2IC30: Compilers and the structure of computer Systems

Jan Friso Groote

TU/e Technische Universiteit Eindhoven University of Technology

Where innovation starts

Higher languages on the CPU

- Interpreter: Simulates a machine that directly executes the given program.
- Compiler: Translates a given program into the language of the machine that must execute it.
- Mixed forms: Compile to intermediate language, which is interpreted. Interpret, but compile frequently used parts.

Translating higher languages:

- First you need a program to read the source and understands the structure (parser)
- Compile by systematically translating separate programming constructs and combine these translations
  - Non-optimising-compiler:
    fast compilation, but the executable code is not optimal
  - Optimising compiler: slower compilation, but more optimised code

    Often a non-optimising compiler is used during development and an optimising compiler is used for deployment.

Parsing yields a parse tree.

```
// Calculate r = 1*2 * 4*5*...*x

r:=1;
while x>0
do
   if (x≠3) then r:=r*x fi;
   x:=x-1
od.
```

Program does not exactly follow the syntax on the next slides.

Context free grammar
BNF: Backus Naur Form

Expression ::= *Number* |
‘(’ *Expression* ‘)’
Variable |
‘–’ *Expression* |
Expression ‘+’ *Expression* |
Expression ‘*’ *Expression* |
Expression ‘<’ *Expression* |
Expression ‘==’ *Expression* |
not *Expression* |
Expression **and** *Expression* |
Function ‘(’ *Expressions* ‘)’

Examples:

7

8*(3+x)

(x+ -y)

7 == **not** x + 3 < y how to exclude this?

x + y * z how to parse?

x + y + z

More BNF grammar

VariableDeclaration ::= *Type* *Variable*

Type ::= **bool** | **int** | **nat**

Identifier ::= [‘a’...‘z’ ‘A’...‘Z’] ([‘a’...‘z’ ‘A’...‘Z’ ‘0’...‘9’])*

Number ::= ‘0’ | [‘1’...‘9’] ([‘0’...‘9’])*

Variable ::= *Identifier*

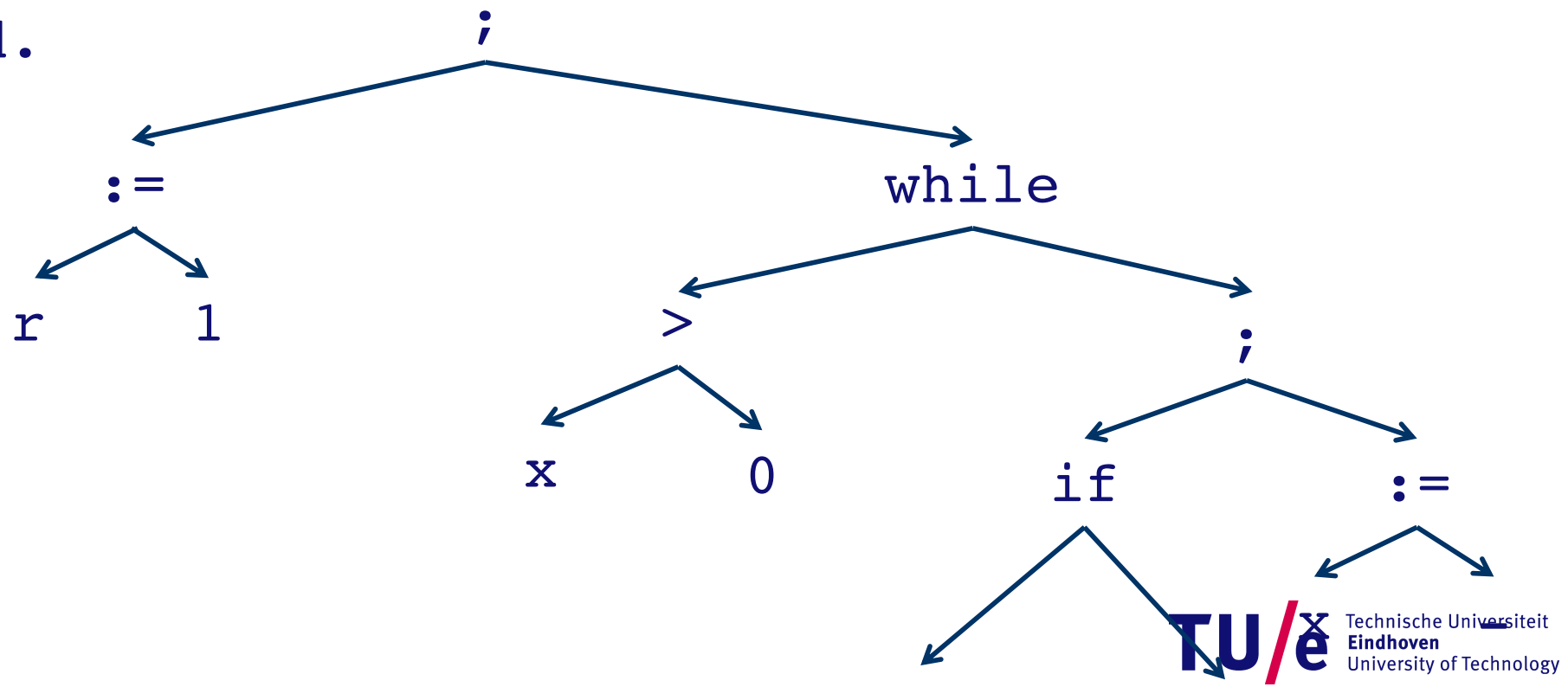
Function ::= *Identifier*

Statement ::= *Variable* ‘:=’ *Expression* |
Statement ‘;’ *Statement* |
return *Expression* |
if *Expression* **then** *Statement* **else** *Statement* **fi** |
while *Expression* **do** *Statement* **od**

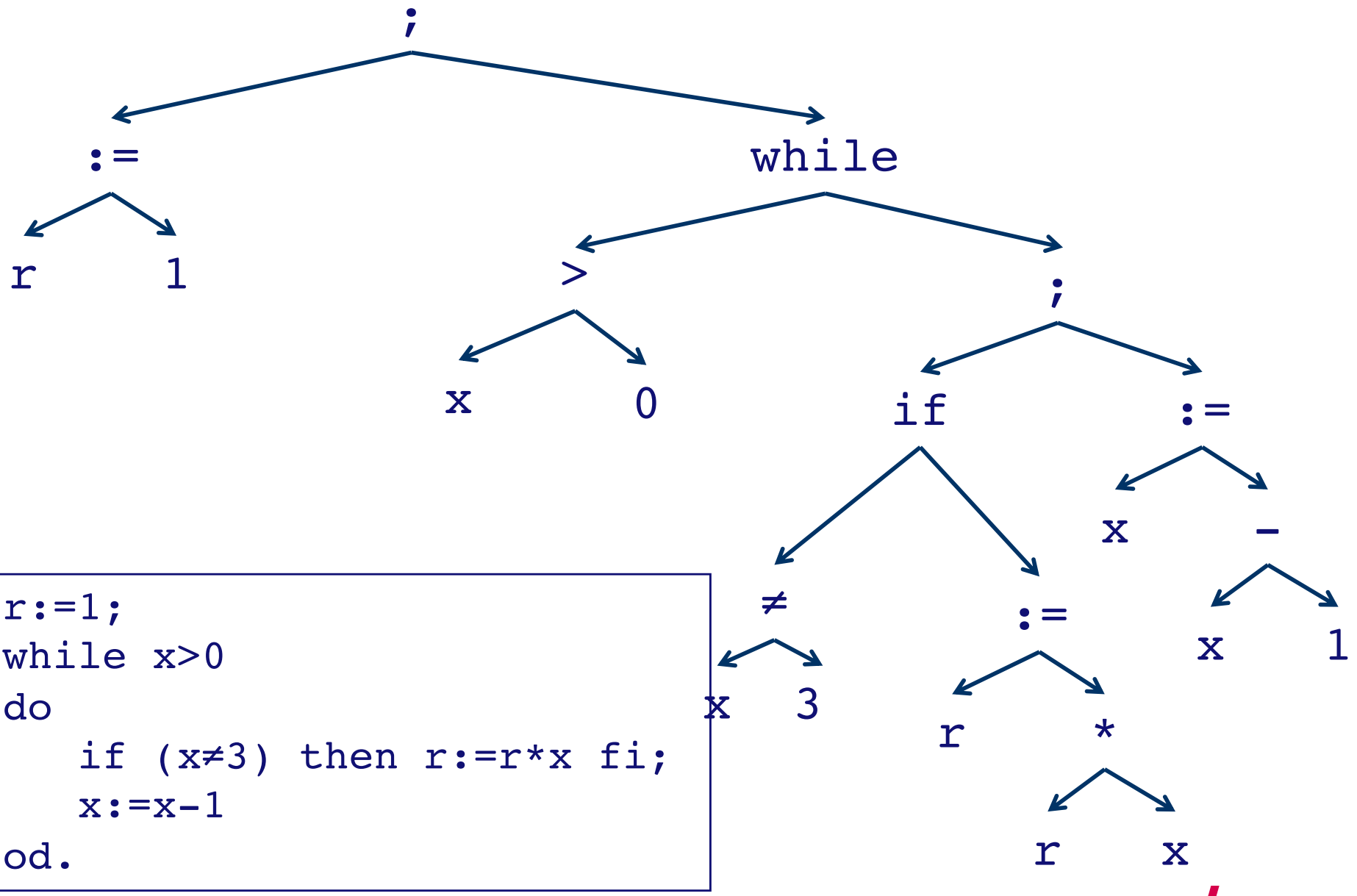
Parsing yields a parse tree.

```
// Calculate r = 1*2 * 4*5*...*x

r:=1;
while x>0
do
   if (x≠3) then r:=r*x fi;
   x:=x-1
od.
```



Parsing yields a parse tree.




```
r:=1;
while x>0
do
   if (x≠3) then r:=r*x fi;
   x:=x-1
od.
```

Translating programs (expressions).

