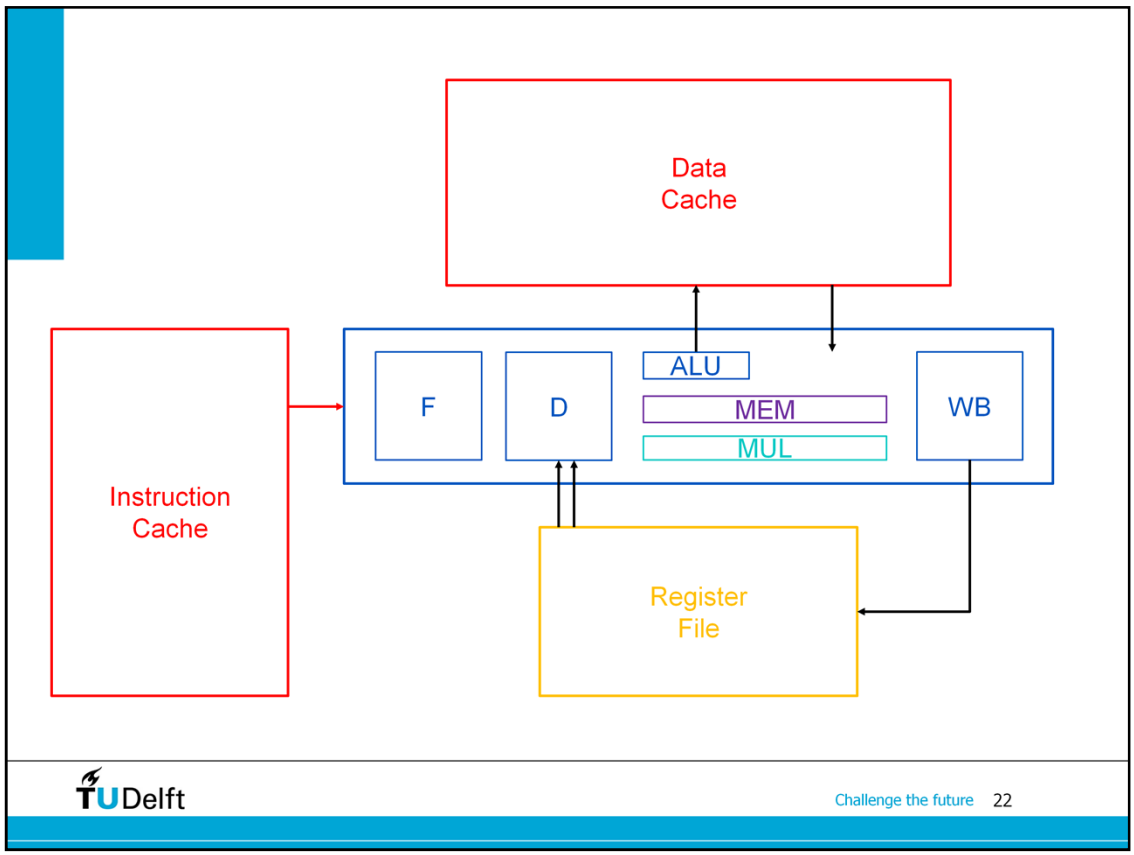
Every cycle, the processor can send 1 type of operation into the pipeline. This is either an ALU type,



