Workshop on System-on-Chip Design

Part I : Background - Programming Languages and Runtime Systems

National Chiao Tung University, Taipei, 21.-23.10.2013

Felix Friedrich, ETH Zürich

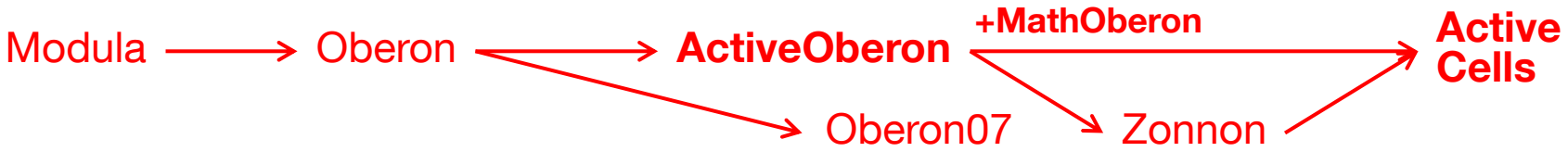
Objectives of Part I

Background information about the tools used in this workshop and labs

- Computing model and systems family
- Programming languages
  Oberon – Active Oberon – Math Oberon
- Loading and Linking
- Compilation

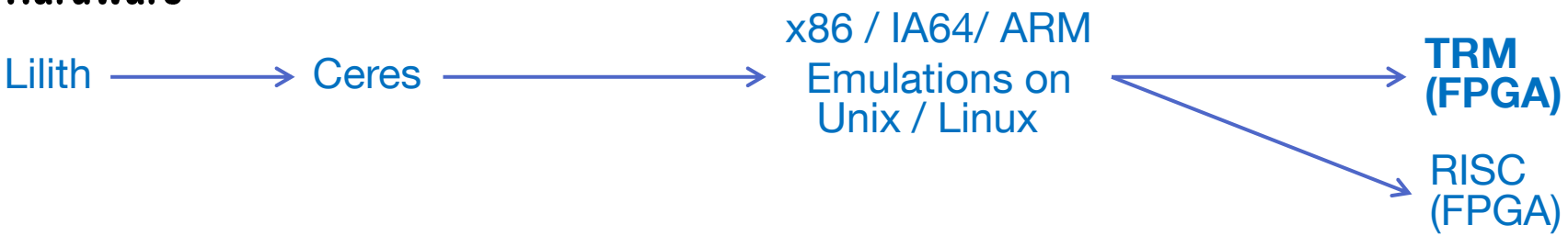
Co-Design @ ETH

Languages (Pascal Family)



Operating / Runtime Systems

Hardware

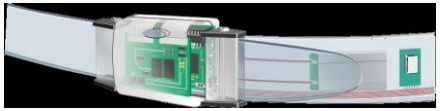


Recent Applications

Industry Controllers

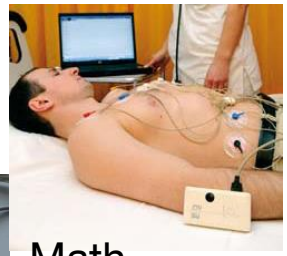
Oberon/A2

Wearable Computing

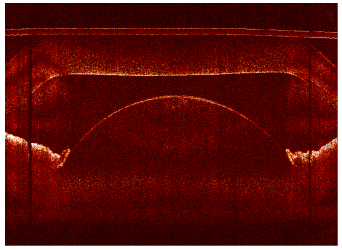


A2

Medical Computing



Math
Oberon



Active Cells +
Minos

Aviation

Minos

Active Cells +
A2

Systems and Purposes

Minos

- Safety-critical monitoring system
- Originally invented for autopilot system for helicopters – still in use and under further development
- *Implementation Highlights*
  - Dynamic remote configurability (dynamic linking)
  - Extremely simple deterministic scheduler, utilizing the run-to-completion semantics

Systems and Purposes

A2 / Active Oberon

- Universal operating system for symmetric multiprocessors (SMP)
- Shared Memory approach
- Currently used as our *development environment*
- Based on the co-design of programming language and OS
- *Implementation Highlights*
  - Simple approach to multi-processing, no heavyweight processes
  - Optimized, fast context switches
  - A lock-free, extremely light weight kernel is being developed.

Systems and Purposes

Active Cells

- Special purpose heterogeneous systems on a chip (SoC)
- Massively parallel hard- and software architecture based on Message Passing
- Software/Hardware co-design: Custom designed computing model for custom designed hardware
- Focus: embedded data-flow applications

Goal

Generic, *high-level*, integrated approach to

construct heterogeneous computing systems

for many purposes with a focus on specialized, embedded systems

Reconfigurable System-On-Chip: one aspect of this in the FPGA domain

THE PROGRAMMING LANGUAGE OBERON

Programming Language Oberon

- Pascal Family
- *Modular* with separate compilation
- *Strongly typed*
  - Static type checking at compile time
  - Runtime (dynamic) support for type guards / tests
- Consequently *high level*
  - Minimal inline assembly in parts of the HAL
  - Specific low level functions in a Pseudo-Module called SYSTEM

Example of a Module

```
MODULE Timer;

IMPORT Kernel, Commands, Streams;

VAR recent: LONGINT; factor: REAL;

PROCEDURE Start*(VAR ticks: LONGINT);
BEGIN ticks := Kernel.GetTicks();
END Start;

(* step timing: update value in ticks *)
PROCEDURE Step*(VAR ticks: LONGINT): REAL;
VAR previous: LONGINT;
BEGIN
   previous := ticks;
   ticks :=  Kernel.GetTicks();
   RETURN (ticks-previous)*factor
END Step;
```

Module Name

Imported Modules

Global Variables

Comments are embraced with (* and *)

module continues

Example of a Module

continued

```
(* Parameterless procedure is a command *)
PROCEDURE Tick*;
BEGIN Start(recent);
END Tick;

PROCEDURE Tock*(context: Commands.Context);
VAR out: Streams.Writer;
BEGIN
   out:= context.out;
   out.String("elapsed seconds ");
   out.Float(Step(recent),20);
   out.Ln;
END Tock;

PROCEDURE Calibrate; BEGIN (* *)
END Calibrate;

BEGIN Calibrate();
END Timer.
```

Commands are parameterless procedures or procedures with a special context parameter

An asterisk after symbol name makes it exported

A module can have a body that is executed once at module load time

Control Structures

- **IF**

```
IF a = 0 THEN
        (* statement sequence *)
END
```

- **WHILE**

```
WHILE x<n DO
        (* statement sequence *)
END
```

- **REPEAT**

```
REPEAT
        (* statement sequence *)
UNTIL x=n;
```

- **FOR**

```
FOR i := 0 TO 100 DO
        (* statement seq *)
END;
```

Builtin Types

- **BOOLEAN**
  ```
  b := TRUE; IF b THEN END;
  ```
- **CHAR**
  ```
  c := 'a'; c := 0AX;
  ```
- **SHORTINT ⊂ INTEGER ⊂ LONGINT ⊂ HUGEINT**
  ```
  i := SHORT(s); l := 10; h := 010H;
  ```
- **REAL LONGREAL**
  ```
  r := 1.0; r := 10E0; d := 1.0D2;
  ```
- **SET**
  ```
  s := {1,2,3}; s := s + {5}: s := s -
  {5}; s := s * {1..6};
  ```
- **ADDRESS, SIZE**

Structured Types

```
TYPE

Device* = POINTER TO RECORD
   id*: INTEGER;
   next*: Device;
END;

Handler* = PROCEDURE(type,adr,fp: LONGINT;VAR res: LONGINT );

NumberType*= REAL;

DeviceName* = ARRAY DeviceNameLength OF CHAR;

Data*= POINTER TO ARRAY OF CHAR;
```

Records are Data Containers

Procedure Type

Type Alias

Array

Dynamic Array

Builtin Functions

- **Increment and decrement**
  ```
  INC(x); DEC(x); INC(x,n); DEC(x,n);
  ```
- **Sets**
  ```
  INCL(set, element); EXCL(set, element);
  ```
- **Assert and Halt**
  ```
  ASSERT(b<0); HALT(100);
  ```
- **Allocation**
  ```
  NEW(x, …);
  ```
- **Shifts**
  ```
  ASH(x,y); LSH(x,y); ROT(x,y);
  ```
- **Conversion**
  ```
  SHORT(x); LONG(x); ORD(ch); CHR(i); ENTIER(r);
  ```
- **Arrays**
  ```
  LEN(x); LEN(x,y); DIM(t);
  ```
- **Misc**
  ```
  ABS(x); MAX(type); MIN(type); ODD(i);
  ```

System Programming with Oberon

Pseudo-Module SYSTEM provides low-level access

- SYSTEM.PUT, SYSTEM.GET and SYSTEM.BIT to write to and read from memory

```
PROCEDURE Send*(portAdr: LONGINT; data: LONGINT);
BEGIN
     SYSTEM.PUT(portAdr,data);
END Send;

PROCEDURE Receive*(portAdr: LONGINT; VAR data: INTEGER);
BEGIN
     REPEAT UNTIL ~SYSTEM.BIT(portAdr+1,0);
     SYSTEM.GET(portAdr,data);
END Receive;
```

Code from TRM.Channels.Mod, module controlling communication of processors on FPGA

→ more later

System Programming with Oberon

- `SYSTEM.VAL` for unsafe, unchecked type-cast

```
PROCEDURE ConvertRI*(l: REAL): LONGINT; (* Floor 32bit *)
VAR x,xe, n, sign: LONGINT;
BEGIN
     x := SYSTEM.VAL(LONGINT,l);
     …
```

- Also useful for system programming:
  high level type `SET` of Oberon used as bit field

```
PROCEDURE Show*(val: SET);
BEGIN
     SYSTEM.PUT(LEDAdr, val); (* val bits indicate lit LEDs*)
END Show;
…
Show({1..3,5}); (* bits 1..3 and 5 set *)
```

But…

… did I not say high level?

Yes:

preceding builtins `SYSTEM.PUT`, `SYSTEM.GET`, `SYSTEM.VAL` are necessary and sufficient for all *system programming* tasks of this workshop.

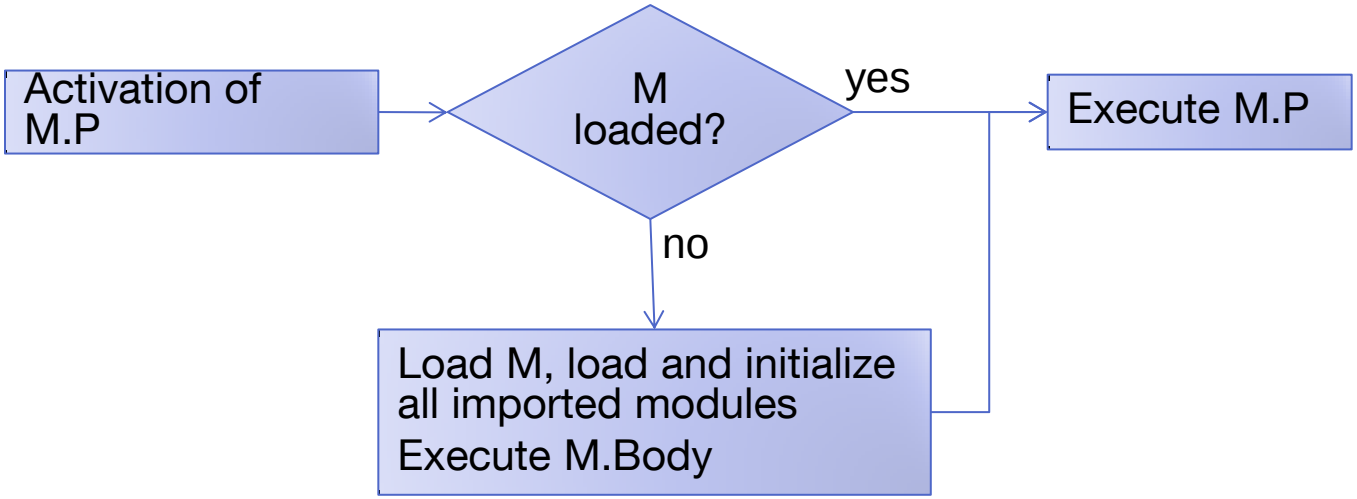
Example: write to memory mapped device

Where are the programs?