Convention:
The result of evaluating an expression is left on the stack.

$[\![constant]\!]_{pos}$ =

```
LOAD R0 constant;
STOR R0 [--SP];
```

$[\![variable]\!]_{pos}$ =

```
LOAD R0 [SP+pos(variable)];
STOR R0 [--SP];
```

$[\![expr_1 * expr_2]\!]_{pos}$ =

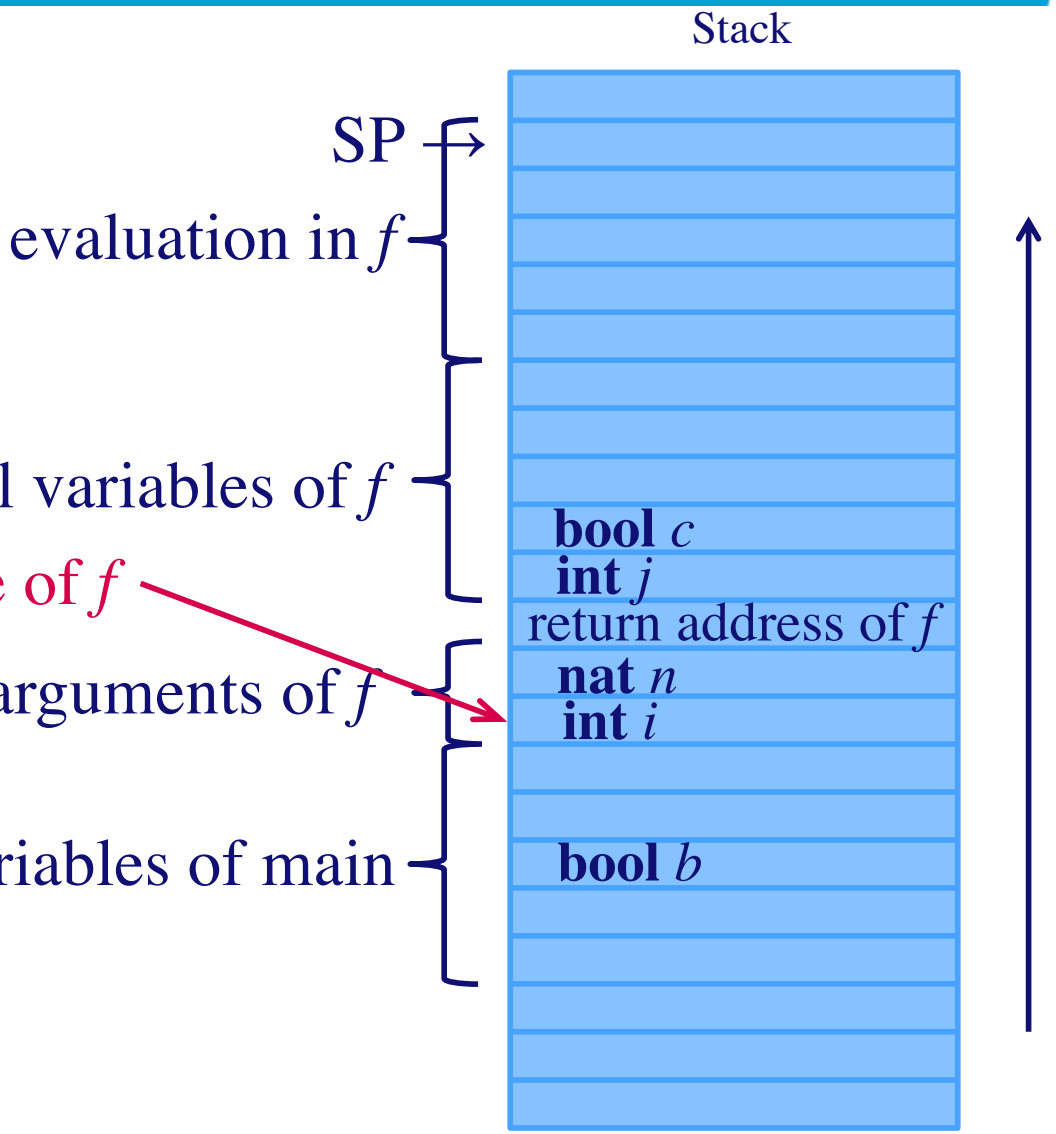
```
[[expr1]] pos;
[[expr2]] pos+1;
BRS calculate_multiplication_on_stack;
ADD SP 1
```

Stack configuration.

```
...
bool b;
...

function bool f(int i, nat n)
   int j;
   bool c;
   ...
   return c
end;

b := f(-3, 5);
```



pos(*v*) gives the relative position of variable *v* to SP

Define: *pos*+1(*v*) = *pos*(*v*)+1.

Translating programs (expressions).

Convention:
The result of evaluating an expression is left on the stack.

$$[\![f(expr_1,\ldots,expr_n)]\!]_{pos} = [\![expr_1]\!]_{pos};$$
$$[\![expr_2]\!]_{pos+1};$$
$$\ldots\ldots$$
$$[\![expr_n]\!]_{pos+n-1};$$
$$\text{BRS calculate\_}f\text{\_on\_the\_stack;}$$
$$\text{ADD SP } n\text{-1}$$

Convention:
We assume that we have a library routine for every function f.

Translating programs (expressions).

Convention:
The result of evaluating an expression is left on the stack.

$[\![variable := expr]\!]_{pos} =$ $[\![expr]\!]_{pos}$;
LOAD R0 [SP++];
STOR R0 [SP+*pos*(*variable*)];

How to compile an if then else?

$[\![$**if** expr **then** B_1 **else** B_2 **endif**$]\!]_{pos}$ =

$[\![$expr$]\!]_{pos}$
LOAD R0 [SP++];
BEQ else_branch; Convention: 0 represents false.

$[\![B_1]\!]_{pos}$
BRA endif;

else_branch:
$[\![B_2]\!]_{pos}$

endif:

How to compile a while loop?

$[\![$**while** expr **do** B **od**$]\!]_{pos}$ =

```
while_loop_begin:
        [[expr]]pos
        LOAD R0 [SP++];
        BEQ end_while;                  Convention: 0 represents false.

        [[B]]pos
        BRA while_loop_begin;
end_while:
```

How to compile sequential composition?

$[\![B_1; B_2]\!]_{pos} =$

$[\![B_1]\!]_{pos}$

$[\![B_2]\!]_{pos}$

Translation of a full program?

$[\![\text{Declarations Statement}]\!] =$ *code*

start_: SUB SP n;

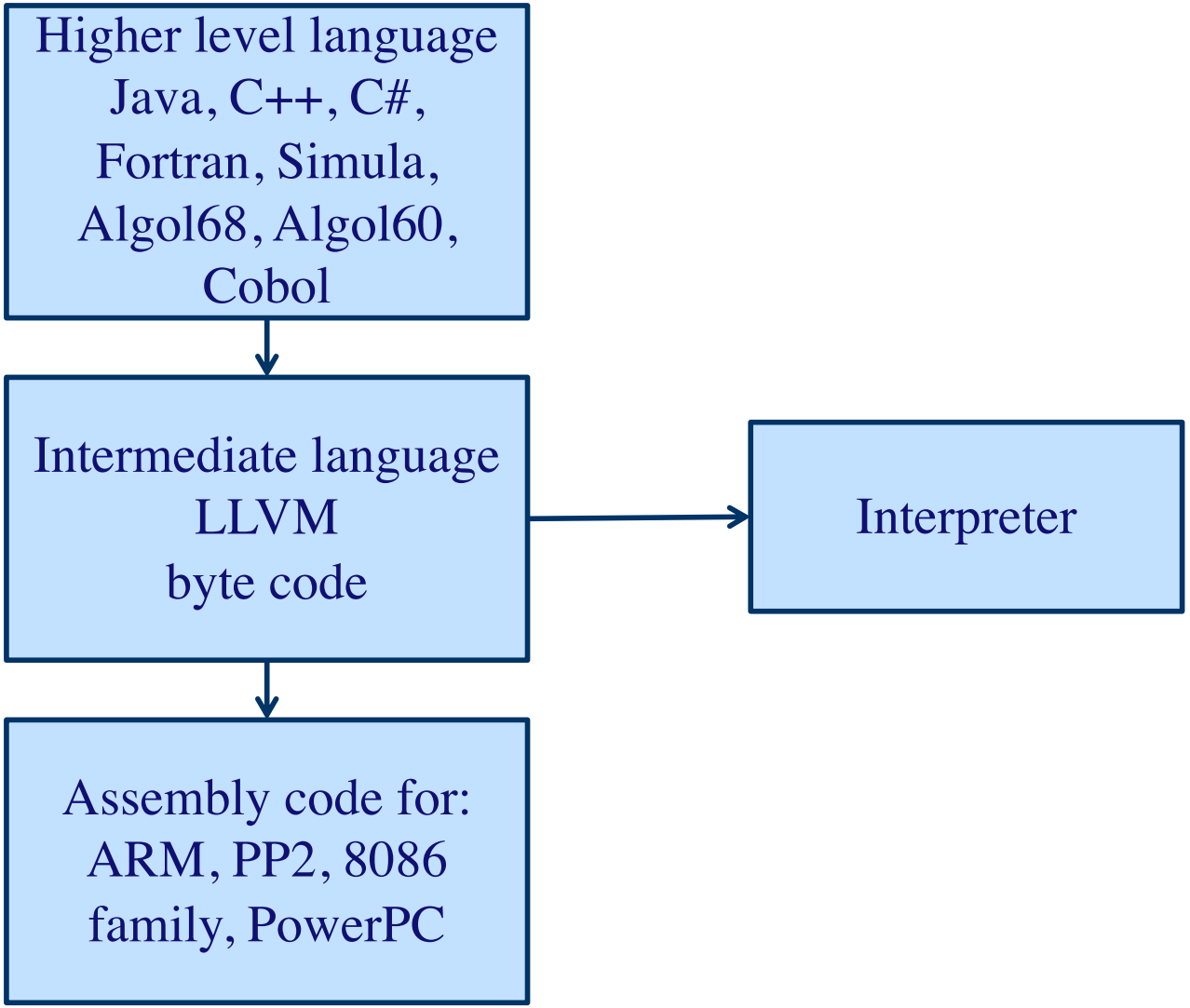
$[\![\text{Statement}]\!]_{pos_1}$

where n is the number of words for local variables and

$\langle code, pos_1 \rangle = [\![\text{Declarations}]\!]_{pos\_init}$.