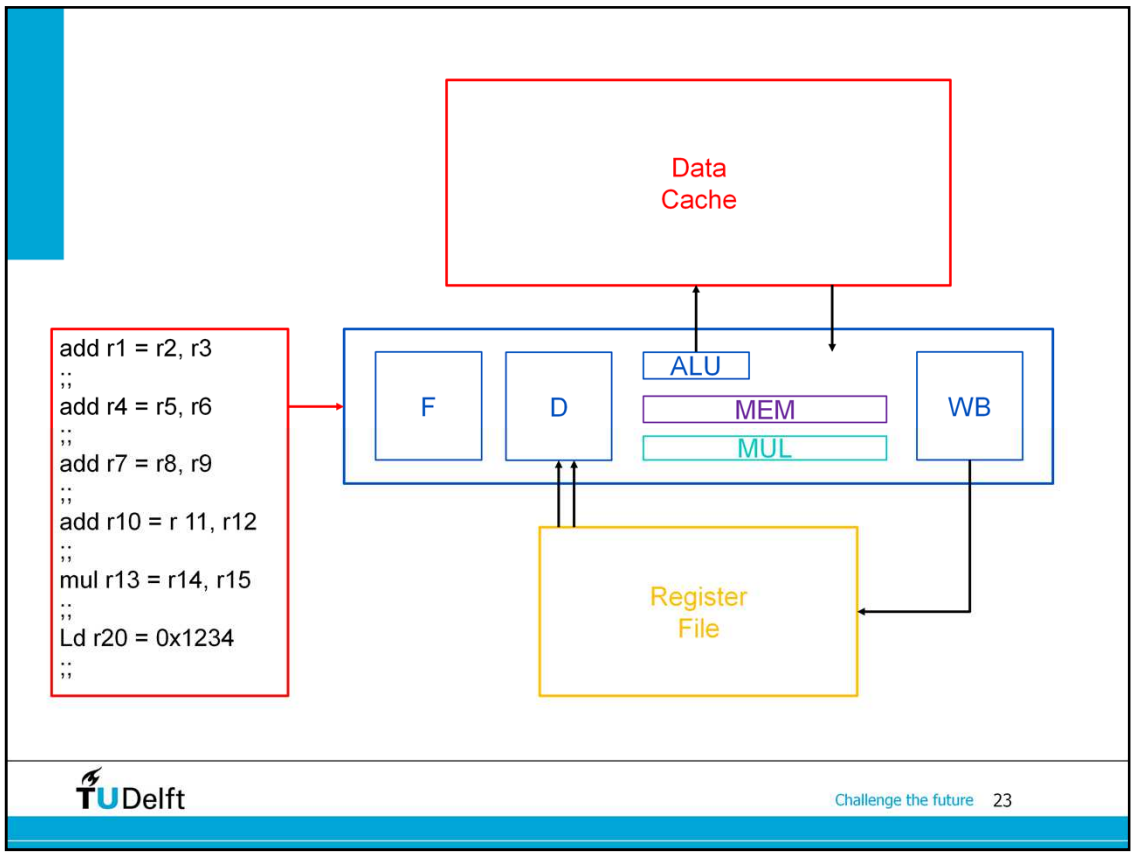
A memory type,



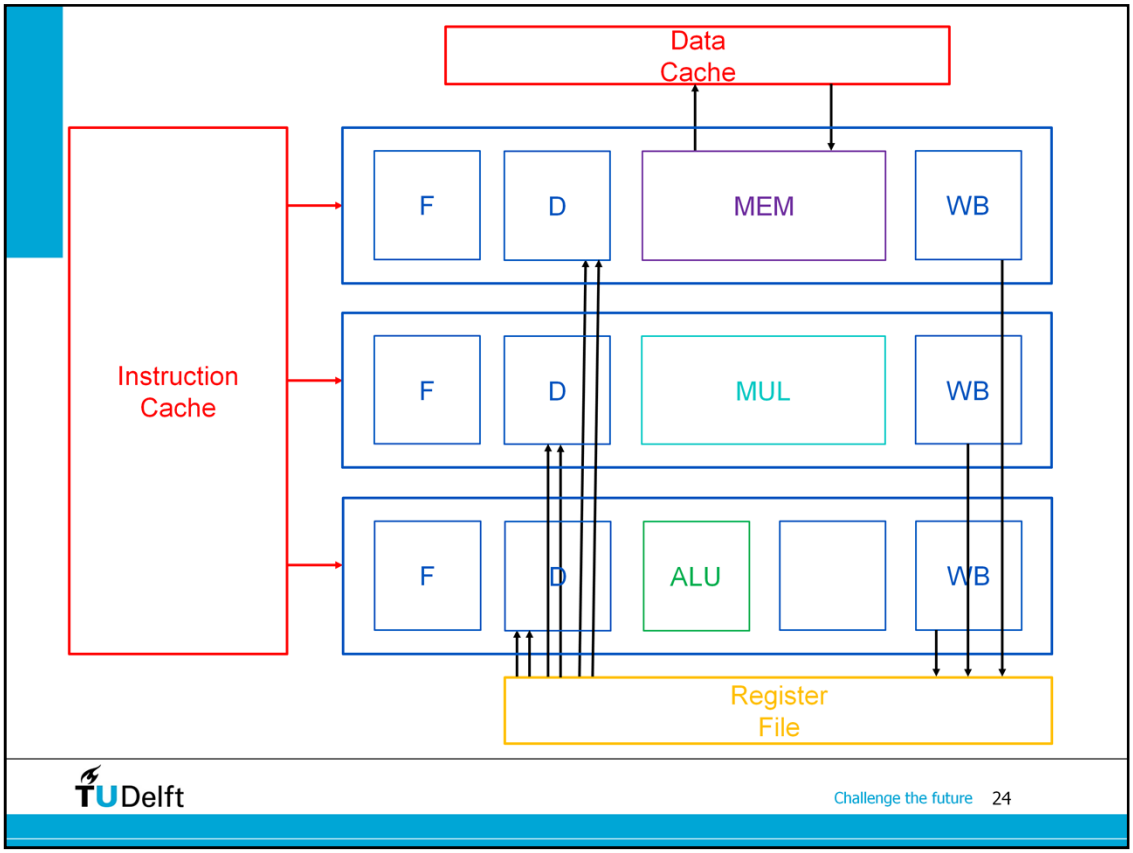
Or a multiplication type (I'm skipping the Branch type here, as it is a fixed functional unit type in VEX – there is alway 1 branch unit)