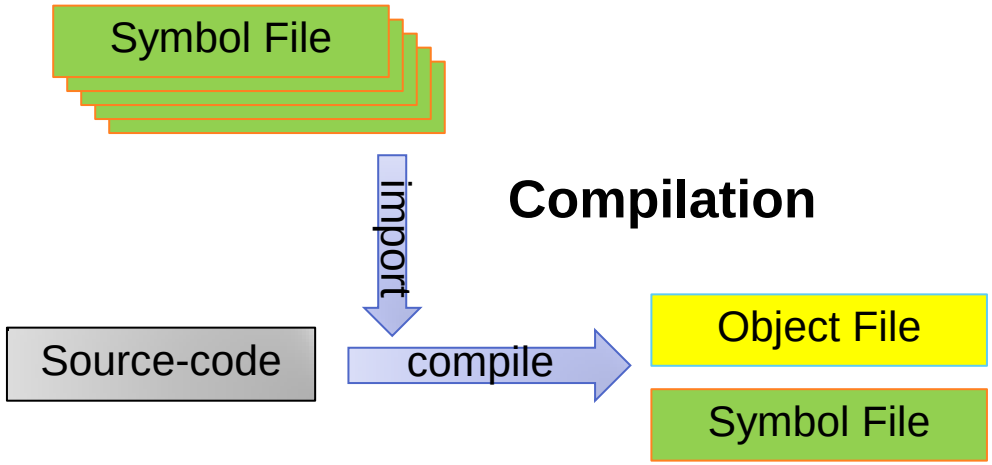
- There is no «program» in Oberon.
- There are modules. Modules can contain commands. Commands can be called.
- Modules can be statically linked to form a kernel (or executable if embedded in other OS)
- Modules can be dynamically linked

Commands and Module Loading

- Modules are loaded on demand
- Statically linked modules are loaded at system-startup
- Exported Procedures without parameters can act as commands
- A modification of a compiled module becomes effective only after (re-) loading the module
- A module M can be unloaded only if no currently loaded module imports M and if M is not statically linked to the Kernel

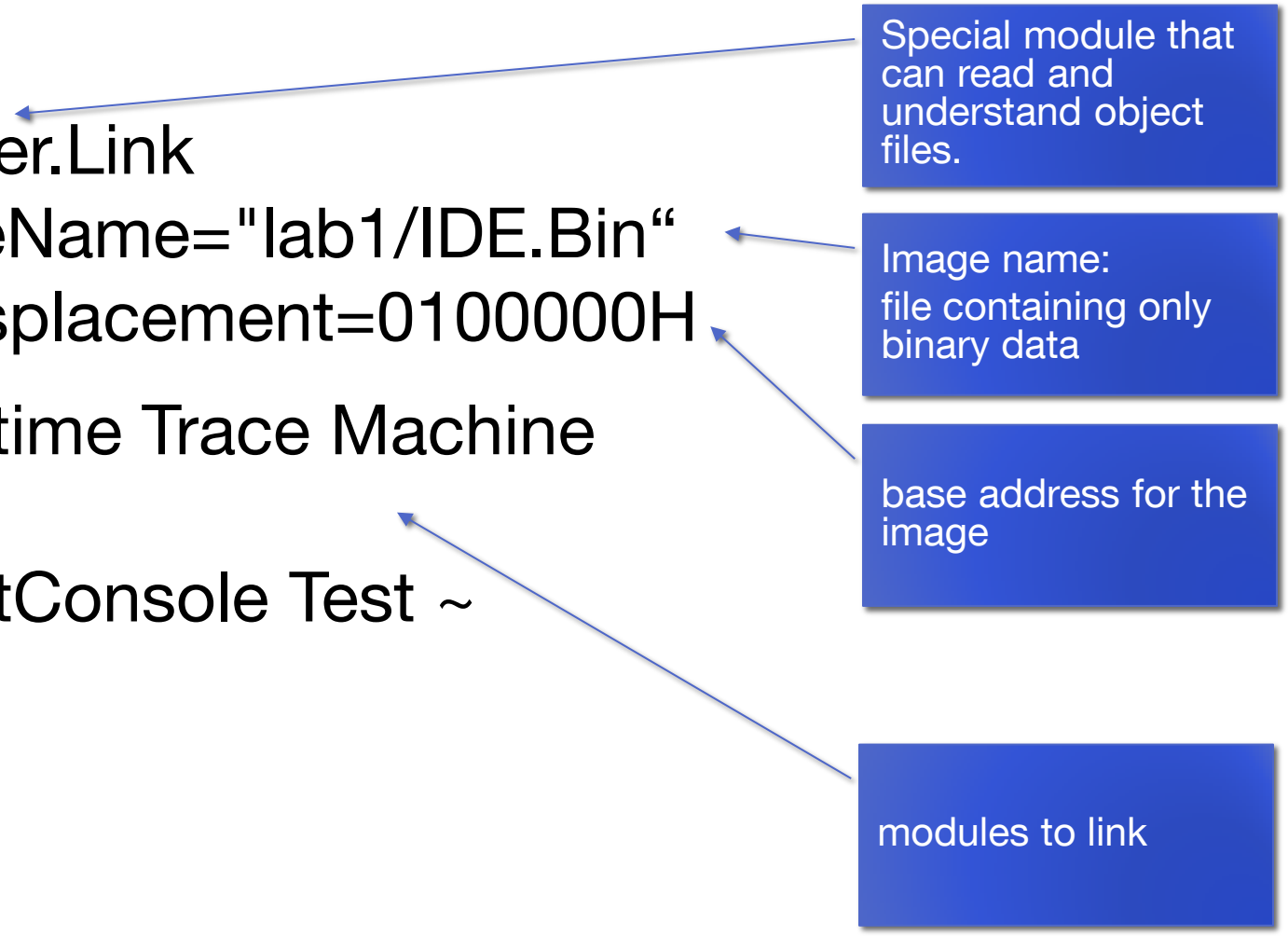


Symbol Files and Object Files



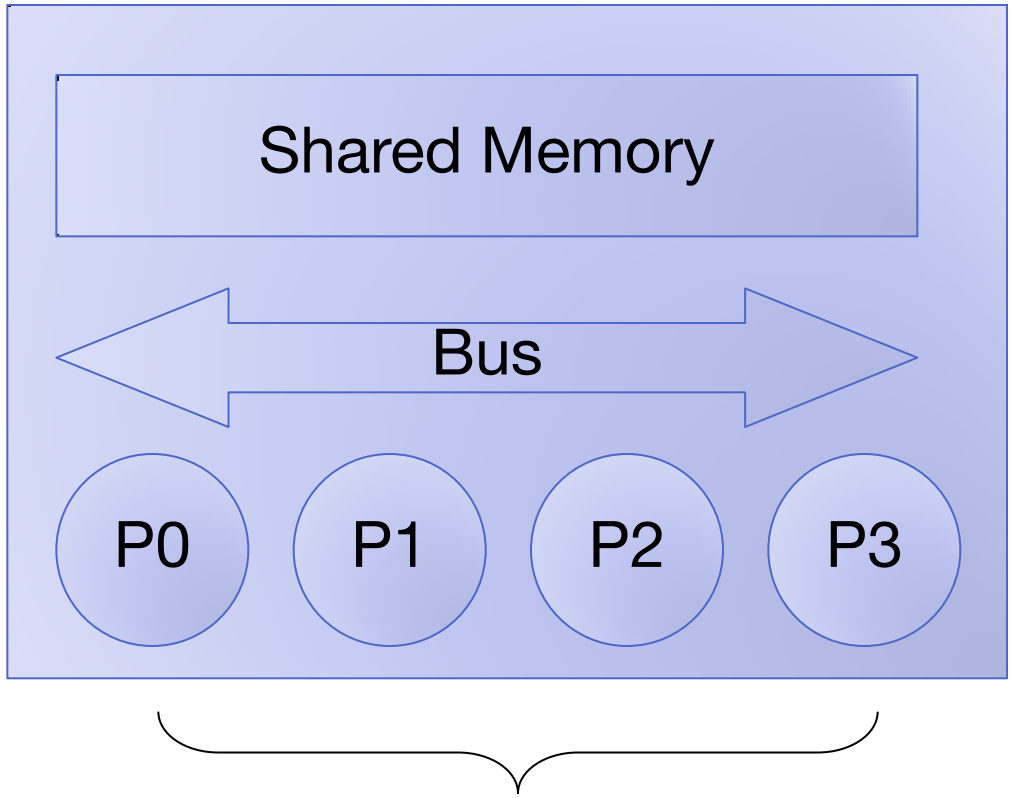
Linking a Kernel

```
StaticLinker.Link
        --fileName="lab1/IDE.Bin“
        --displacement=0100000H
        Runtime Trace Machine
        ….
        BootConsole Test ~
```