When defining your own compiler, first define the translation abstractly and maintain this abstract translation.

Intermediate languages



Compilers are now very good in optimising code:

x := c	LOAD R0 "value(c)" STOR R0 [SP+*pos*(x)]
x := x+1	INC [SP+*pos*(x)]

Pre-compute constant expressions.

Direct translation of simple arithmetics.

Use registers for simple (parts of) computations and optimise register transfers.

Do not execute unused computations.

Inlining functions to avoid BRS routines and stack manipulations.

The compiler can reorder instructions, if the (sequential) semantics of the programming language allows so.

10x performance improvement is possible.

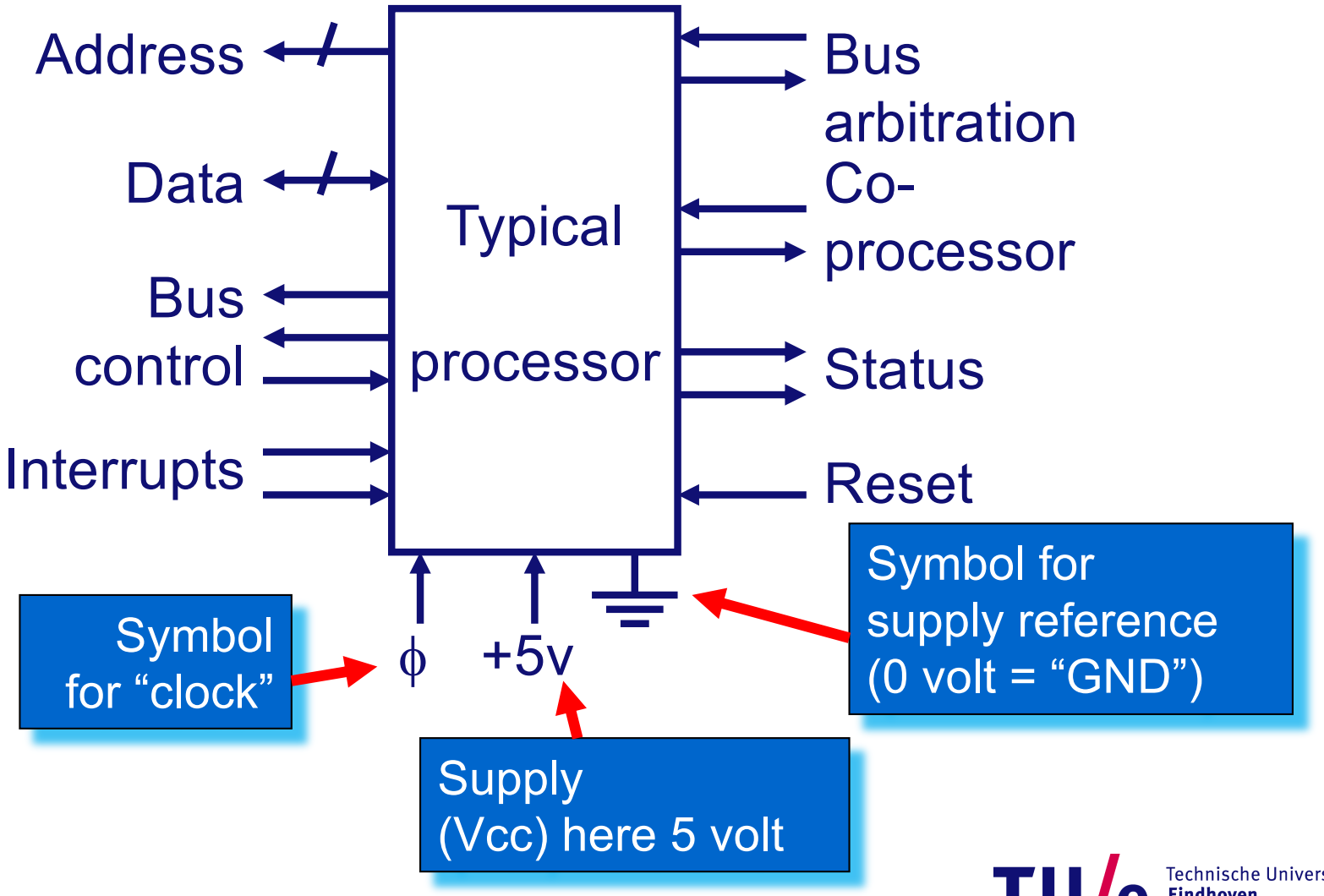
Questions?

The system's architecture

- We consider the computer as a whole:
  - The processor
  - Memory
  - Input / Output (I/O) controllers
  - The "bus"-ses: the infrastructure

- Complete computer on a single chip is possible
  - "Microcomputer" is standard component
  - "System on a chip" specialised (often > 1 CPU)

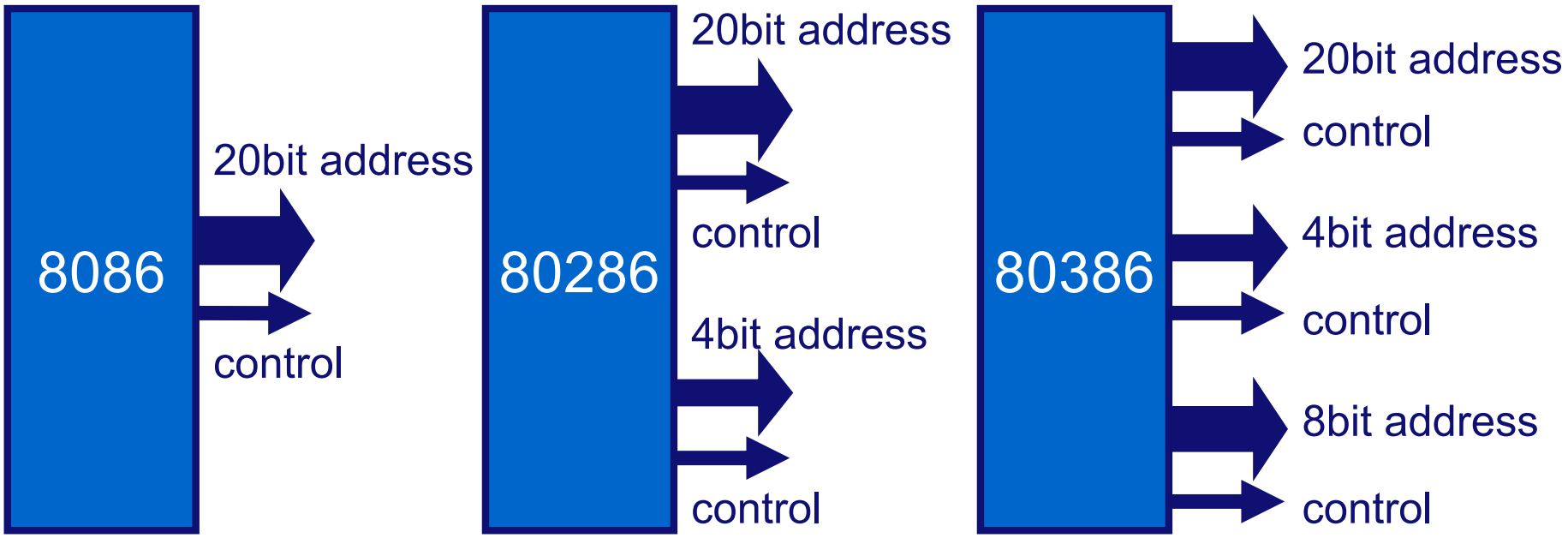
Processor chip symbol



The most important pins of a processor

- Physical electrical contacts (from 8 up to > 500)
  - Grouped by function
  - Make up one or more busses for communication
- Standard bus contains three groups of pins
  - Address: M pins address 2^M memory locations
    common for M: 16, 20, 24, 32, 36, 42, 64
  - Data: N pins parallel I/O – the more the better
    common for N: 8, 16, 32, 64, 128, 256
  - Control: read / write, security, timing….

Standards … change often



Bill Gates (1981): “640 kilobytes ought to be enough for anybody”

4 years later: PC-AT with 16 megabytes

Another 3 years later: “Windows won’t fit !”, so let’s have 4 gigabytes ...

Connecting memory to the CPU

- *Static* RAM and (EP/flash) ROM: easy
  - Just connect address, data and control pins
- *Dynamic* RAM: storage element is a *capacitor*
  - Address is split (RAS/CAS): multiplexer required
  - Charges must be *refreshed* (very) periodically
  - Different control pins
  - Timing is very critical for maximum speed
  - Usually a "*dynamic RAM controller*" is used to implement *internal data retrieval:* DDR2, DDR3, DDR4, GDDR5.

Dynamic RAM.