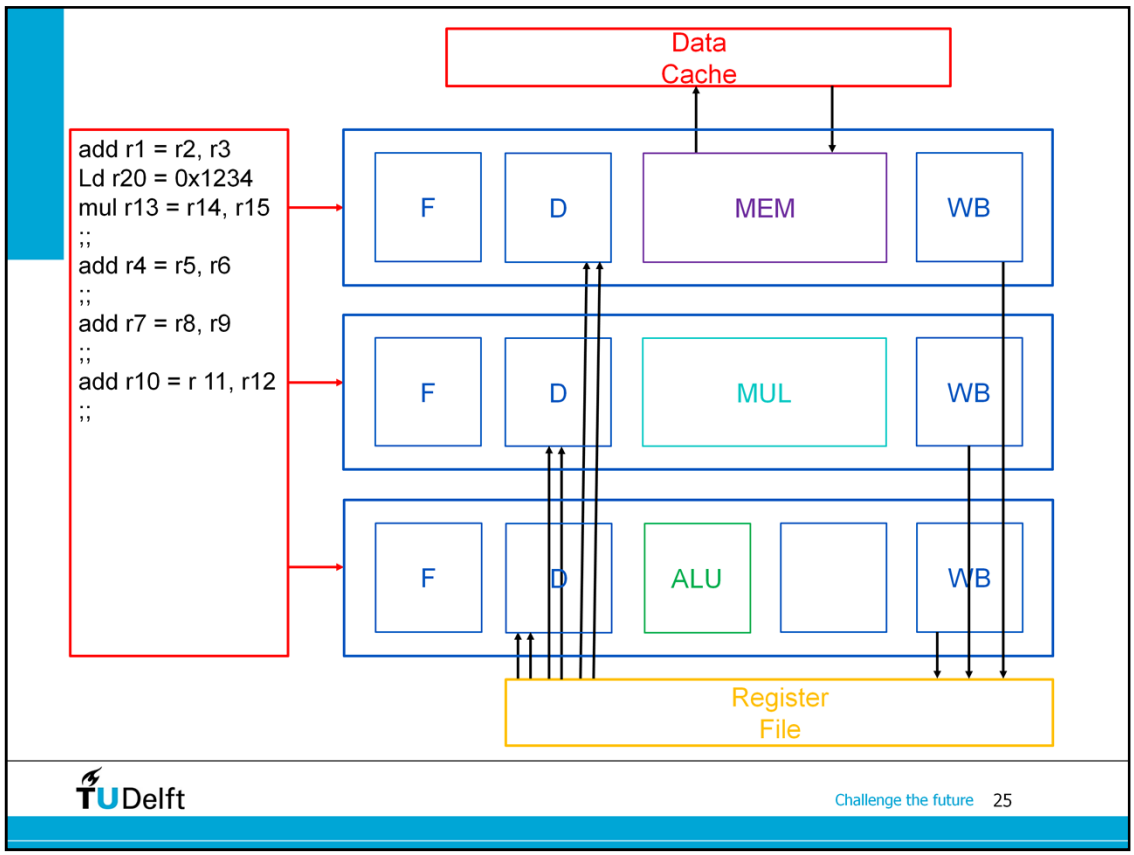
Another way of depicting this. There are 3 functional units but because the rest of the logic (F, D, WB) is scalar, the processor can only use 1 of these units at the same time.