ACTIVE OBERON

Target Architecture

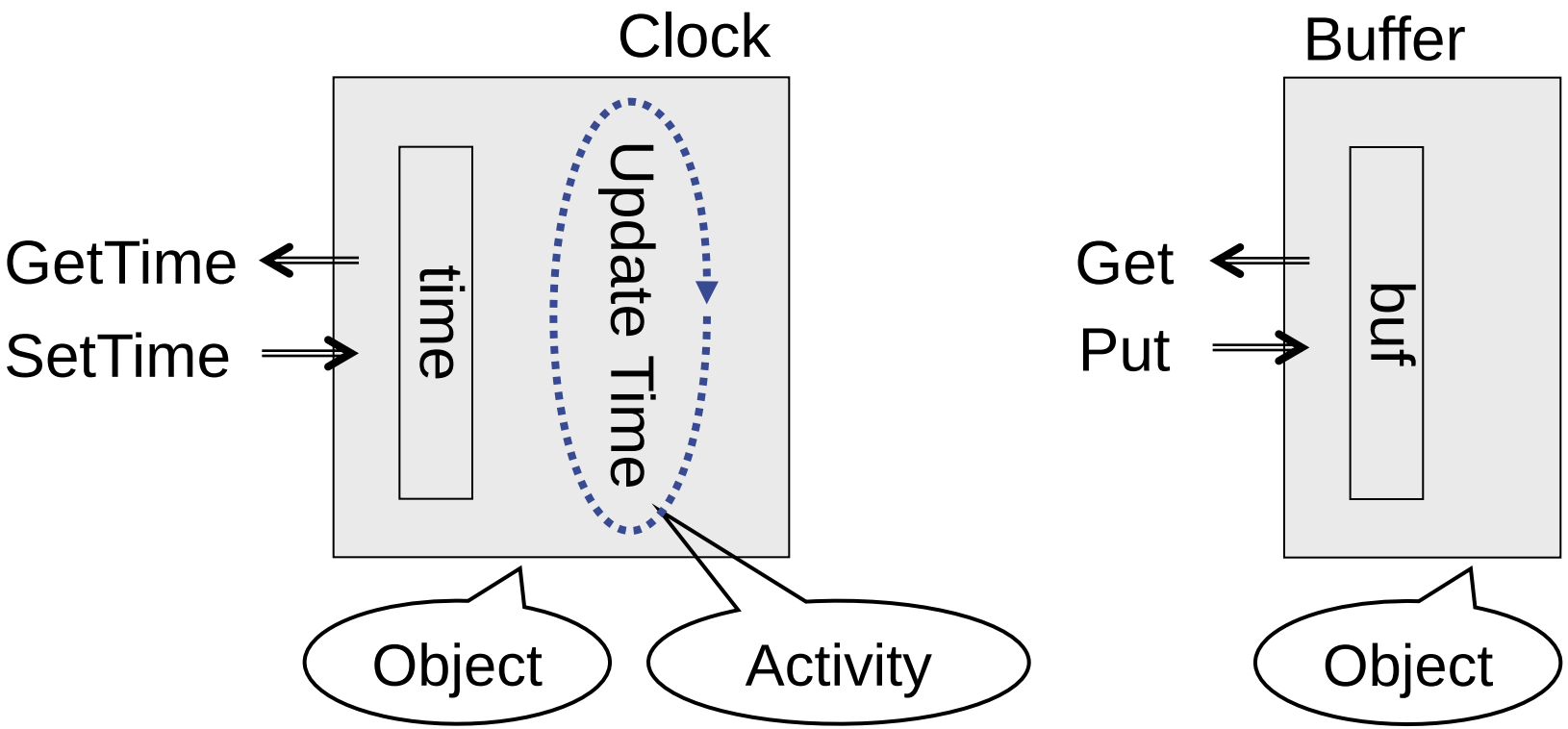


Symmetrical Multiple Processors (SMP)

Computing Model

- Participating entities
  - Interoperating «active» objects with encapsulated behavior
- Synchronization
  - Object as monitors:
    mutual exclusion within object contexts
  - Synchronisation:
    Wait for object-local condition

Active vs. Passive Objects



Synchronisation and Protection

```
TYPE
   MyObject = OBJECT
     VAR i: INTEGER; x: X; (* state space *)

     PROCEDURE & Init (a, b: X);
     BEGIN... (* initialization *) END Init;

     PROCEDURE f (a, b: X): X;
     BEGIN{EXCLUSIVE}

        ...
       AWAIT i >= 10;

        ...
     END f;

     PROCEDURE g();
     BEGIN
        ...
        BEGIN {EXCLUSIVE}
         INC(i)
        END;
        ...
     END g;

   END MyObject;
```

Object in the OO sense: procedures are methods

Protection: blocks tagged «exclusive» run under mutual exclusion

Synchronisation: wait for i>= to hold

fine grained protection:
blocks can be tagged exclusive

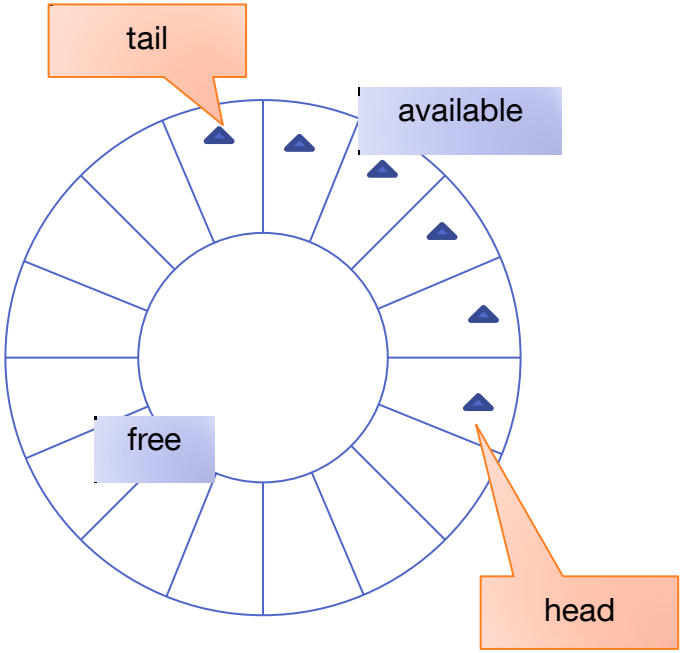
Semantics

- Only one thread can enter the monitor at a time
- Threads that run into a non-fulfilled `AWAIT` temporarily exit the monitor
- *Conditions are re-evaluated when a thread exits the monitor*
- Threads awaiting a condition have higher priority than threads awaiting the lock

The `await` Construct

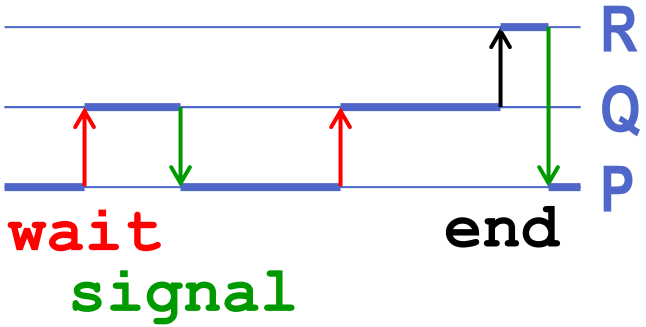
- Finite Buffer as a shared resource

```
VAR head, tail, available, free: INTEGER;
    buf: ARRAY N of object; (* circular *)

  PROCEDURE Produce (x: object);
  BEGIN{EXCLUSIVE}
    AWAIT(free # 0);
    DEC(free); buf[tail] := x;
    tail := (tail + 1) mod N;
    INC(available);
  END Produce;

  PROCEDURE Consume (): object;
    VAR x: object;
  BEGIN{EXCLUSIVE}
    AWAIT(available # 0);
    DEC(available); x := buf[head];
    head := (head + 1) MOD N;
    INC(free); RETURN x
  END Consume;
```



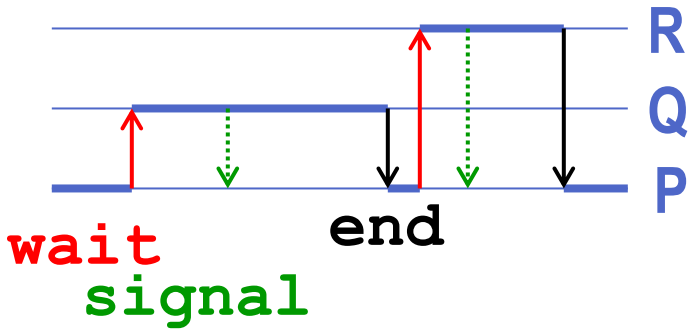
`await` Implementation

"Signal-And-Wait"



[Java, C#]

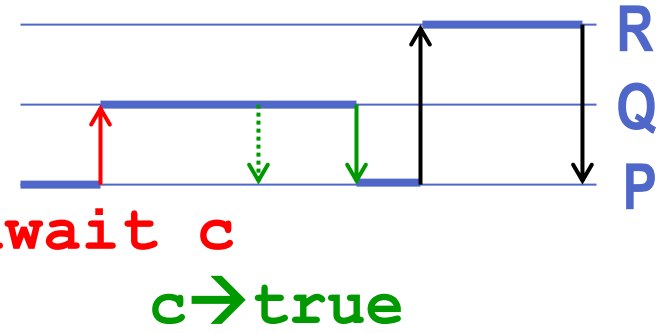
"Signal-And-Continue"



no context switch

Active Oberon:

(await queues have priority*)



Activities

```
TYPE
   MyActiveObject = OBJECT
      ...

   BEGIN{ACTIVE}
      ...
      g();
      ...
   END MyActiveObject;

   ...

   VAR o: MyActiveObject;
   BEGIN
        NEW(o);
        ...
```

Parallelism:
active body of an object - behavior as an encapsulated thread

Together with the memory for o, a new thread is created that executes the body of o:

- allocate memory
- call constructor
- create thread
- publish object

Example: Idle Thread

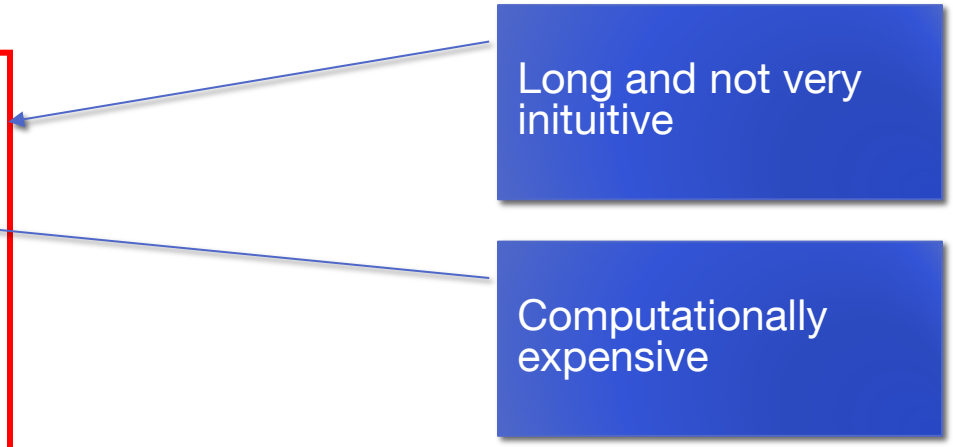
```
Idle = OBJECT
BEGIN {ACTIVE, SAFE, PRIORITY(0)}
  LOOP
    REPEAT
      ProcessorHLT
    UNTIL maxReady >= lowestAllowedPriority;
    Yield
  END
END Idle;
```

MATH OBERON

Example

Part of Singular Value Decomposition algorithm

```
var u: pointer to array of array of real ;
s,h,f: real ; i,j,k,l,m,n: integer;
(* ... *)
for j := l to n do
    s := 0.0;
    for k := i to m do
            s := s + u[k, i] * u[k, j]
    end;
    f := s / h;
    for k := i to m do
            u[k, j] := u[k, j] + f * u[k, i]
    end
end;
```



Same Example using Math Oberon

Part of Singular Value Decomposition algorithm

```
var u: array [*, *] of real ;
s,h,f: real ; i,j,k,l,m,n: integer;
(*  ... *)
for j := l to n do

    s := u[i..m,i] +* u [i..m,j];
    u[i..m,j] := u[i..m,j] + s/h * u[i..m,i];

end;
```