SDRAM: one word per clock cycle

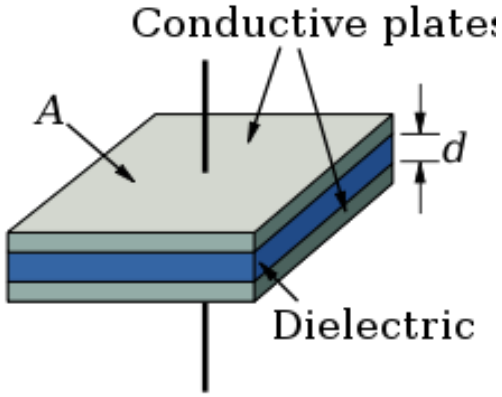
DDR: Two words transfer

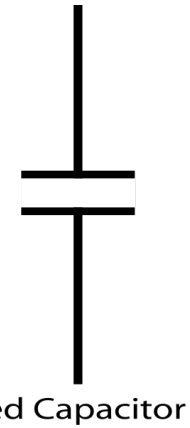
DDR2: Four words transfer

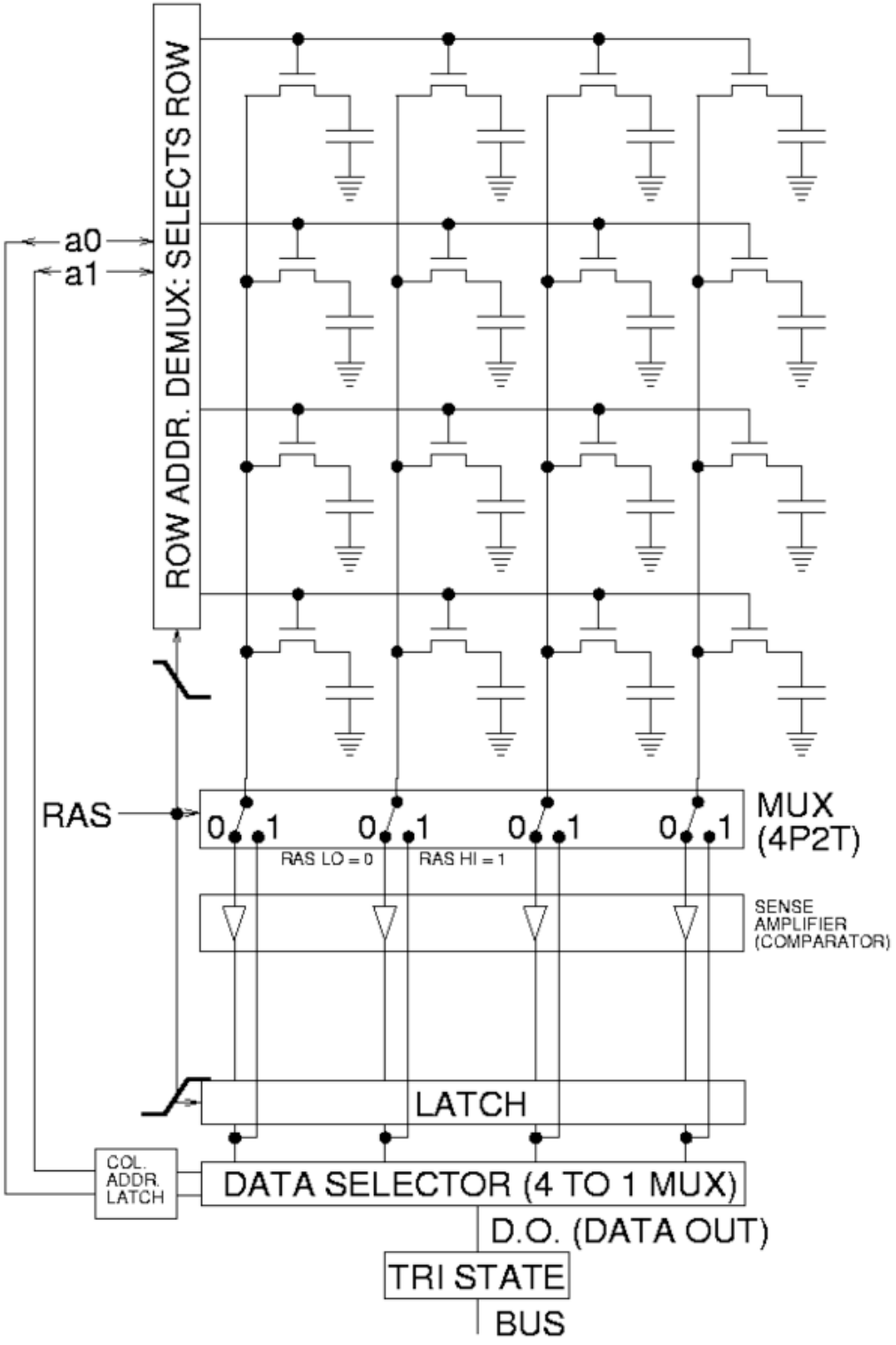
DDR3: Eight words transfer

DDR4: 2 eight words transfer interleaved

GDDR5: Graphics variant.

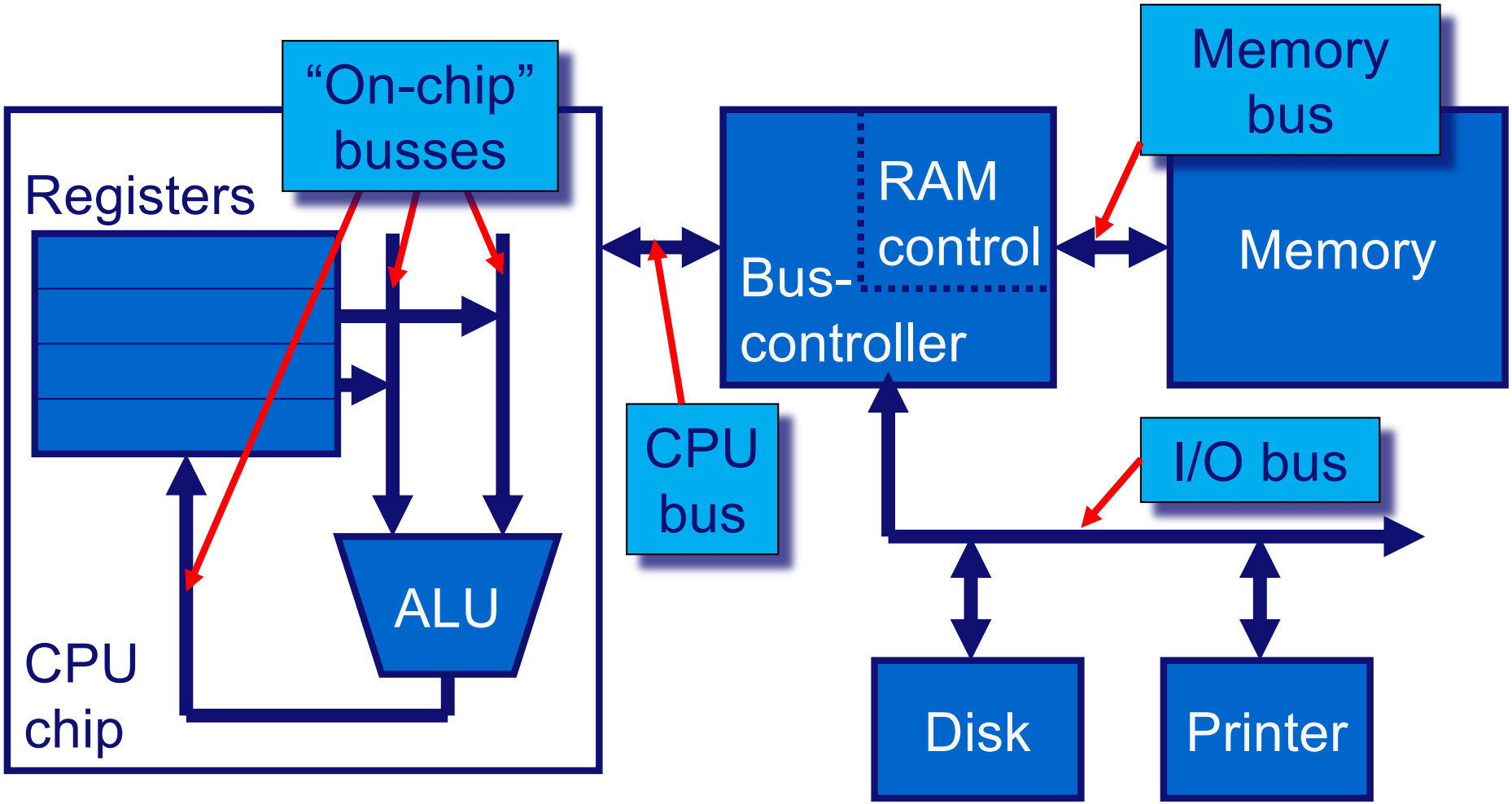






Questions?