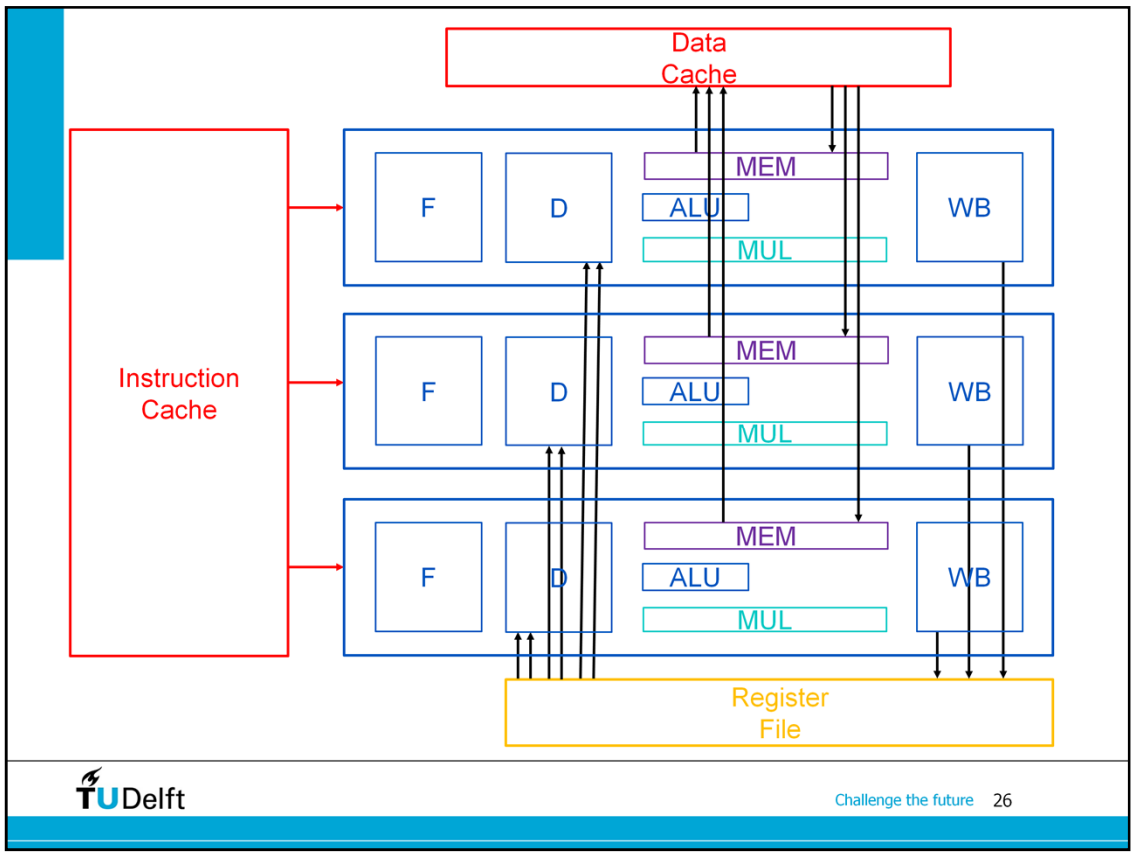
This processor needs 6 cycles to complete this example piece of code (not considering latency). There are no dependencies.