+* denotes scalar product

Matlab-like notation

Shorter, more readable

Optimized algorithms behind the operators

Array sub-domain

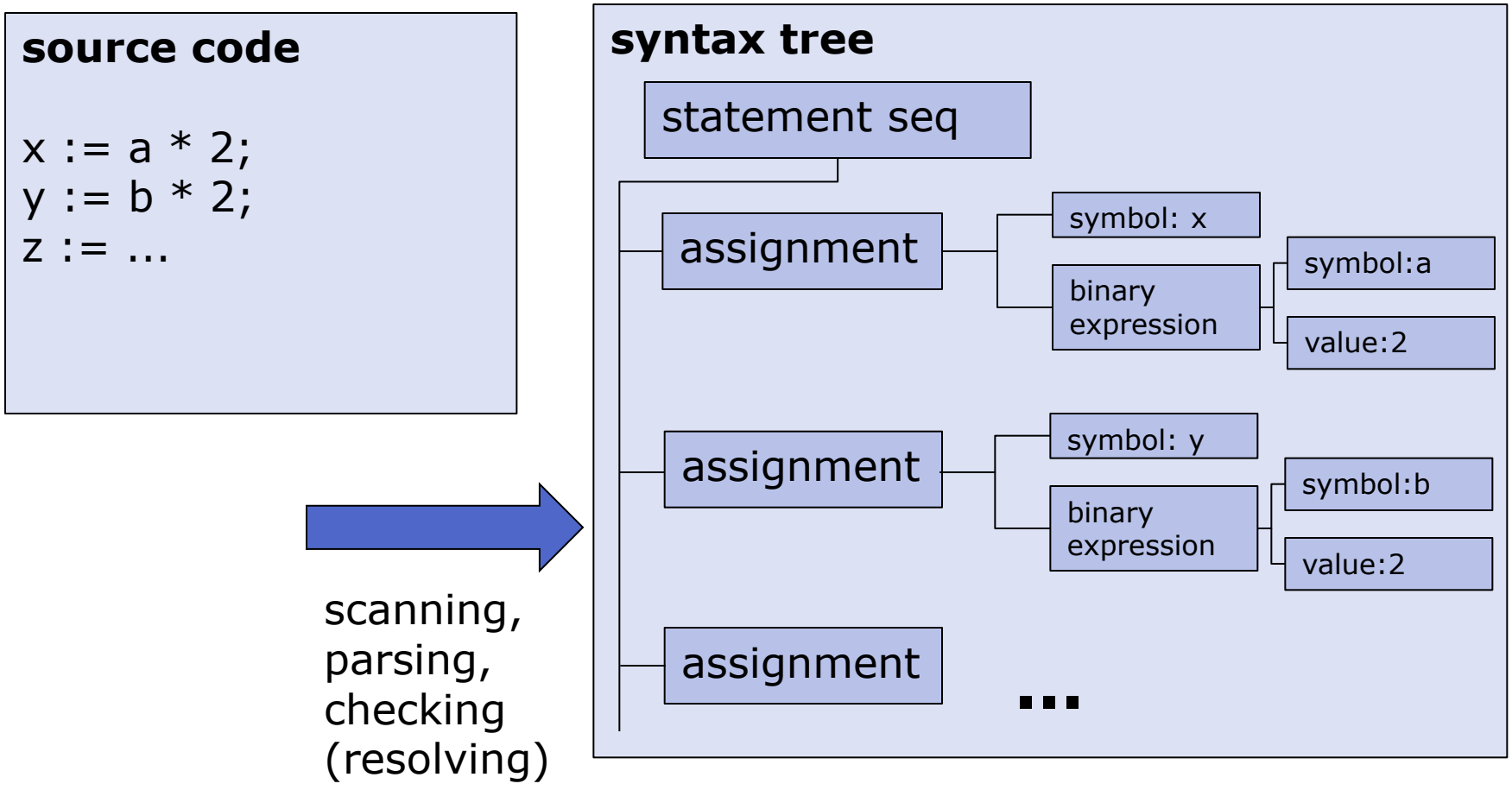
Objectives of Math Oberon

- Abstract formulation on a high-level
  - built-in operators (Linear Algebra)
  - array domains
  - tensors
- Performance
  - operators and access patterns optimized in runtime library
    - vectorized
    - multi-core
    - cache-aware

COMPILATION, OBJECT FILES AND LINKING

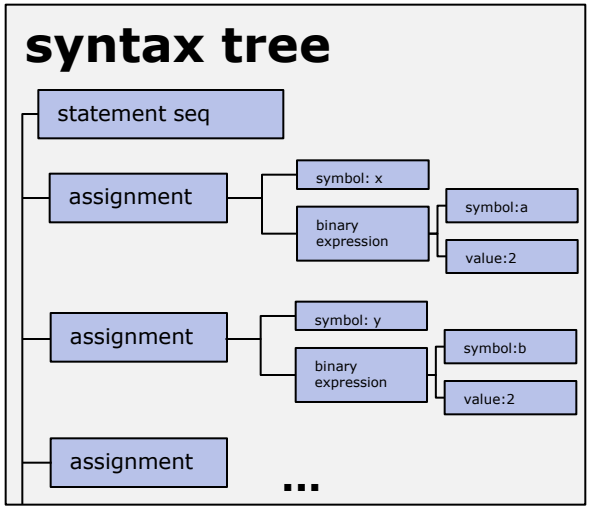
The Compilation Process

- Syntax Tree Generation



The Compilation Process

- Intermediate Code Generation



syntax tree traversal

intermediate code

```
.module A
 ....

.code A.P
  ....
  1:   mul s32 [A.x:0], [A.a:0], 2
  2:   mul s32 [A.y:0], [A.y:0], 2
   ....
```

The Compilation Process

- Binary Code Generation

intermediate code

```
.module A
 ....

.code A.P
  ....
  1:  mul s32 [A.x:0], [A.a:0], 2
  2:  mul s32 [A.y:0], [A.y:0], 2
   ....
```

IR code traversal

binary code

```
fixup 2 <-- A.a (displ=0) [3+7 abs]
fixup 6 <-- A.x (displ=0) [3+7 abs]
fixup 7 <-- A.b (displ=0) [3+7 abs]
...
[  2] 30807 ; LD R1, [0]
[  3] 00401 ; MOV R0, R1
[  4] 2801F ; ROR R0, 31
[  5] 14001 ; BIC R0, 1
[  6] 34007 ; ST R0, [0]
....
```

Fixups must be patched by linker

Symbolic References in the IR

Source Code / Interface (Symbol File)

```
MODULE A;
VAR a*: LONGINT;

PROCEDURE P*;
BEGIN a := a*10 END P;
END A.
```

```
MODULE A;
    VAR a*: LONGINT;
    PROCEDURE P*;
```

```
MODULE B;
IMPORT A;
BEGIN A.a := 1; A.P() END
B.
```

```
MODULE B;
    IMPORT A;
```

Intermediate Code

```
.module A
.const A.@moduleSelf offset=0
  0:  data u32 0
.var A.a offset=-4
  0:  reserve 4
.code A.P offset=0
  0:  enter 0, 0
  1:  mul s32 [A.a:0], [A.a:0], 10
  2:  leave 0
  3:  return 0
```

```
.module B
.imports A
.const B.@moduleSelf offset=0
    0:  data u32 0
.code B.$$Body offset=0
  0:  enter 0, 0
  1:  mov s32 [A.a:0], 1
  2:  call u32 A.P:0, 0
  3:  leave 0
  4:  return 0
```

Fixup Locations in Object File

Object File*

```
const A.@moduleSelf 651605535 8 aligned 4 0 4
    00000000
data A.a 471859334 8 aligned 4 0 4
    00000000
code A.P 2046820492 8 aligned 0 2 24
    abs 12 A.a 471859334 0 0 1 0 32
    abs 18 A.a 471859334 0 0 1 0 32
    8C0000008BA0000000F0FA50000000009850000000009C3C
code A.$$BODY -278328333 8 aligned 0 0 6
    8C0000009C3C
```

Fingerprint

consistency

```
const B.@moduleSelf 651605535 8 aligned 4 0 4
    00000000
code B.$$Body -345437455 8 aligned 0 2 21
    abs 6 A.a 471859334 0 0 1 0 32
    rel 15 A.P 2046820492 -4 0 1 0 32
    8C0000007C5000000000100000008EDEFFFFFF9C3C
```

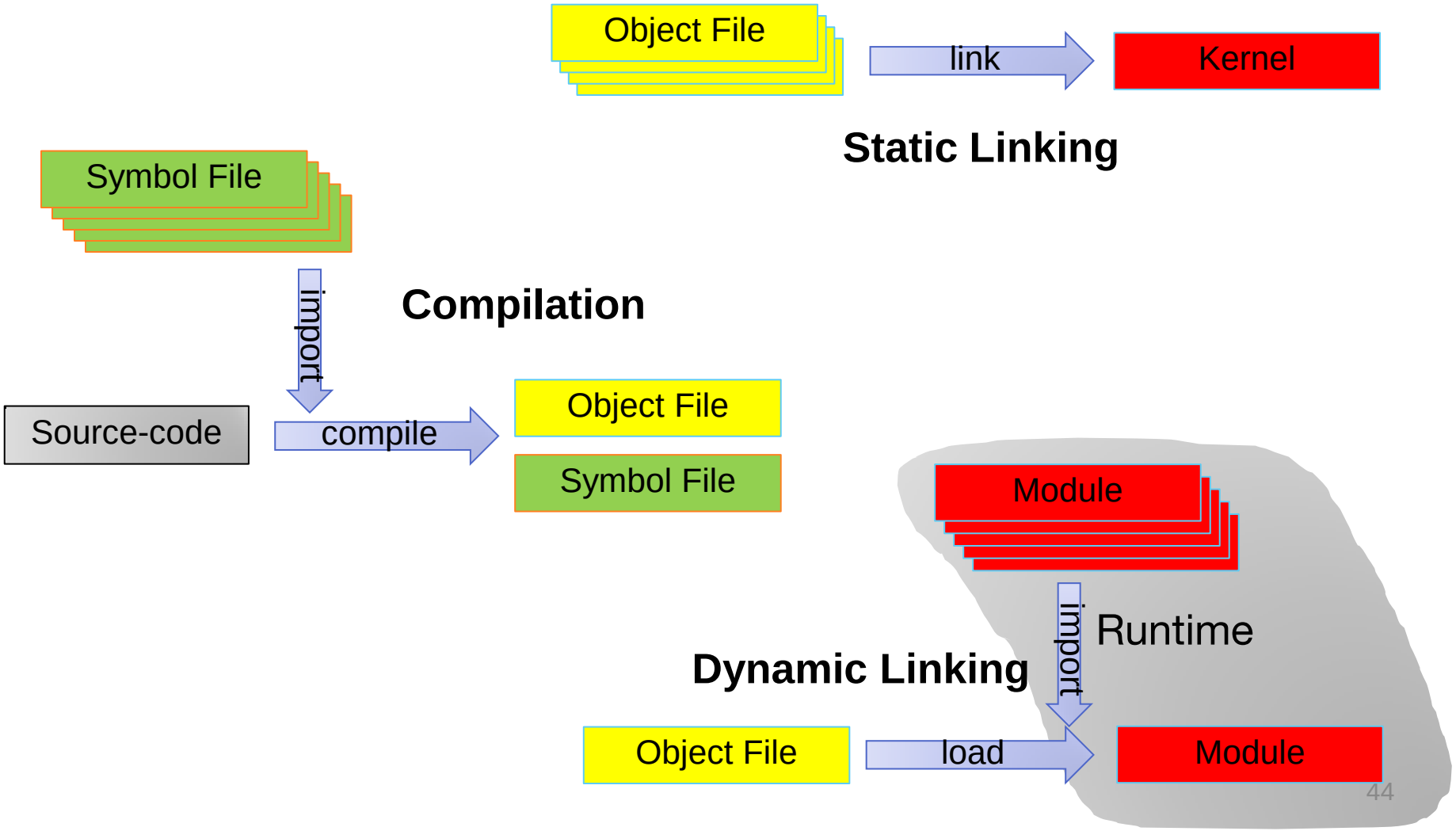
Fingerprint (must match)

(relative) Fixup

*here: Intel x86

The role of the Object File

in a system with dynamic loading



The Linking Process

- Linker or loader must provide at least a mechanism to resolve references when arranging sections of an object file in memory.
- Our special approach
  - The compiler generates all **metadata as ordinary data sections** and uses the fixup mechanism to establish the necessary links
  - Loader and linker are nearly identical and can use the same code base for patching fixups
  - No special metadata section in the object file

Object File Loading / Linking

- Traverse all sections of an object file
- Allocate space for code and data
- Arrange sections according to their placement rules
- Patch fixups

Linker: keep sections in a list of sections

Linker: sections available in the list

Loader: has to identify sections in already loaded modules (module.export) using a global hash table

Loader:
Call module body

Today's Lab

What? [Objectives]

- Learn to know the development environment A2 and language Oberon.
- Understand the synchronisation semantics of Active Oberon.
- Understand the compilation and linking process.
- Fun!