Busses inside and outside the CPU



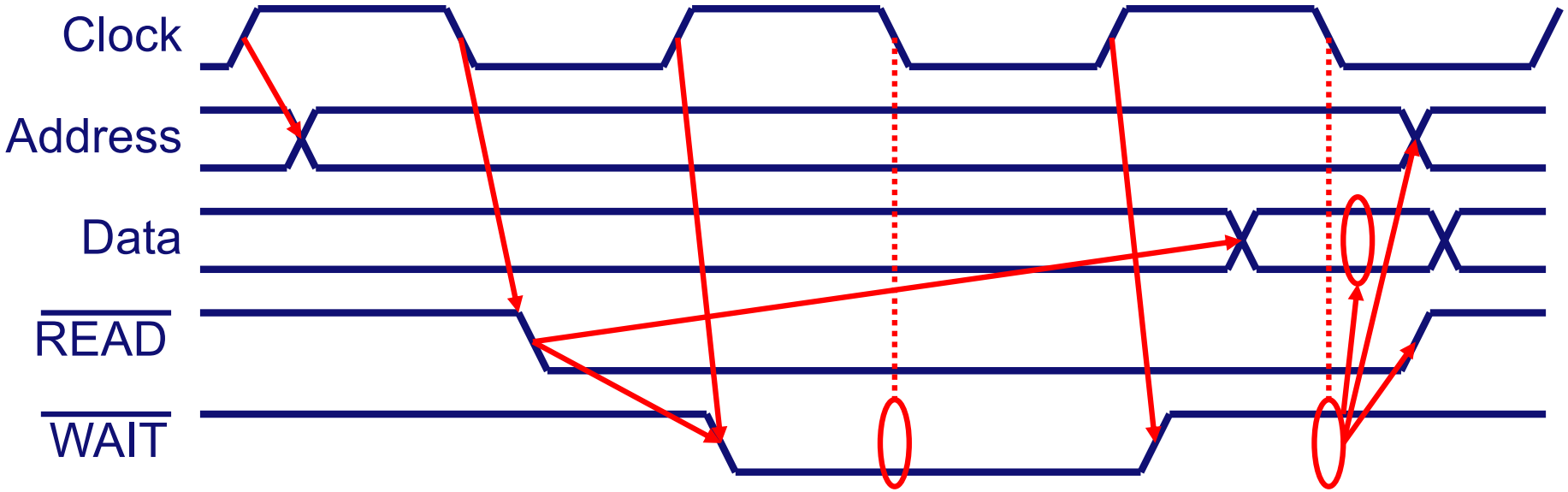
Masters and slaves: who's the boss ?

- Users of a bus have different roles
  - Master initiates bus accesses
  - Slave is passive and waits for masters
  - Combinations exist, but only one at a time !

Master	Slave	Example
CPU	Memory	Fetch instructions
CPU	I/O	Read status
CPU	Co-processor	Write command
I/O	Memory	Direct Memory Access (DMA)
Co-processor	CPU	Fetch operands

Synchronous busses: with “clock”

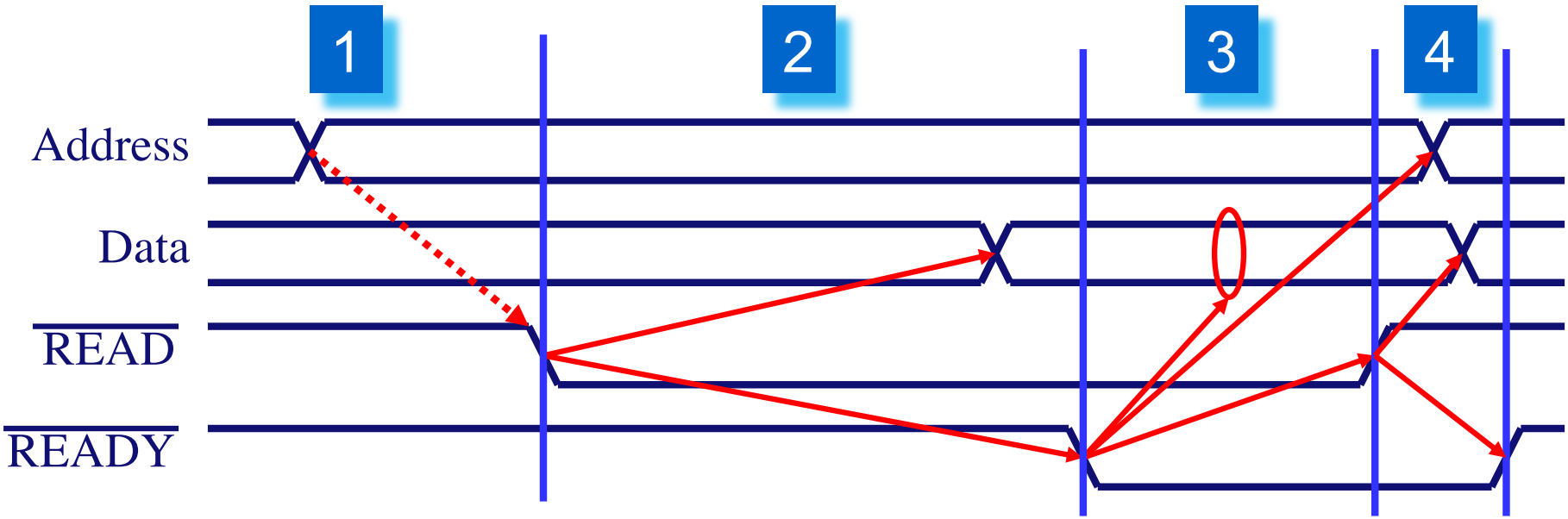
- Bus timing depends on edges of the clock
  - Advantage: everything happens at the same time (synchronous)
  - Drawback: everything takes a multitude of the cycle time



Timing diagram

Asynchronous busses: shaking hands

- *Handshake signals* provide *synchronisation*
  - Advantage: everything happens as soon as possible
  - Drawback: the CPU needs to synchronise with the clock



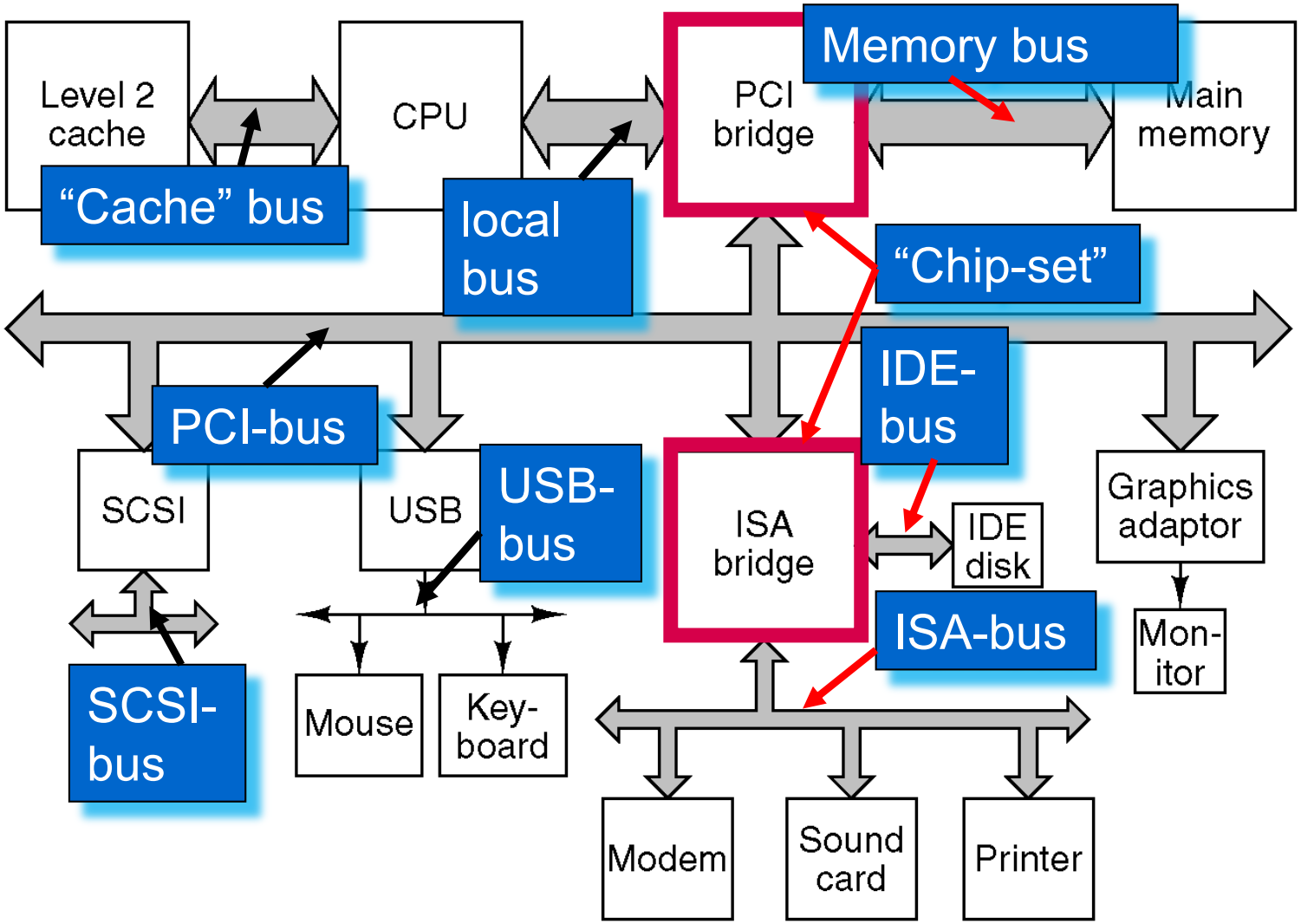
"4 phase handshake"