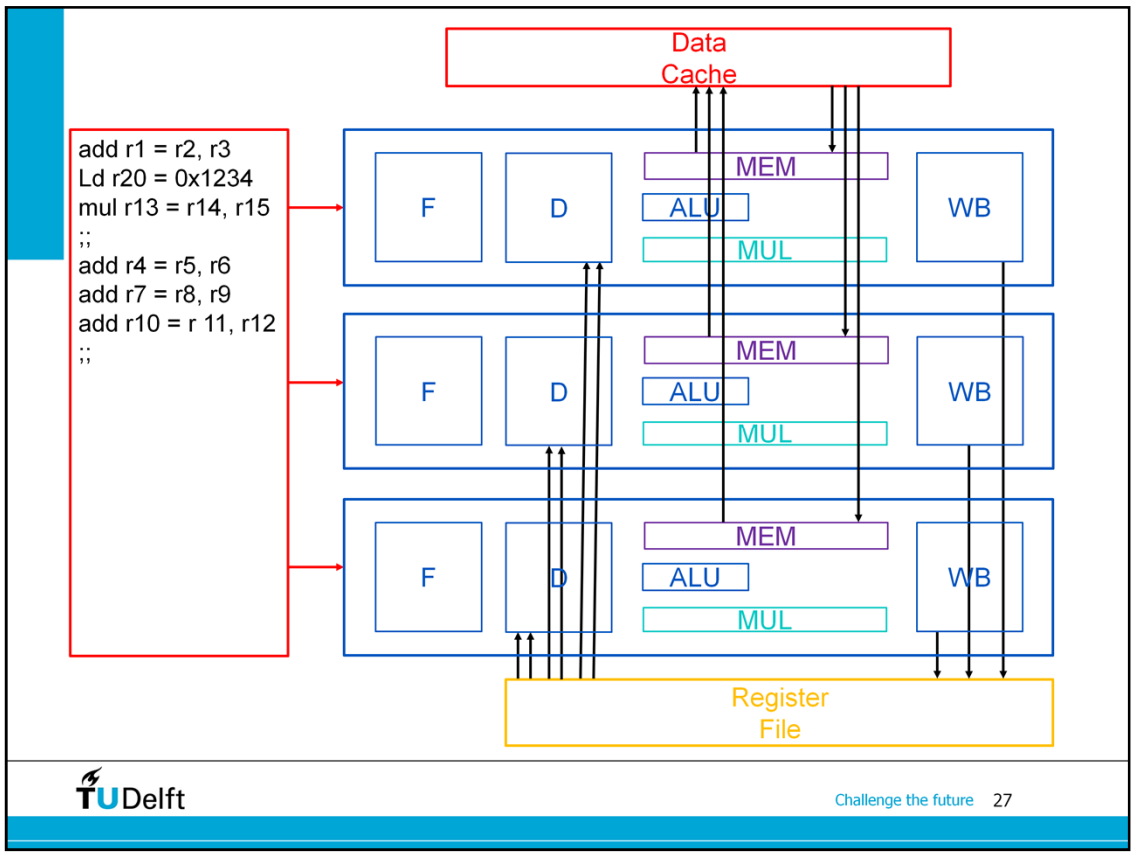
Now let's duplicate the rest of the logic so that the functional units can be used at the same time. This processor is a VLIW. The number of functional units is still 3, but now we have increased the issue width from 1 to 3. So here, the nr. of FUs = issue width. You can see that, besides all other pipeline logic being duplicated, the register file now needs 6 read ports and 3 write ports. This is very expensive in terms of chip area.