How? [Tasks]

- Complete an unfinished, non thread-safe implementation of a bounded buffer.
- Build and debug a kernel.

Workshop on System-on-Chip Design

Part II : Custom-Designed Systems On Chip on FPGA

National Chiao Tung University, 21.-23.10.2013

Felix Friedrich, ETH Zürich

Objectives of Part II

- Background and Motivation
- TRM Processor and Interconnects
- **Software Hardware Co-Design**
- The Active Cells Toolchain
- Case Studies and Examples

Motivation: Multicore Systems Challenges

- Cache Coherence
- Shared Memory Communication Bottleneck
- Thread Synchronization Overhead

⇨ Hard to predict performance of a program

⇨ Difficult to scale the design to massive multi-core architecture

Operating System Challenges

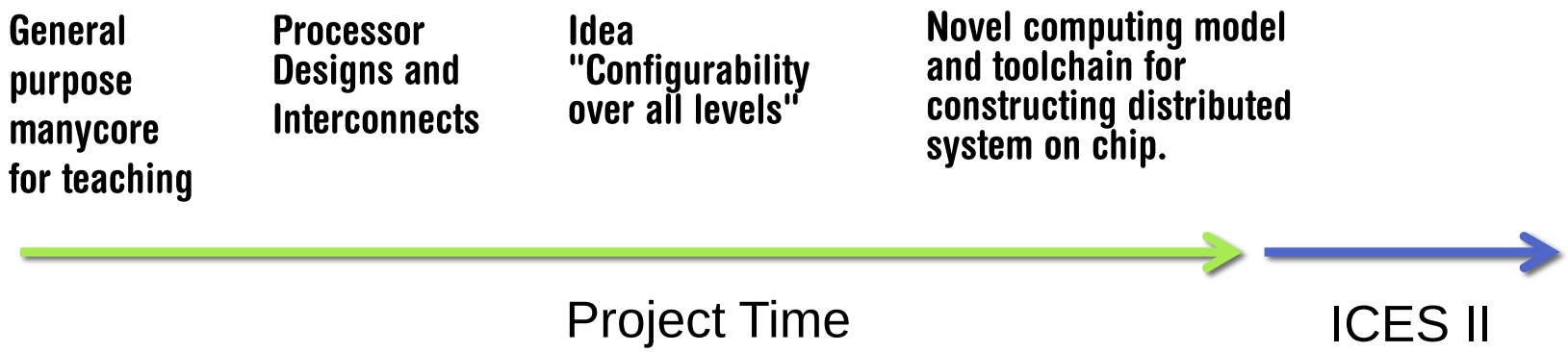
- Processor Time Sharing
  - Interrupts
  - Context Switches
  - Thread Synchronisation
- Memory Sharing
  - Inter-process: Paging
  - Intra-process, Inter-Thread: Monitors

Project *Supercomputer in the Pocket*

Funded by Microsoft in ICES programme, 2009 - 2014

Manycore architecture for embedded systems on the basis of programmable hardware (FPGA)

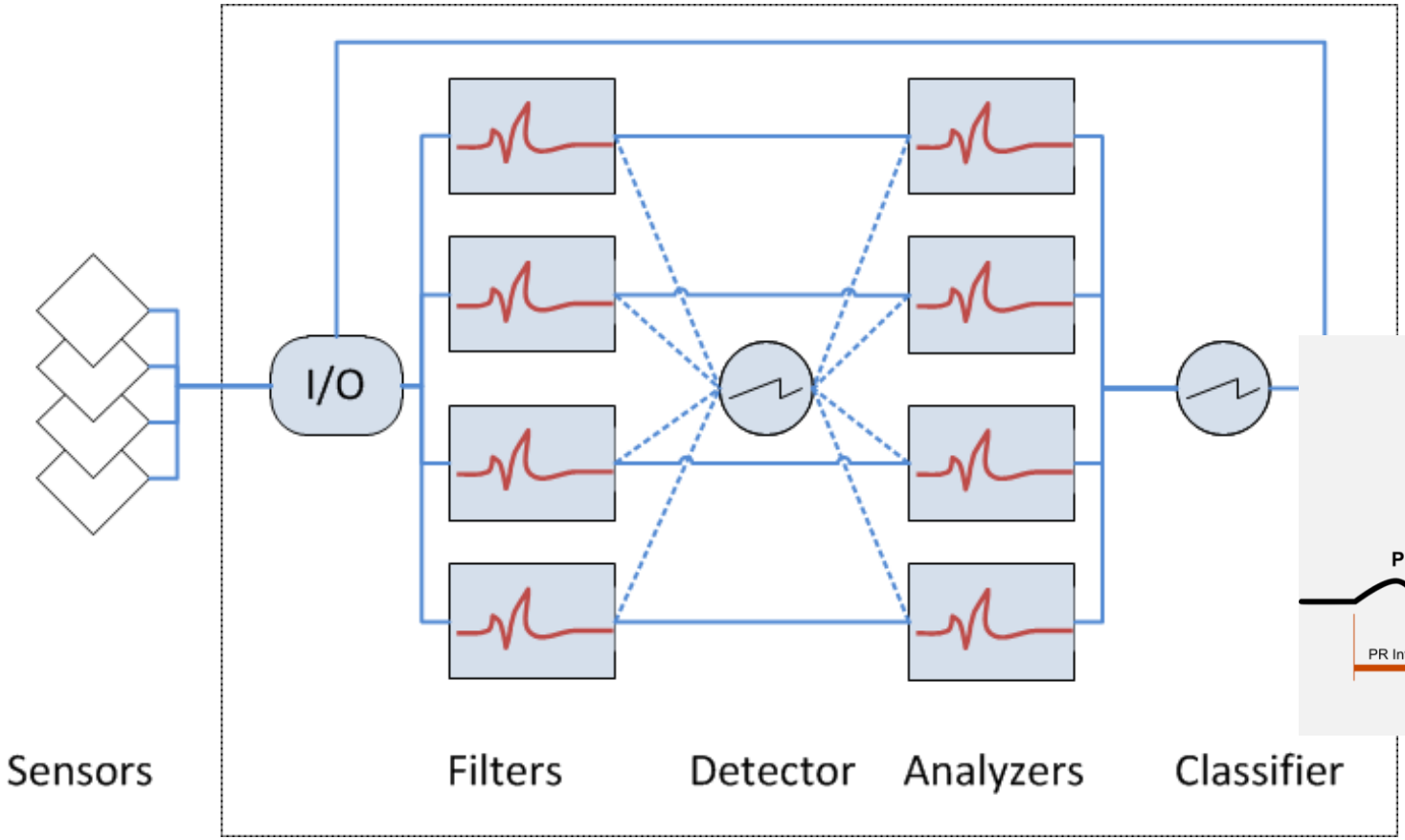
- Emphasis on high-performance computing in the small in the field of sensor driven medical IT
- Enhance industrial applications and ease teaching of parallel computing

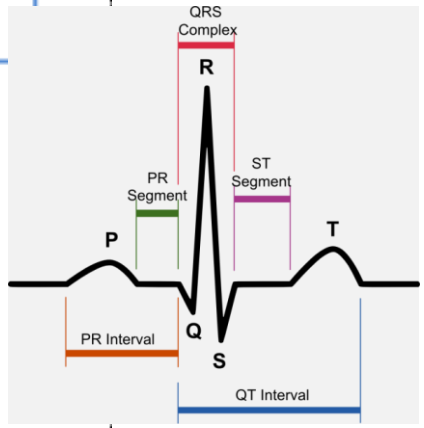


Credits

- Microsoft Research
  - Chuck Thacker (consultant)
- ETH Zürich
  - Jürg Gutknecht (computing model)
  - Niklaus Wirth (consulting, processor design)
  - Ling Liu (hardware and processor design)
  - Felix Friedrich (computing model and compiler)
  - Alexey Gokhberg (integration with .NET)
- University Hospital Basel
  - Patrick Hunziker (medicine and medical algorithms)
  - Alexey Morozov (medical signal processing, FPGA engines)
- National Chiao Tung University Taiwan
  - Shiao Li Tsao (energy aware computing)