Bus standards … exist plenty

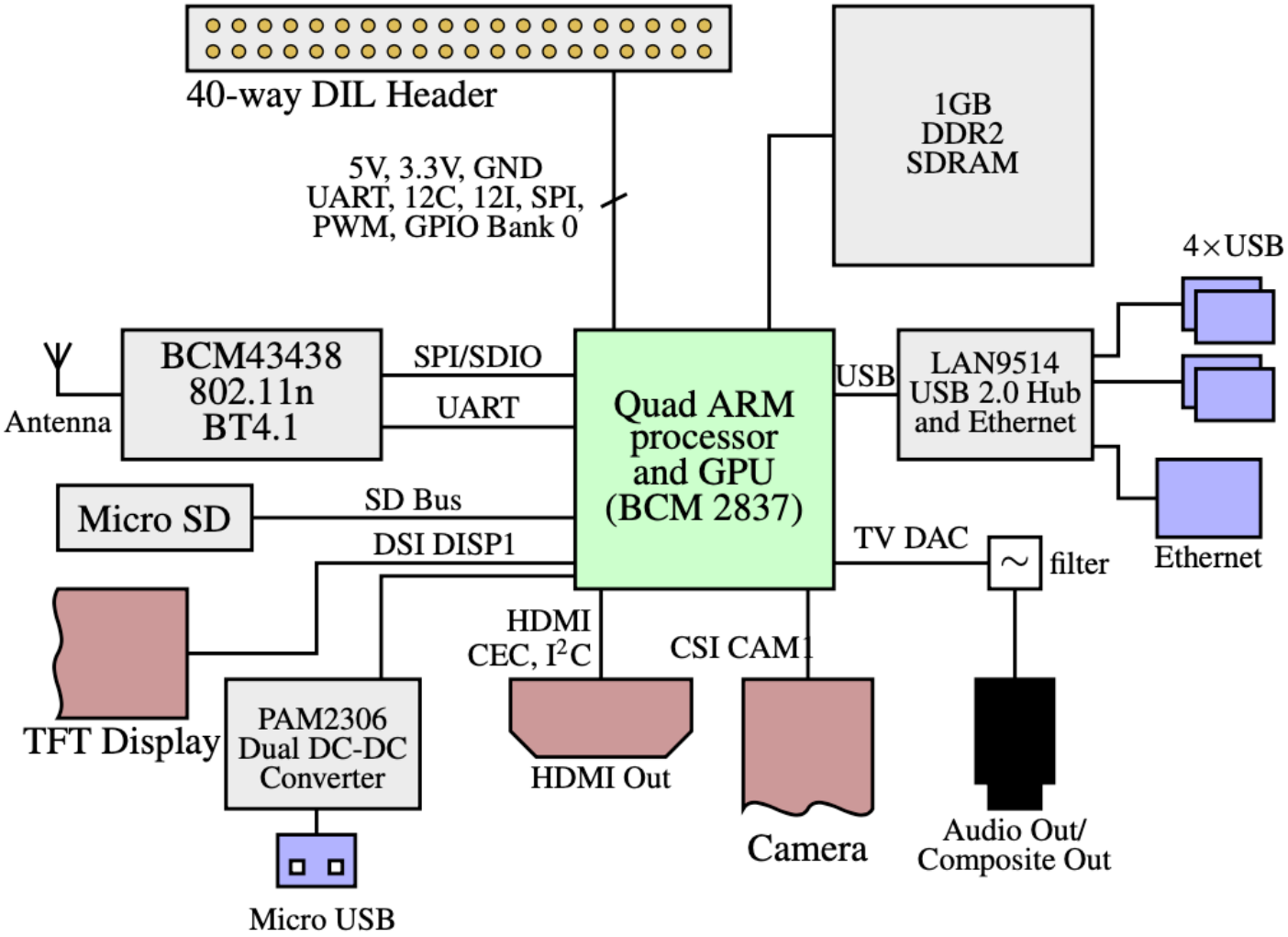
- "System busses": CPU to I/O and memory
  - PC world is a mess: ISA, EISA, Microchannel, PCI
  - Outside is a mess too: VME, Nubus, Camac, Multibus

    each of those in different flavours
- I/O busses: connect devices to the computer
  - SCSI, IDE, HPIB (laboratory instruments)
- Nowadays mostly *serial* –one bit at a time– I/O busses
  - Fewer wires and no skew (hence: fast anyway)
  - USB, Firewire, Serial ATA

Many of this in Pentium-class PC

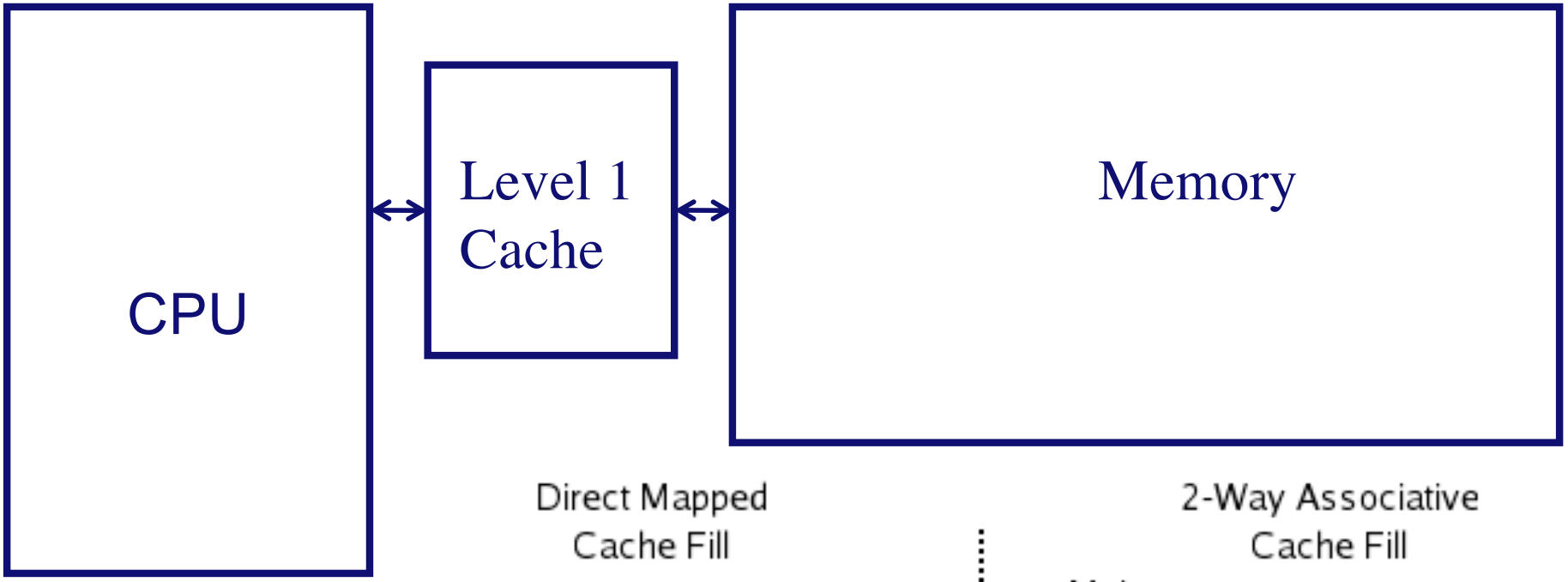


The ARM surrounded by equipment.

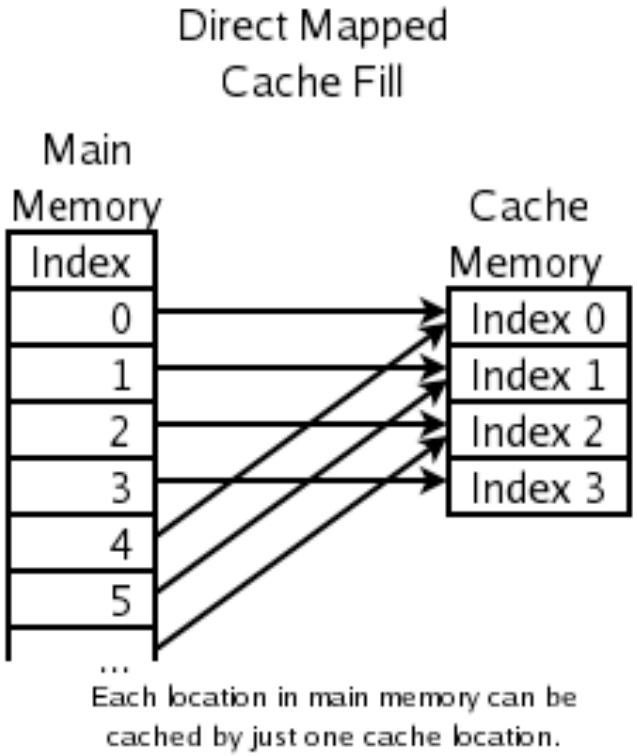


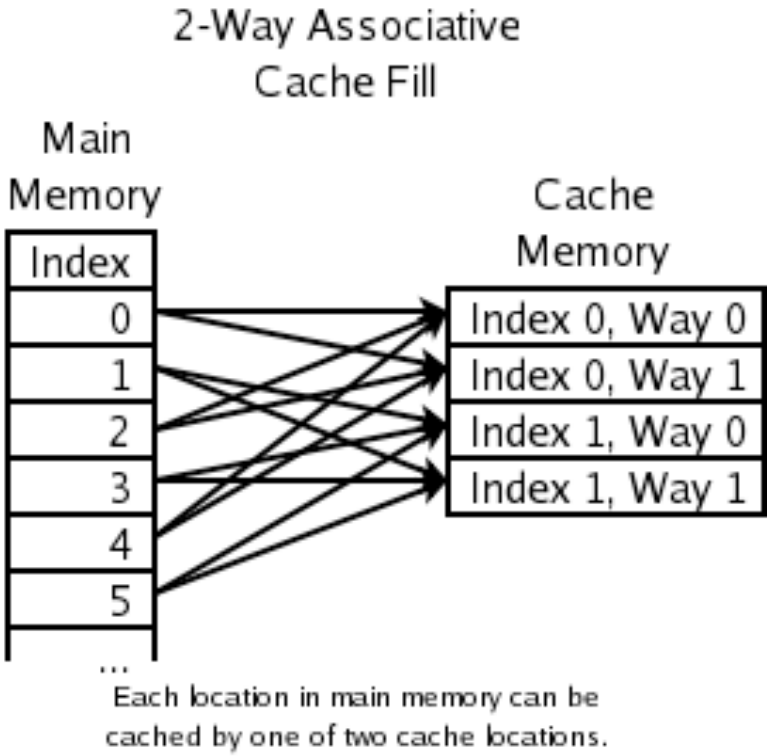
Questions ?