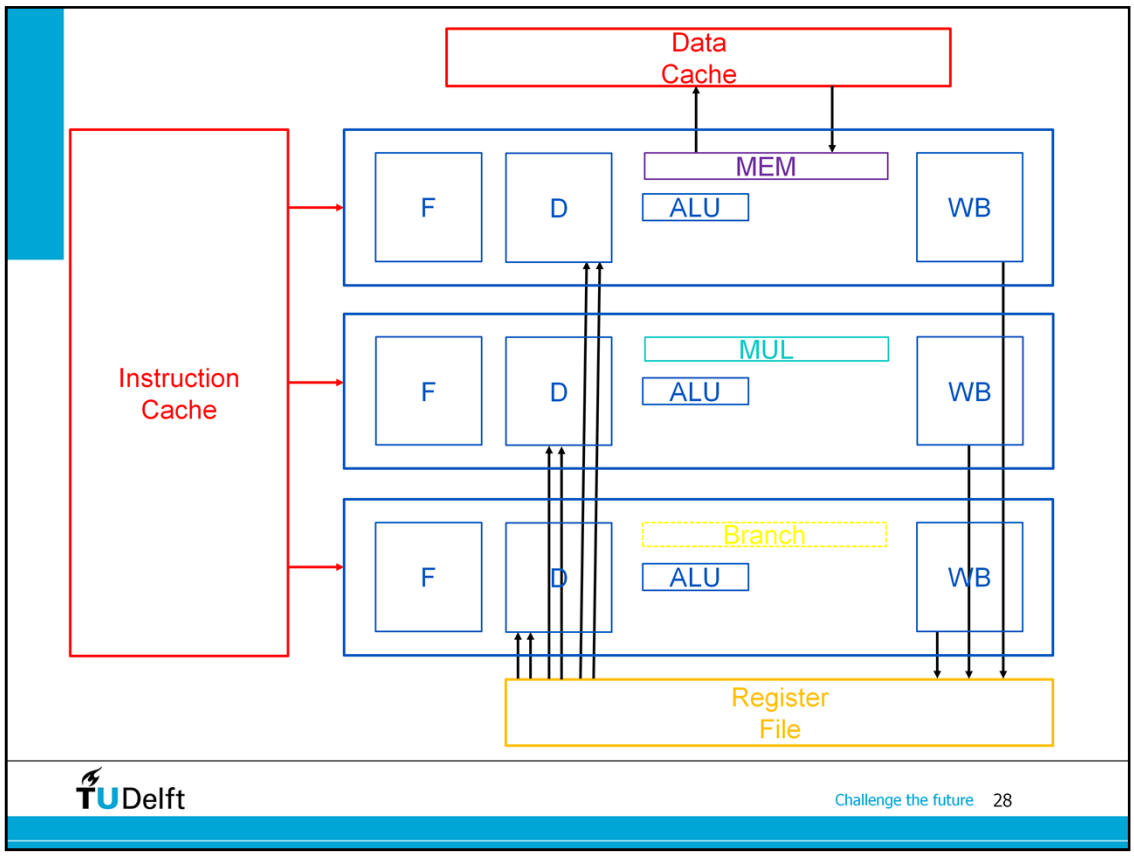
As the functional units can now be used at the same time, the example piece of code can now finish in 4 cycles. However, there is only 1 ALU so the additions need to be performed sequentially. The MEM and MUL pipelines are idle during these last 3 cycles; they are executing No-Operation instructions (the NOPs that are notorious for VLIW processors).