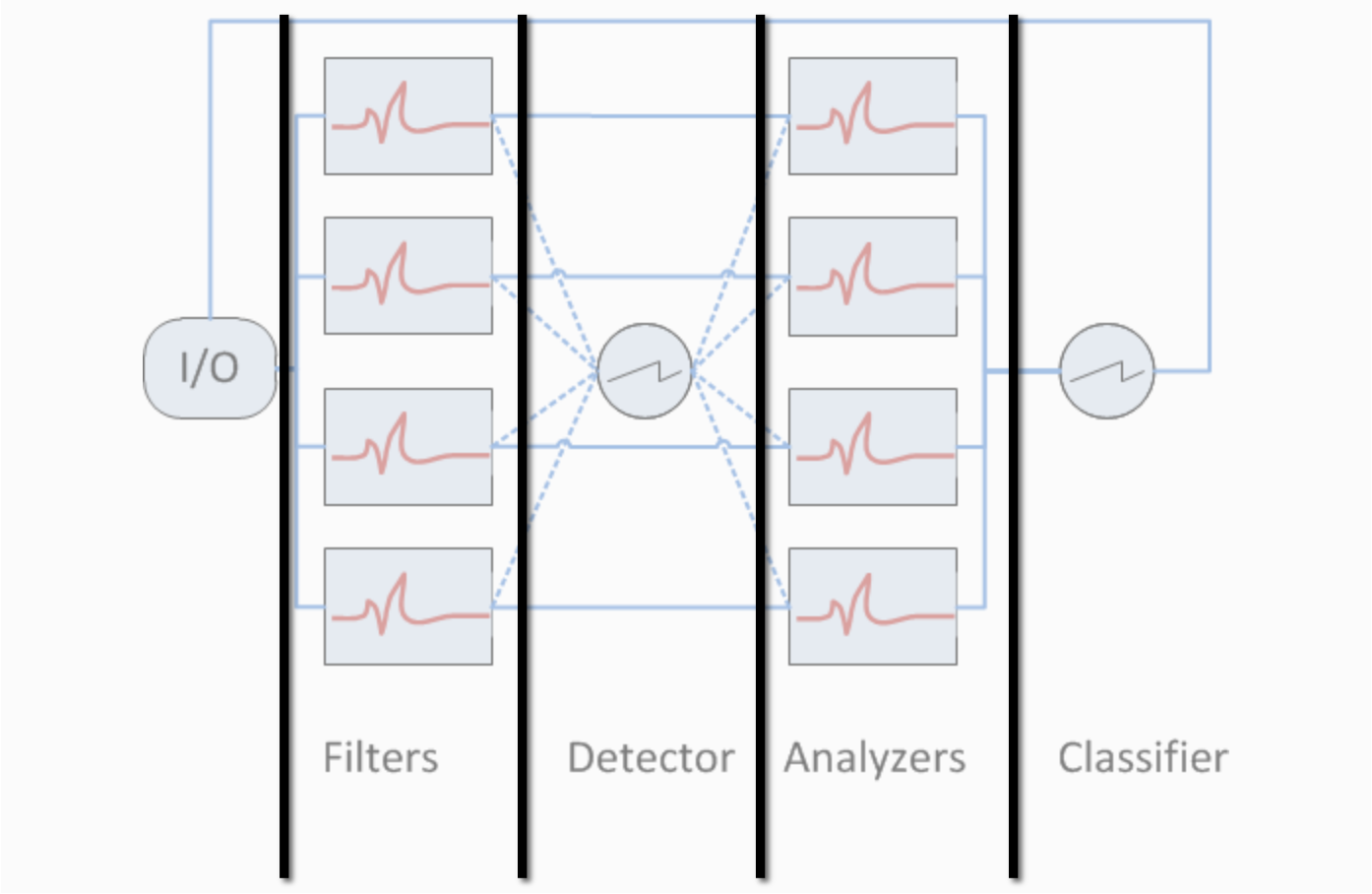
Focus: Streaming Applications





Structural Example:
ECG for realtime desease detection

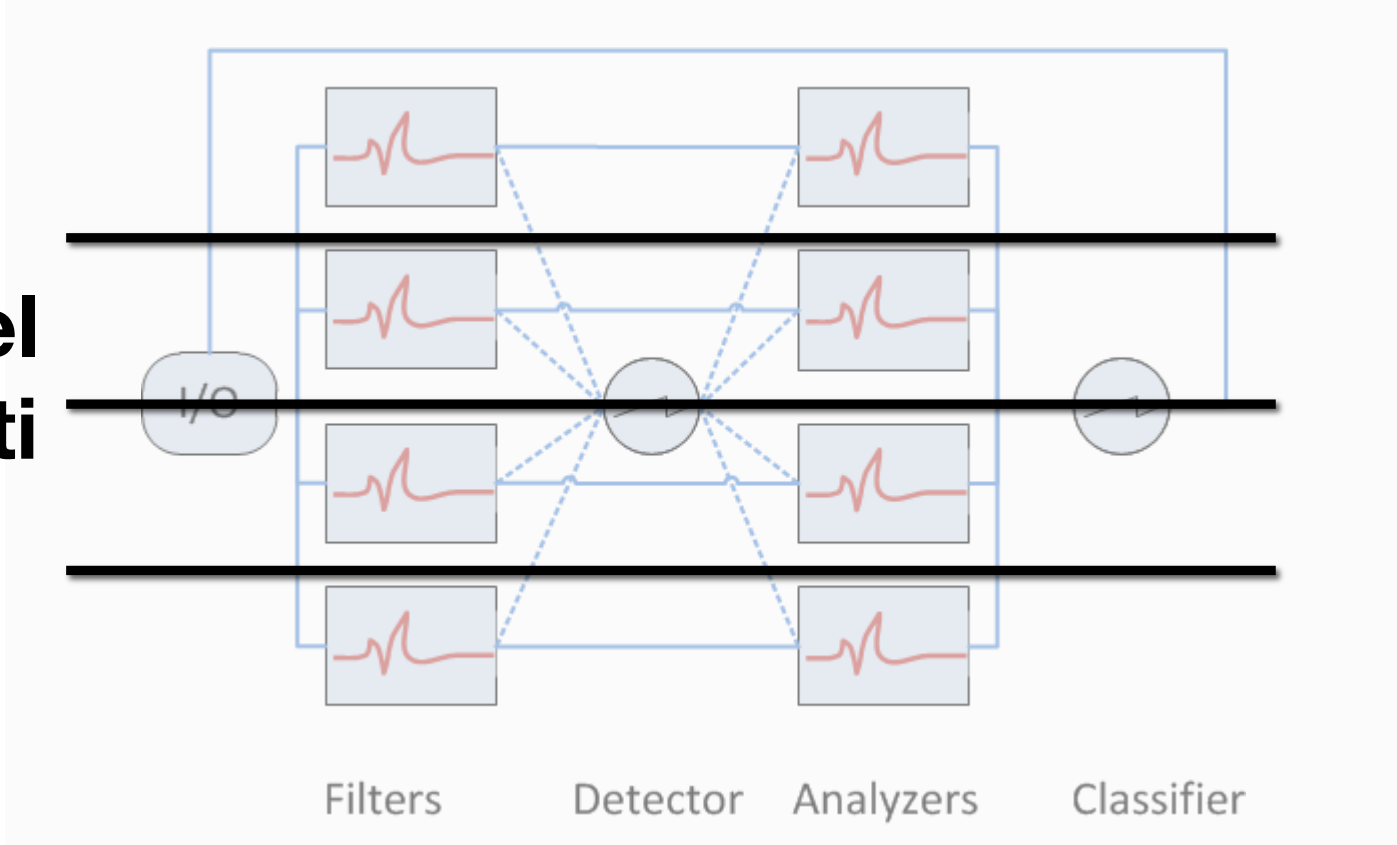
Stream Parallelism



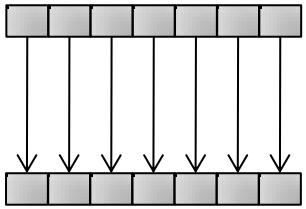
Pipelining

Task Parallelism

Parallel Execution



Data Parallelism



Vector Computing
Loop-level parallelism

Key Idea: On-chip distributed system

- Replace shared memory by local memory
  - Message passing for interaction between processes
- Separate processor for each process
  - Very simple processors
  - No scheduling, no interrupts,
  - Application-aware processors

→ Minimal operating system

→ Conceptually no memory bottleneck

→ Higher reliability and predictability by design

THE TINY REGISTER MACHINE – TRM

TRM: Tiny Register Machine*

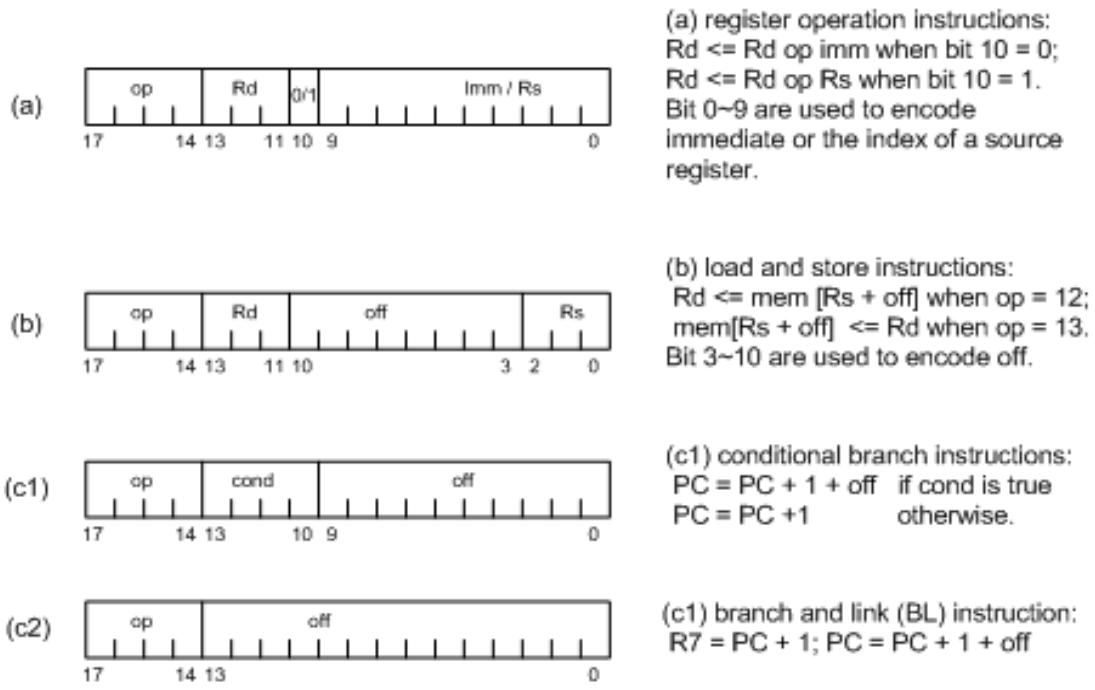
- Extremely simple processor on FPGA with Harvard architecture.
- Two-stage pipelined**
- Each TRM contains
  - Arithmetic-logic unit (ALU) and a shifter.
  - 32-bit operands and results stored in a bank of 2*8 registers.
  - local data memory: d*512 words of 32 bits.
  - local program memory: i*1024 instructions with 18 bits.
  - Register H for storing the high 32 bits of a product, and 4 conditional registers C, N, V, Z.
- No caches

* Invented and implemented by Dr. Ling Liu and Prof. Niklaus Wirth

Design Principle 1

Simplicity favors regularity

- Consistent instruction format
- Same number of operands, easier to encode and handle in hardware



TRM instruction encoding

from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

Design Principle 2

Make the common case fast

- ALU operations are performed on registers and constants
- TRM is a ***reduced instruction set computer*** **(RISC)**, with a small number of simple instructions.
  - The main characteristic of *RISC* architecture is to allow most instructions to be executed in one clock cycle.

from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

Design Principle 3

Smaller is Faster

- TRM includes only a small number of registers
- Just as retrieving data from a few books on your table is faster than sorting through 1000 books, retrieving data from 32 registers is faster than retrieving it from 1000 registers or a large memory.

from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

Design Principle 4

Good design demands good compromises

- Multiple instruction formats allow flexibility
  - `ADD`, `SUB`: use 2 register operands
  - `LD`, `ST`: use 2 register operands and a constant
- Number of instruction formats kept small
  - to adhere to design principles 1 and 3 (simplicity favors regularity and smaller is faster).

from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

TRM Machine Language

- Machine language: binary representation of instructions
- 18-bit instructions
- Three instruction types:
  - Type a: arithmetical and logical operations
  - Type b: load and store instructions
  - Type c: branch instructions (for jumping)