Level 1, 2, etc. memory caches.

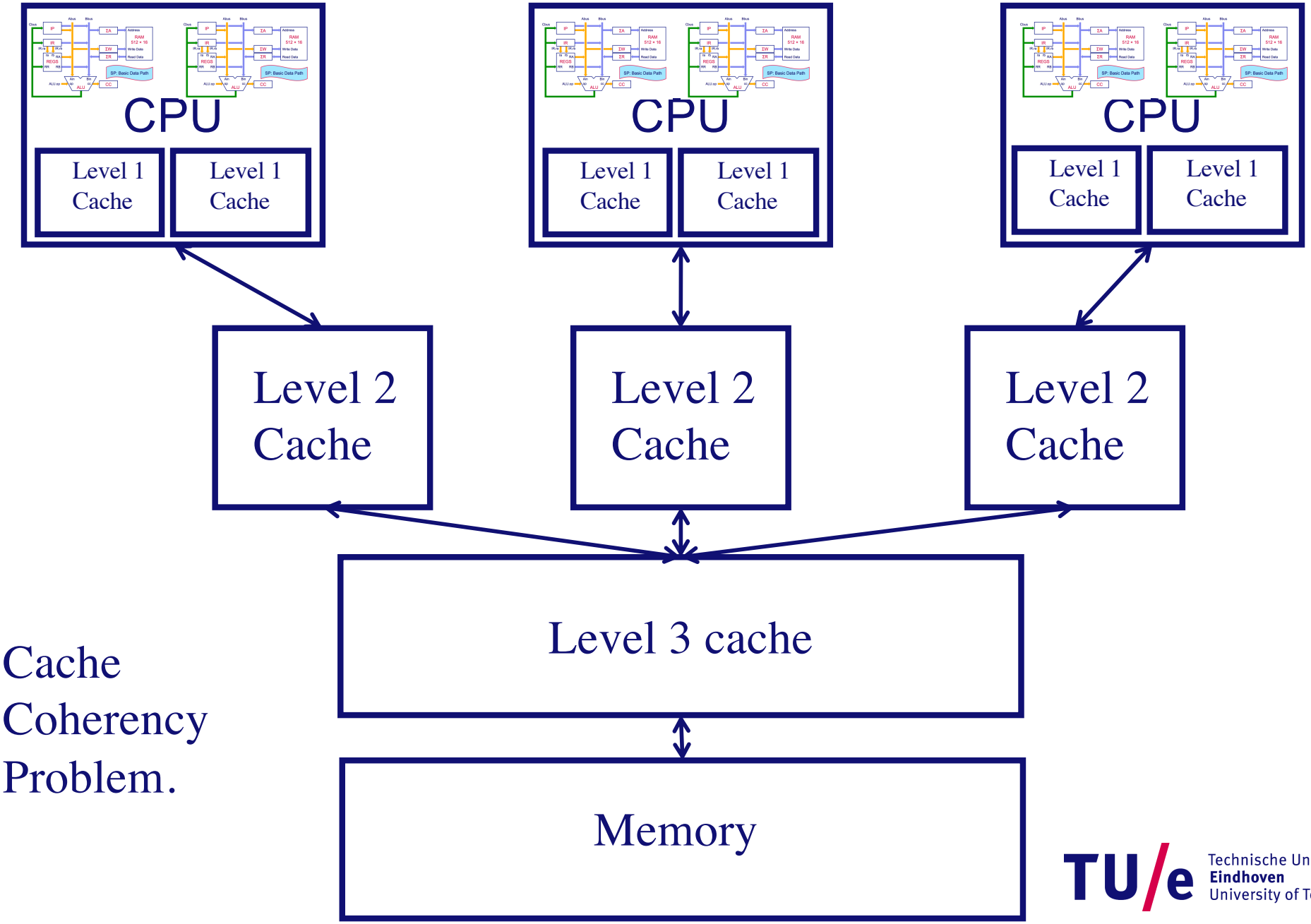


What is a good refresh policy?
First in first out?
Last used first out?



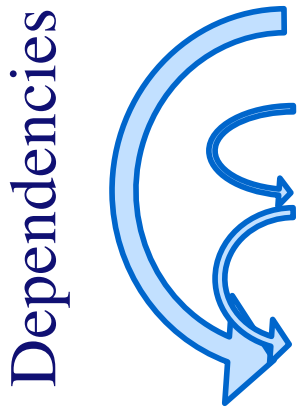


Hyperthreading: solve cache latencies.



Cache Coherency Problem.

Out of order execution.



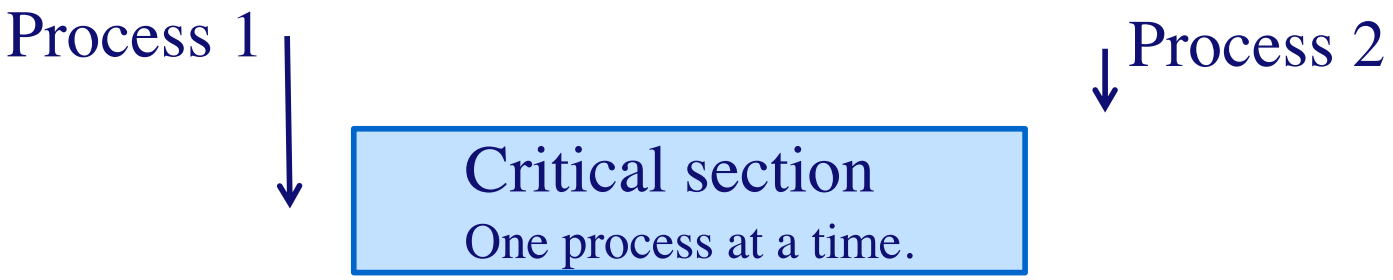

```
LOAD R1 [BP+x]
LOAD R0 1
STOR R0 [BP+y]    ;  We assume x≠y.
ADD R2 1
STOR R2 [BP+R1]
```

Modern processors use out of order execution.
Only sequential dependencies are respected.
Special `barrier` instructions can be used to prevent reordering.

Acquire/release semantics.
Release: all operations before the instruction must be finished first.
Acquire: this operation must be done before all later operations.
E.g. write-release, read acquire.

Out of order execution.

Peterson's mutual exclusion algorithm (from Wikipedia).




```
P0:       flag[0] = true;
P0_gate:  turn = 1;
          while (flag[1] == true && turn == 1)
          {
              // busy wait
          }
          // critical section
          ...
          // end of critical section
          flag[0] = false;
```

```
P1:       flag[1] = true;
P1_gate:  turn = 0;
          while (flag[0] == true && turn == 0)
          {
              // busy wait
          }
          // critical section
          ...
          // end of critical section
          flag[1] = false;
```

Out of order execution.

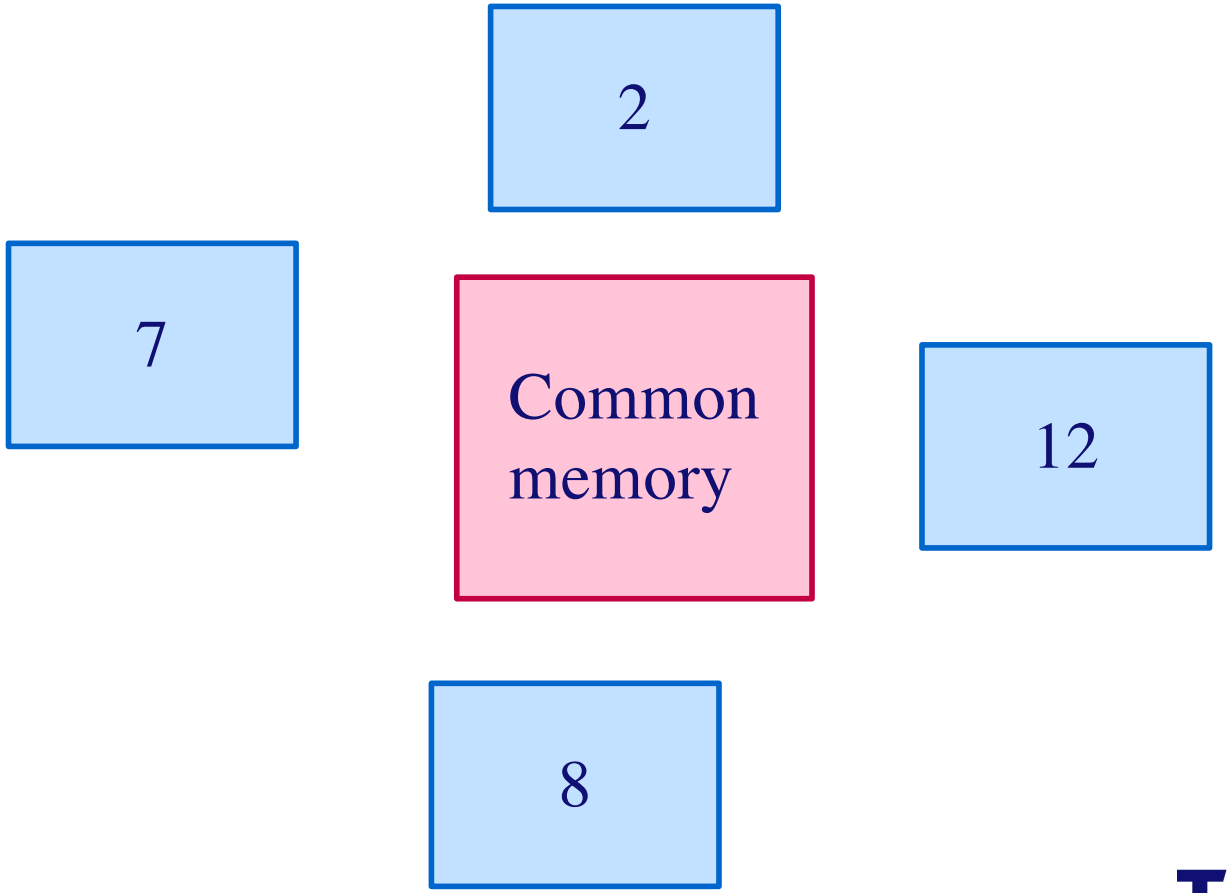
Peterson's mutual exclusion algorithm (from Wikipedia).

```
P0:       flag[0] = true;
P0_gate:  turn = 1;
          while (flag[1] == true && turn == 1)
          {
               // busy wait
          }
          // critical section
          ...
          // end of critical section
          flag[0] = false;
```

Order of these instructions is crucial but they do not have a sequential dependency.
This goes wrong on modern processors.