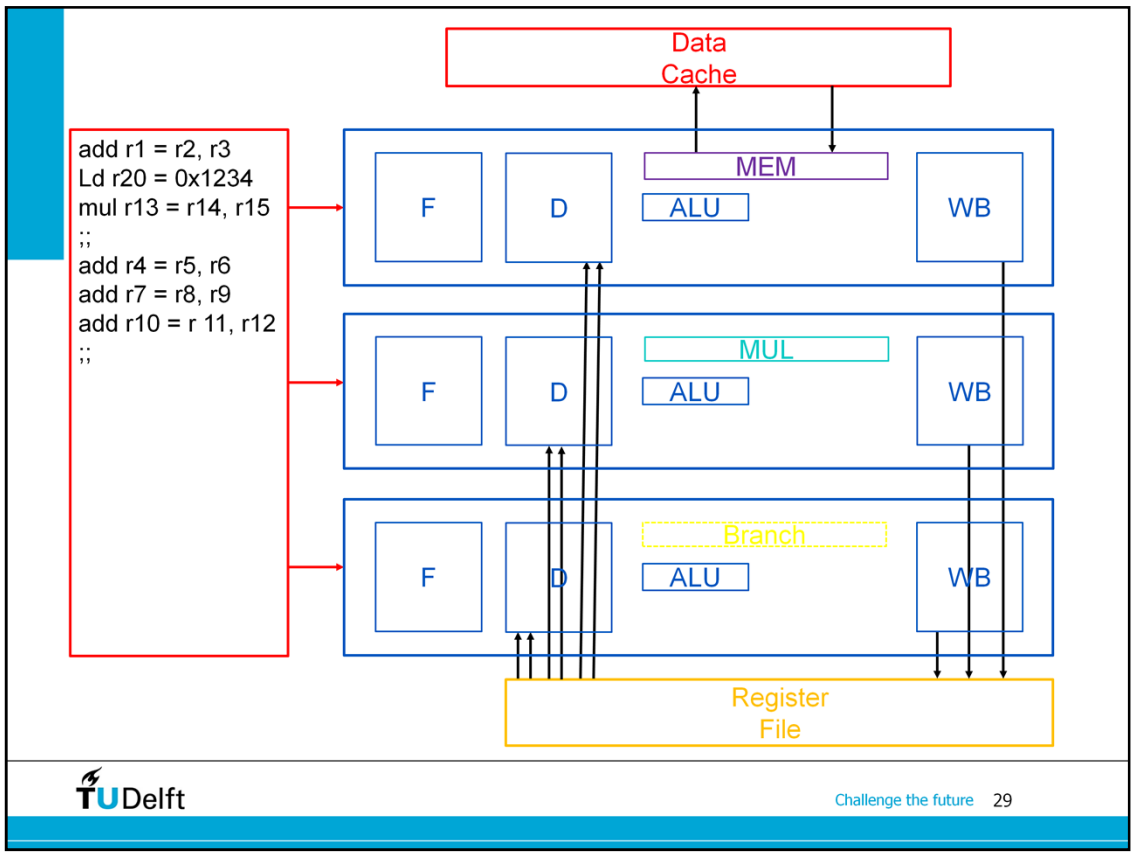
Instead of only duplicating the other pipeline logic, why not just duplicate everything including the functional units?



This design would finish the example code in only 2 cycles. However, note that 6 out of 9 functional units will allways be idle. Also, the data cache needs 3 times the number of access ports which will make it more expensive. Not every application will be able to benefit from this.



This is a design that distributes the functional units over the pipelines, while still allowing most operation types to be executed in parallel. To be complete, I have also depicted the branch unit here to show that every pipeline has 2 units.



The example code will run equally fast on this processor.

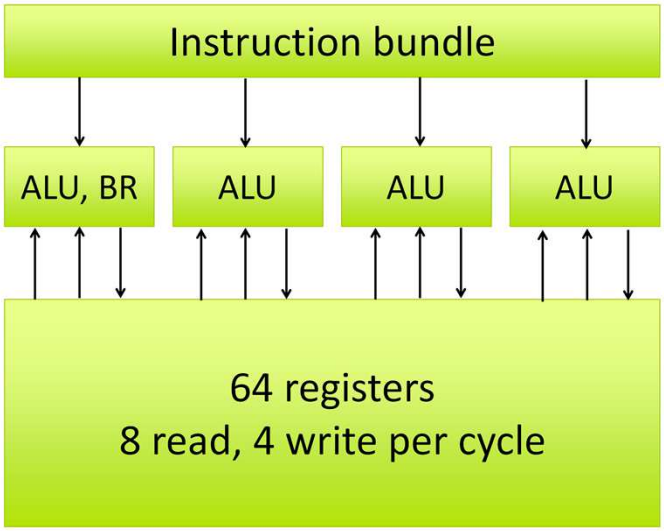
Intermezzo: VEX clusters

- A **cluster** is a section of a VLIW processor that has its own register file and resources
- Only cluster 0 can execute control operations (branch, goto, call, etc.)
- multicluster != multicore