from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

TRM architecture

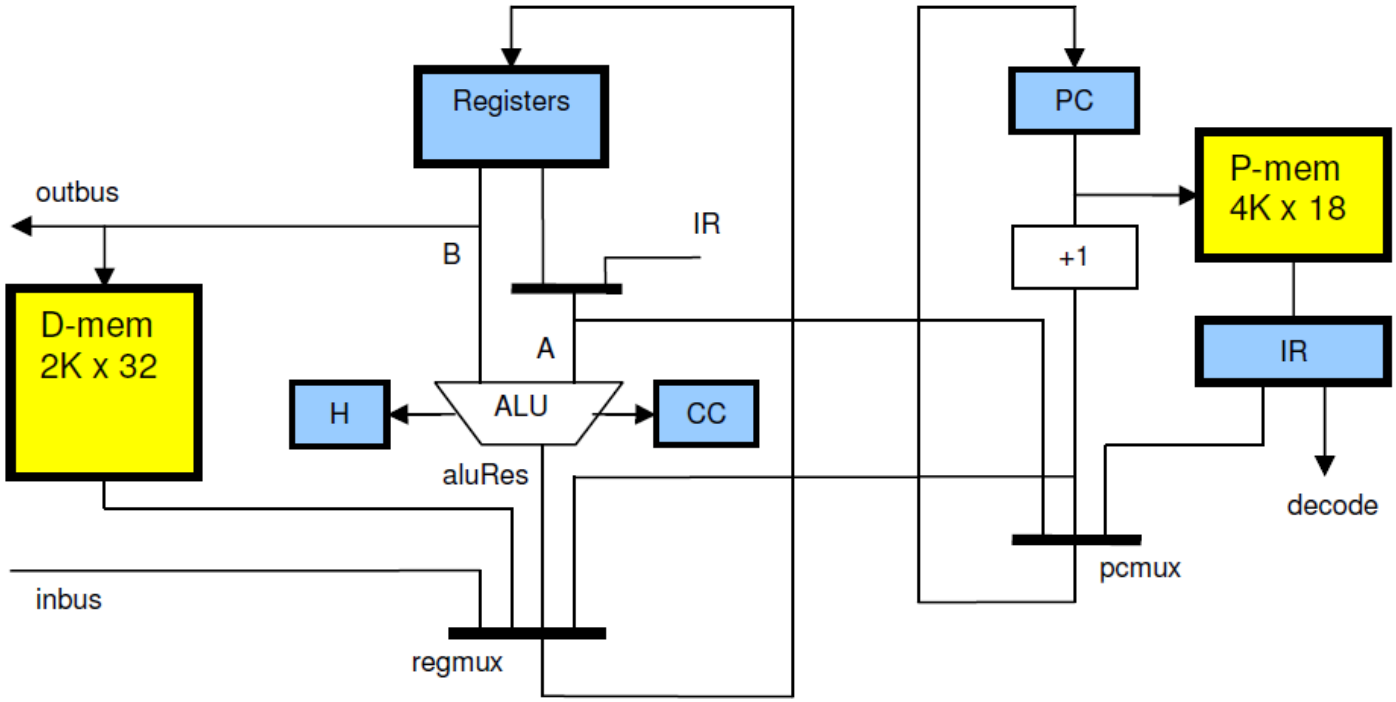


Figure from: Niklaus Wirth, *Experiments in Computer System Design*, Technical Report, August 2010
http://www.inf.ethz.ch/personal/wirth/Articles/FPGA-relatedWork/ComputerSystemDesign.pdf

The TRM Registers

Name	Register Number	Usage
R0~R7	0 ~ 7	Working register
R7	7	Normally used as link register
R6	6	Normally used as stack pointer
PC		Program pointer
C		Carry flag (1 bit)
N		Sign flag (1 bit)
Z		Zero flag (1 bit)
V		Overflow flag (1 bit)

from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

Variants of TRM

- FTRM
  - includes floating point unit
- VTRM (Master Thesis Dan Tecu)
  - includes a vector processing unit
  - supports 8 x 8-word registers
  - available with / without FP unit
  - two-stage pipelined

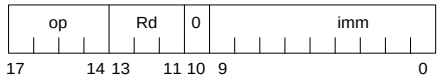
(V)TRM Instructions Summary

- RISC architecture with a regular instruction set
- 18bit instructions.
- Supports interrupts → irq instruction table at lowest instruction memory.
- Instructions
  - Register Operations (Reg/Imm, Reg/Reg, VReg/Vreg)
    - mov, not, add, sub, and, bic, or, xor, mul, ror,
    - spsr, br
  - Load and Store (Mem[Reg + offs])
    - ld, st
  - Conditional Branch and Branch-and-Link
    - bc, bl
  - Special Instructions
    - ldh, blr, rti, hadd

Encoding Overview

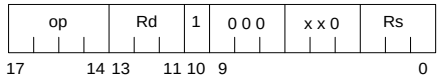
- Register Operations

(a)

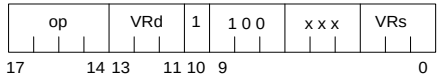


imm is zero extended to 32 bits

(b)

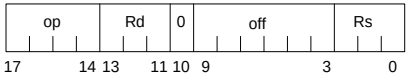


(c)



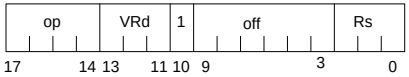
- Load and Store

(a)

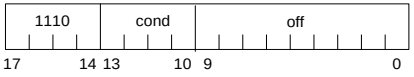


off is zero extended to 13 bits

(b)



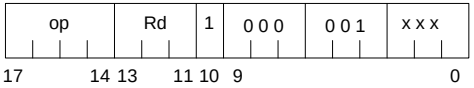
- Conditional Branches



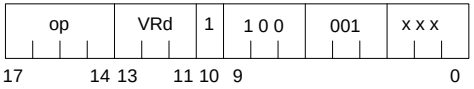
off is sign extended to 12 bits

- Special Instructions

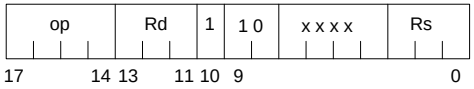
(a)



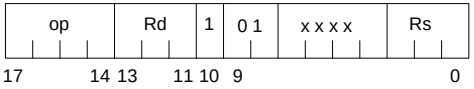
(b)



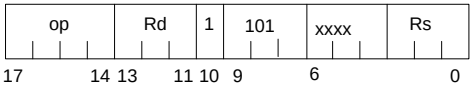
(c)



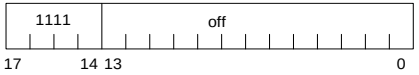
(d)



(e)

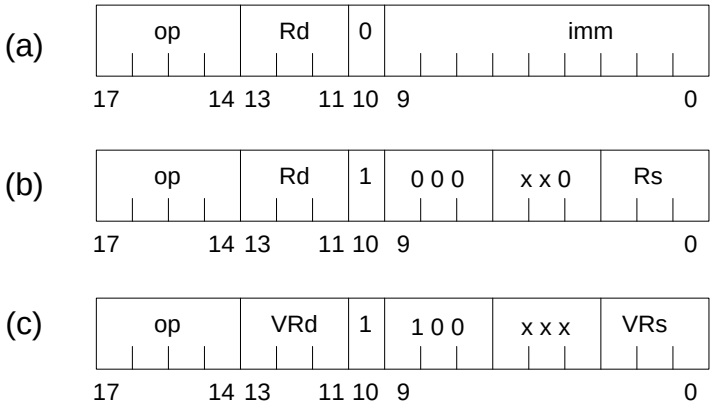


- Branch and Link



off is 14-bit offset

Encoding Example



imm is zero extended to 32 bits

op	instruction	operation	encoding
0000	MOV Rd, imm	Rd := imm	(a)
0000	MOV Rd, Rs	Rd := Rs	(b)
0000	MOV VRd, VRs	VRd := VRs	(c)
0001	NOT Rd, imm	Rd := ~imm	(a)
....			
1011	BR Rs	PC := Rs	(b) with Rd=0

PL vs. HDL

Programming Language

- Sequential execution
- No notion of time

```
var a,b,c: integer;

a := 1;
b := 2;
c := a + b;
```

unknown mapping to machine cycles

Hardware Description Language

- Continuous execution (combinational logic)

```
wire [31:0] a,b,c;

assign a=1;
assign b=2;
assign c=a+b;
```

no memory associated

- Synchronous execution (register transfer)

```
reg [31:0] a,b,c;

always @ (posedge clk)
begin
   a <= 1;
   b <= 2;
   c <= a+b;
end;
```

synchronous at rising edge of the clock

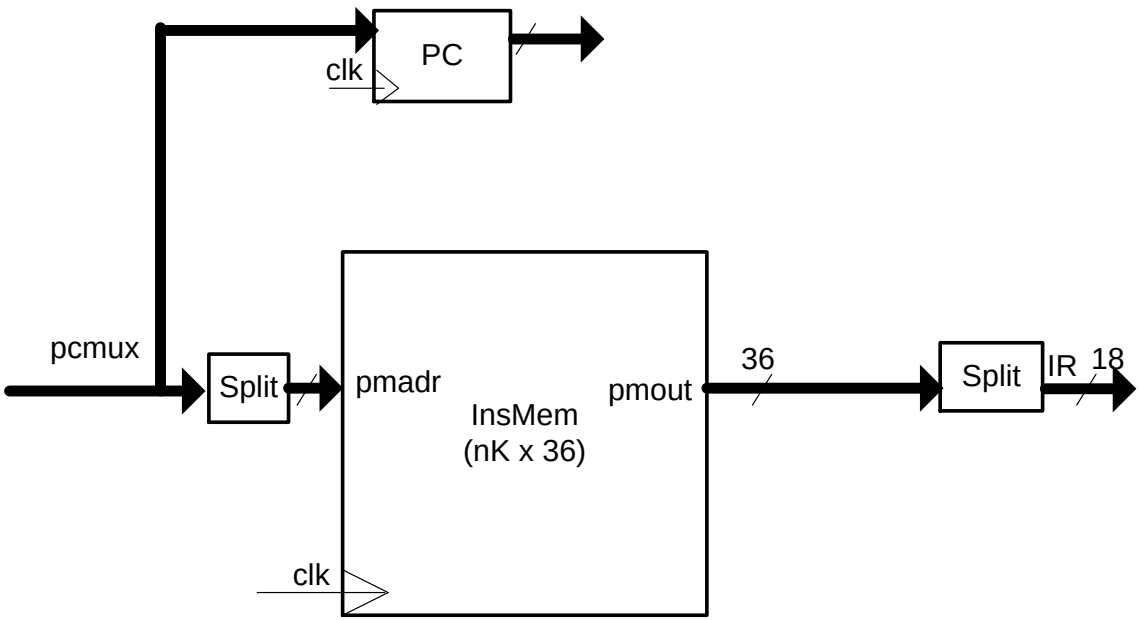
TRM Architectural State

- PC
- 8 registers
- flag registers
- Memory (configurable)
  - nK * 36 bits instruction memory (1k = 1024)
  - nk * 32 bits data memory

from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

Single-Cycle Datapath: arithmetical logical instruction fetch

- We consider executing **ADD R0, R1 (0x08401)**
- **STEP 1:** Fetch instruction



from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

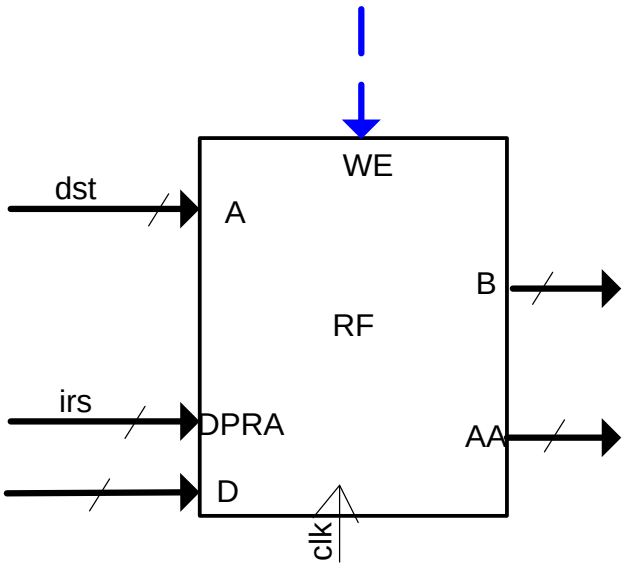
Single-Cycle Datapath: instruction fetch

```
wire [PAW-1:0] pcmux, nxpc;
wire [17:0] IR;
reg [PAW-1:0] PC;

IM #(.BN(IMB)) imx(.clk(clk), .pmadr({{{33-PAW}{1'b0}},
            pcmux[PAW-1:1]}), .pmout(pmout));

assign IR = (~rst)? NOP: (PC[0]) ? pmout[35:18] : pmout[17:0];

always @ (posedge clk, negedge rst) begin
  if (~rst) PC <= 0;
  else if (stall0)
        PC <= PC;
  else
        PC <= pcmux;
  end
```