Consensus problem

Several parties offer a number. Agree on a value offered by one of the parties. If parties are equal and they can only write and read in common memory this is not solvable.



Special instructions.

```
test_and_set [reg]
             if  RAM[reg]==0
             then RAM[reg]:=1

compare_and_swap  reg1  reg2   reg3
             if  RAM[reg1]==reg2
             then RAM[reg1]:=reg3; Z:=1
             else Z:=0
```

Essential for parallel programming.

Multithreading on the ARM

LDREX LOAD EXCLUSIVE

STREX STORE EXCLUSIVE

LDREX R1, [R0] R1:=RAM[R0], own RAM[R0].

STREX R1, R2, [R3] RAM[R3]:=R2

R1:=1
release RAM[R3]
} if RAM[R3] is owned

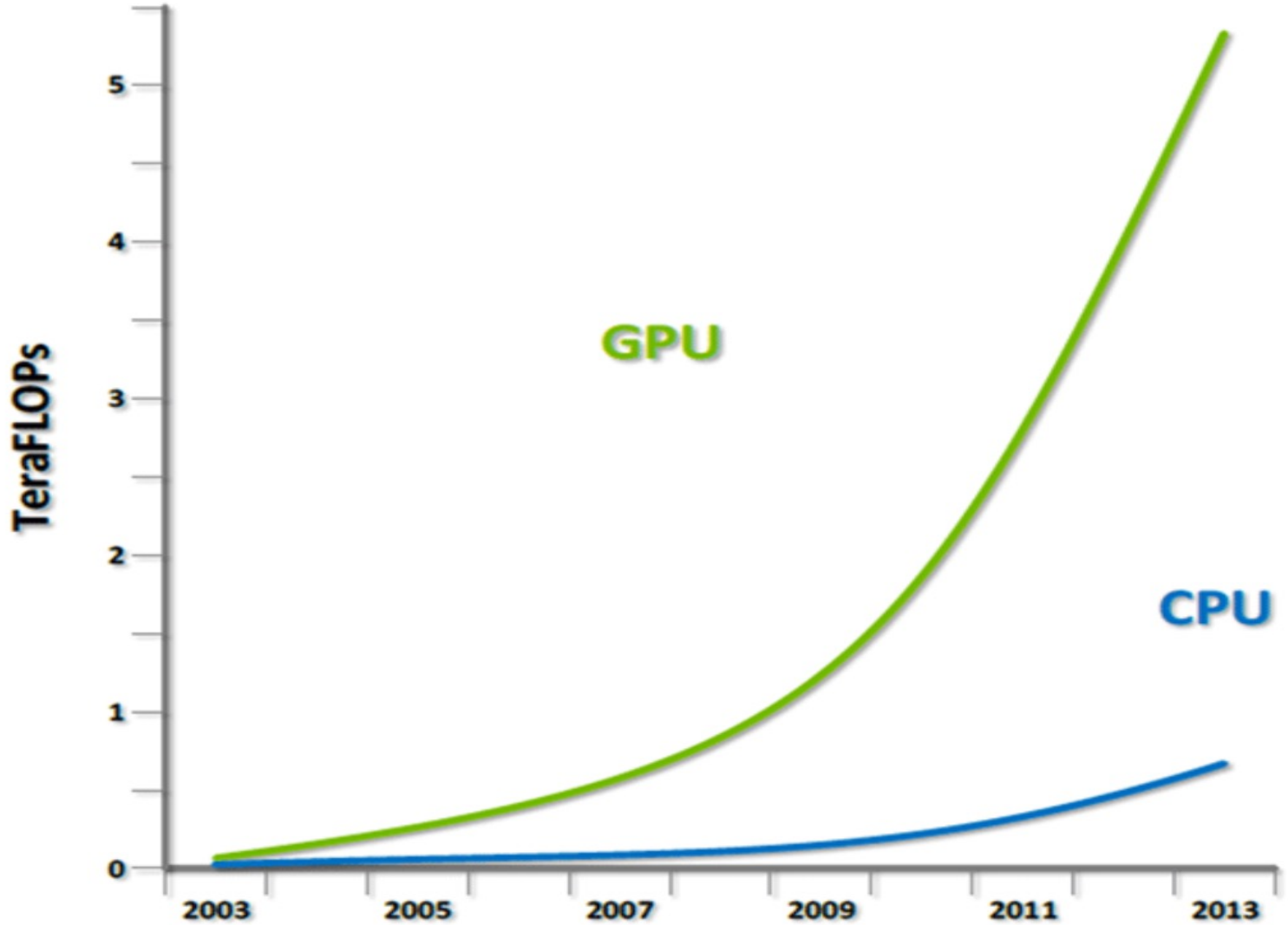
R1:=0 otherwise.

Only one RAM address can be owned per processor.

Questions ?

Graphics processor (GPU).

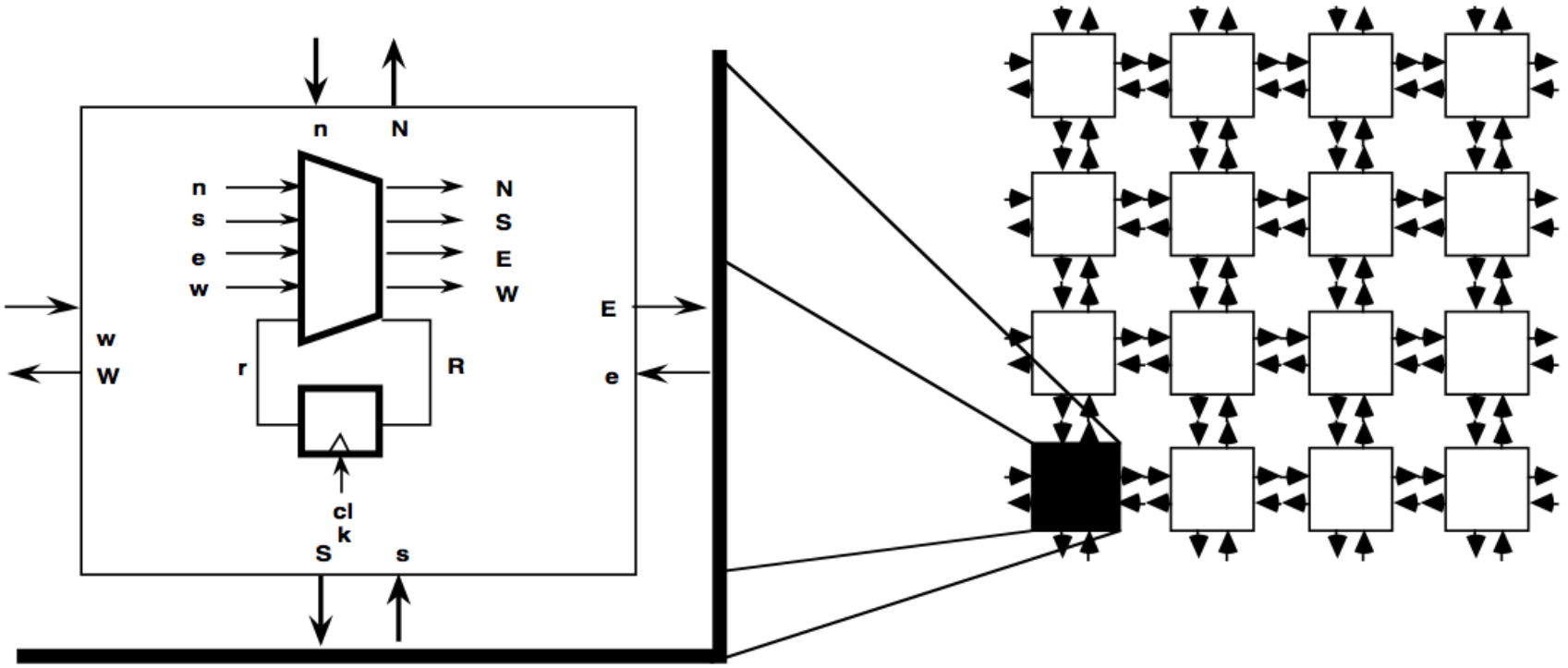
Graphical CPUs outperform CPUs



Graphics processor (GPU)

Figure 1-2. The GPU Devotes More Transistors to Data Processing

Programmable active memory (PAM)



In a PAM a node reacts only to its four neighbours.
Big challenge: how to program such machines?

Questions ?

Summary

- We saw how we can build a compiler, by recursively defining code generation on the basis of a parse tree.
- A computer consists of a complex infrastructure, outside the core processor.
- Dynamic RAM is built with capacitors. Writing and reading data is quite complex and nowadays far slower than the processor.
- Caching, multiple processors all introduce their own issues. Programming becomes increasingly detached from the hardware.
- New computational concepts like GPUs or PAMs may change the architecture of the processors of the future.