TUDelft

Intermezzo: VEX clusters

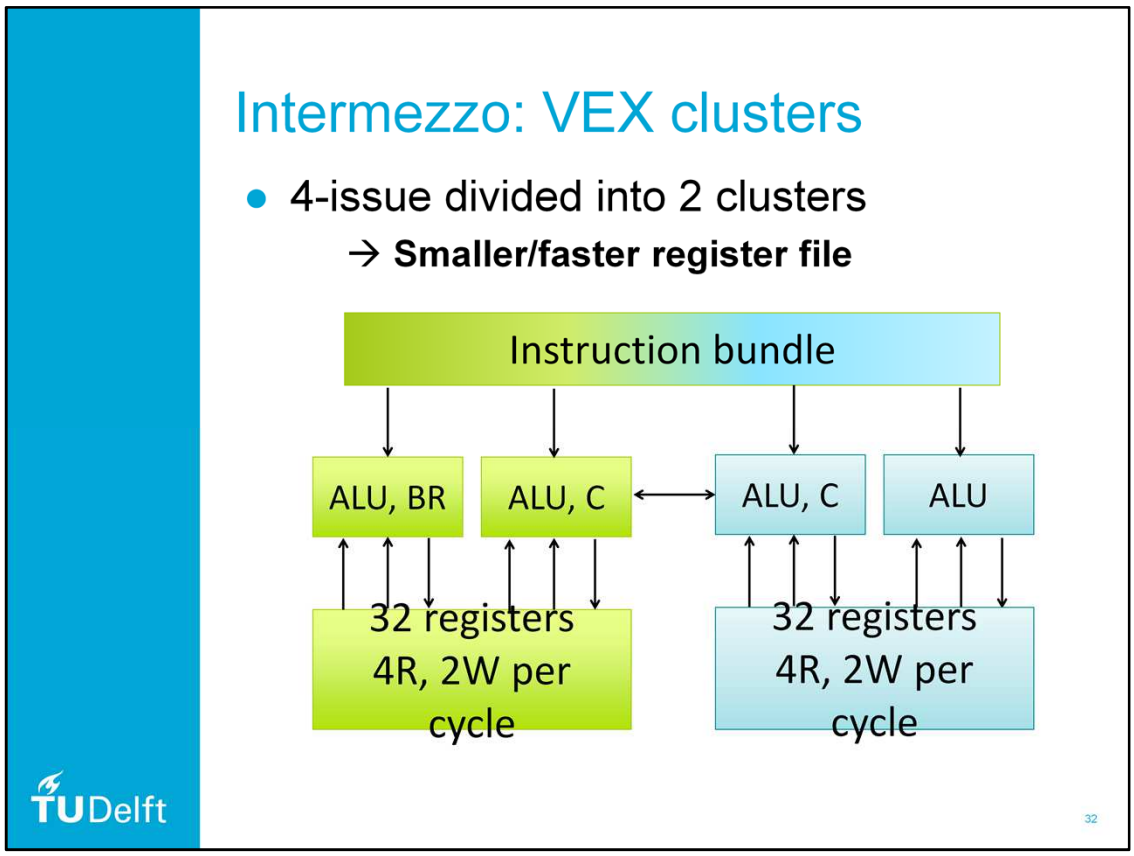
- 4-issue without clusters:



TU Delft

31

Without clustering, the register file grows superlinear with the issue width of the core (more precisely, the number of access ports)



When using clustering, the core is split into multiple parts that all have access to their own section of the register file. Communication between the parts needs to be performed explicitly, and there is a penalty involved! On the other side, the area savings for the register file can be substantial, especially for large issue-width configurations.

Intermezzo: VEX

- c*i* at the start of the line denotes cluster
- So does the first number in the register names

```
;;
   c0     sub  $r0.8  = $r0.13, $r0.15
   c0=c1  mov  $r0.3  = $r1.5
;;
   c0     add  $r0.15 = $r0.15, 32
   c0     shl  $r0.14 = $r0.14, $r0.8
   c1     add  $r1.2  = $r1.6, ~0x3f
;;
```

TUDelft

33

This is an example of intercluster communication