Single-Cycle Datapath: register read

ADD R0 R1 (0x08401)

- **STEP 2:** Read source operands from register file



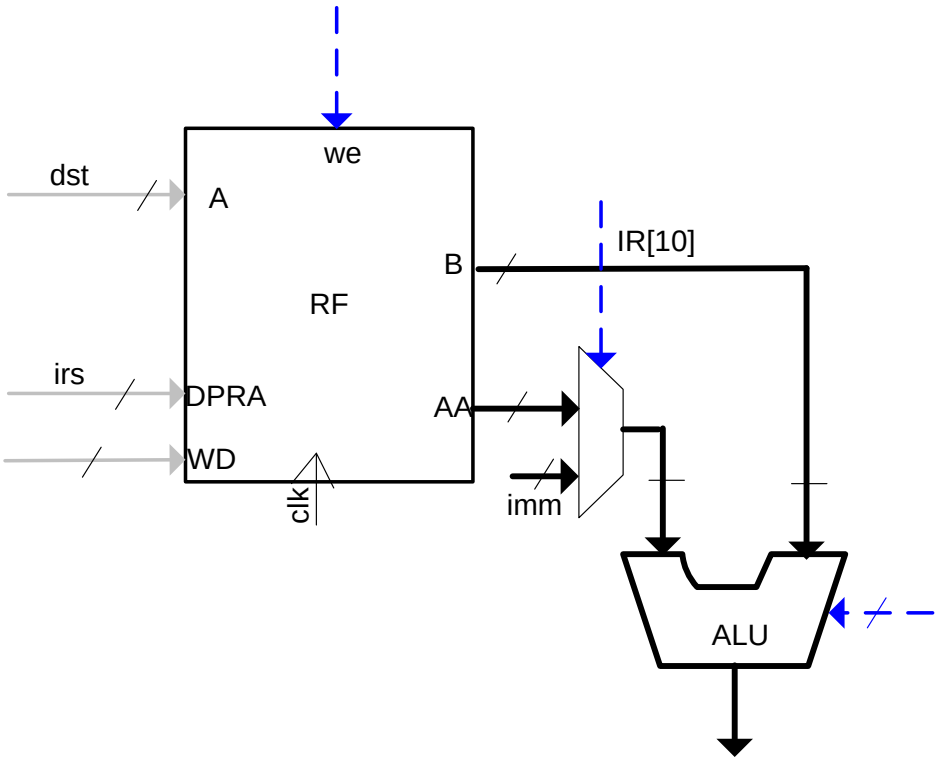

```
wire [2:0] rd, rs;
wire regWr;
wire [31:0] rdOut, rsOut;

//register file
…

assign irs = IR[2:0];
assign dst = (BL)? 7: ird;
```


from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

Single-Cycle Datapath: ALU

ADD R0 R1 (0x08401)

- **STEP 3:** Compute the result via ALU




```
wire [31:0] AA, A, B, imm;
wire [32:0] aluRes;

assign A = (IR[10]? AA:
                    {22'b0, imm};

//alu
…
```


from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

Single-Cycle Datapath: ALU

ADD R0 R1 (0x08401)

$IR_{17:14}$	Function
0000	B := A
0001	B := ~A
0010	B := B + A
0011	B := B – A
0100	B := B & A
0101	B := B & ~A
0110	B := B | A
0111	B := B ^ A

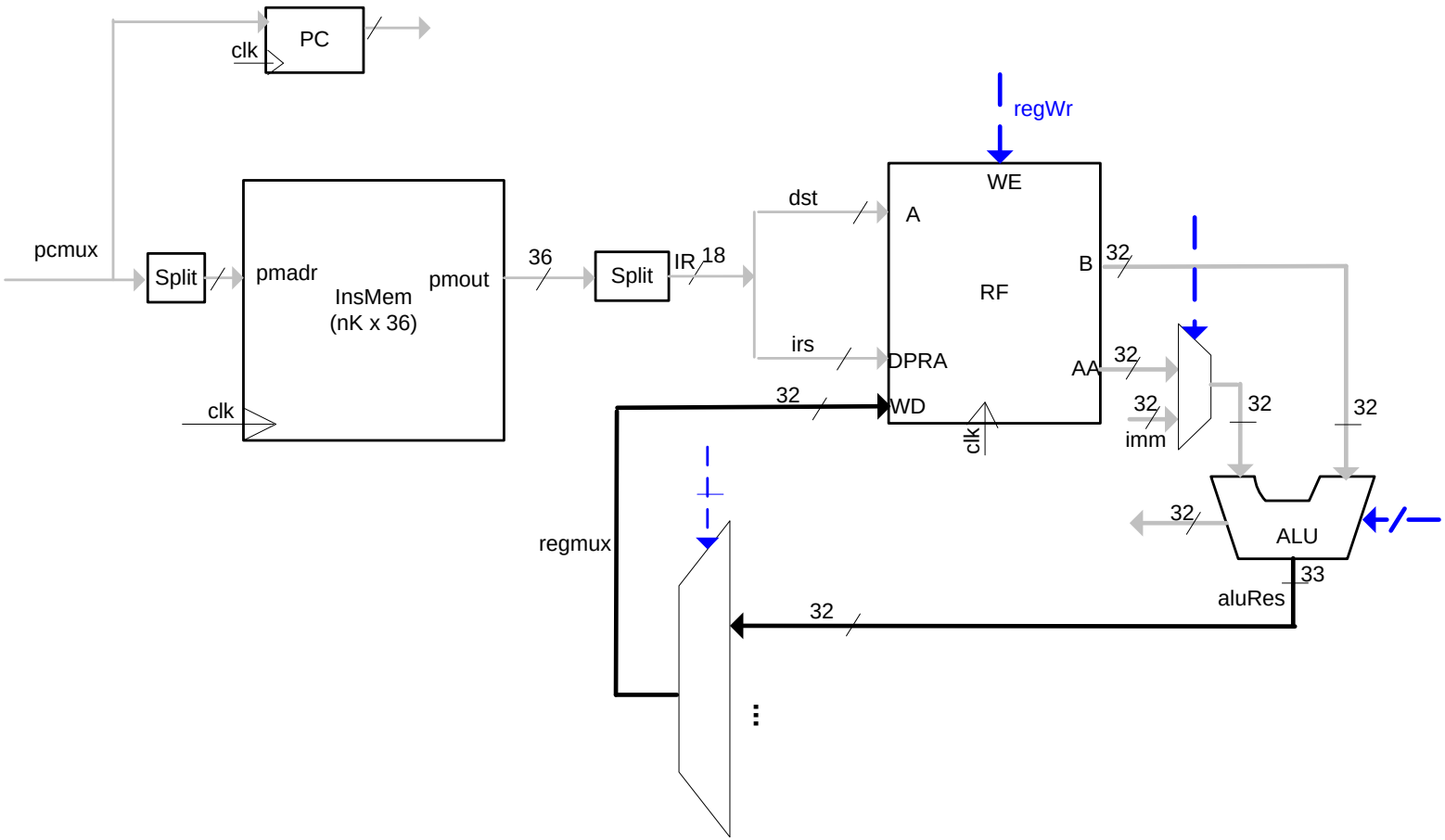
Verilog code in TRM module

```
wire [32:0] aluRes, minusA;

assign minusA = {1'b0, ~A} + 33'd1;
assign aluRes =
        (MOV)? A:
        (ADD)? {1'b0, B} + {1'b0, A} :
        (SUB)? {1'b0, B} + minusA :
        (AND)? B & A :
        (BIC)? B & ~A :
        (OR)? B | A :
        (XOR)? B ^ A :
        ~A;
```

Single-Cycle Datapath: write back to Rd

ADD R0 R1 (0x08401)

- **STEP 4:** Write result back to Rd



from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

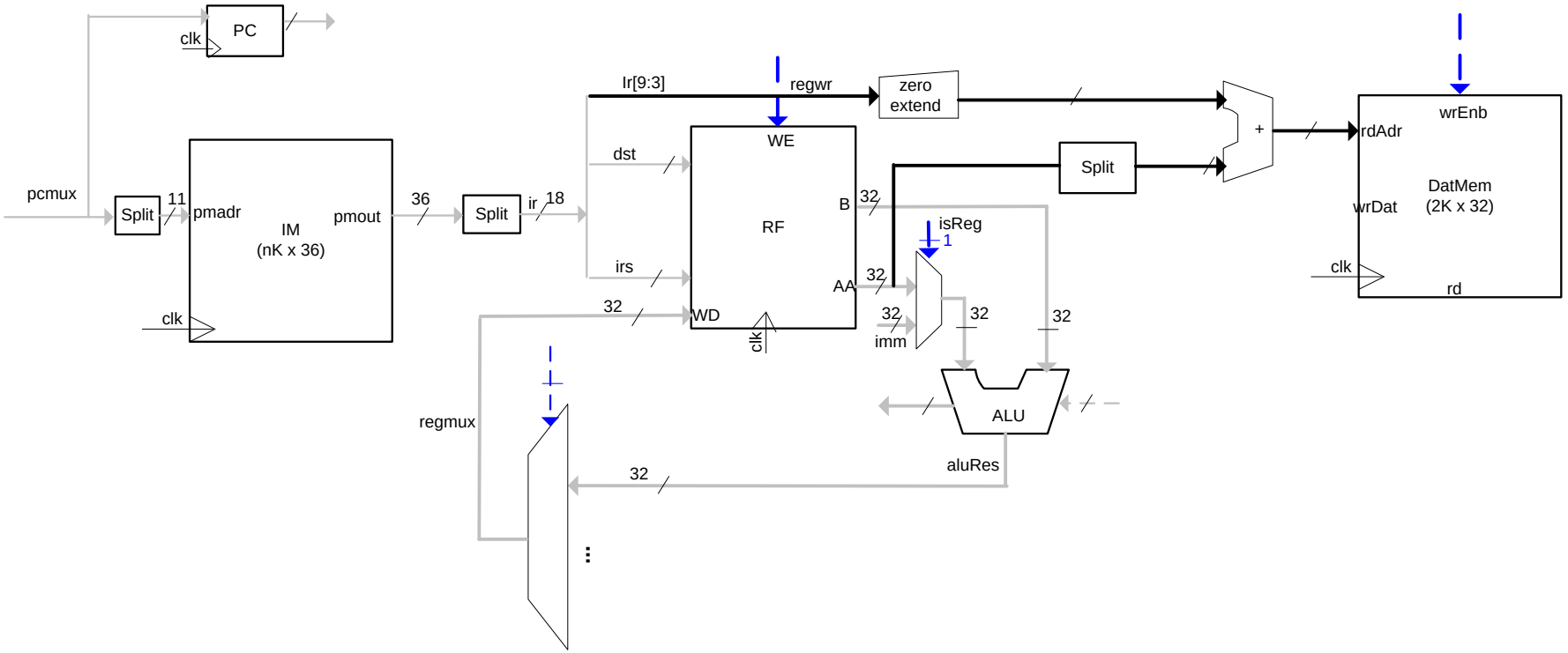
Single-Cycle Datapath: write back to Rd

ADD R0 R1 (0x08401)

```
wire [31:0] regmux;
wire regwr;

//register file
...

assign regwr = (BL | BLR | LDR & ~IR[10] |
                    ( ~(IR[17] & IR[16]) & ~BR & ~vector)) &
               ~stall0;

assign regmux = ...
          aluRes;
```

Single-Cycle Datapath: LD

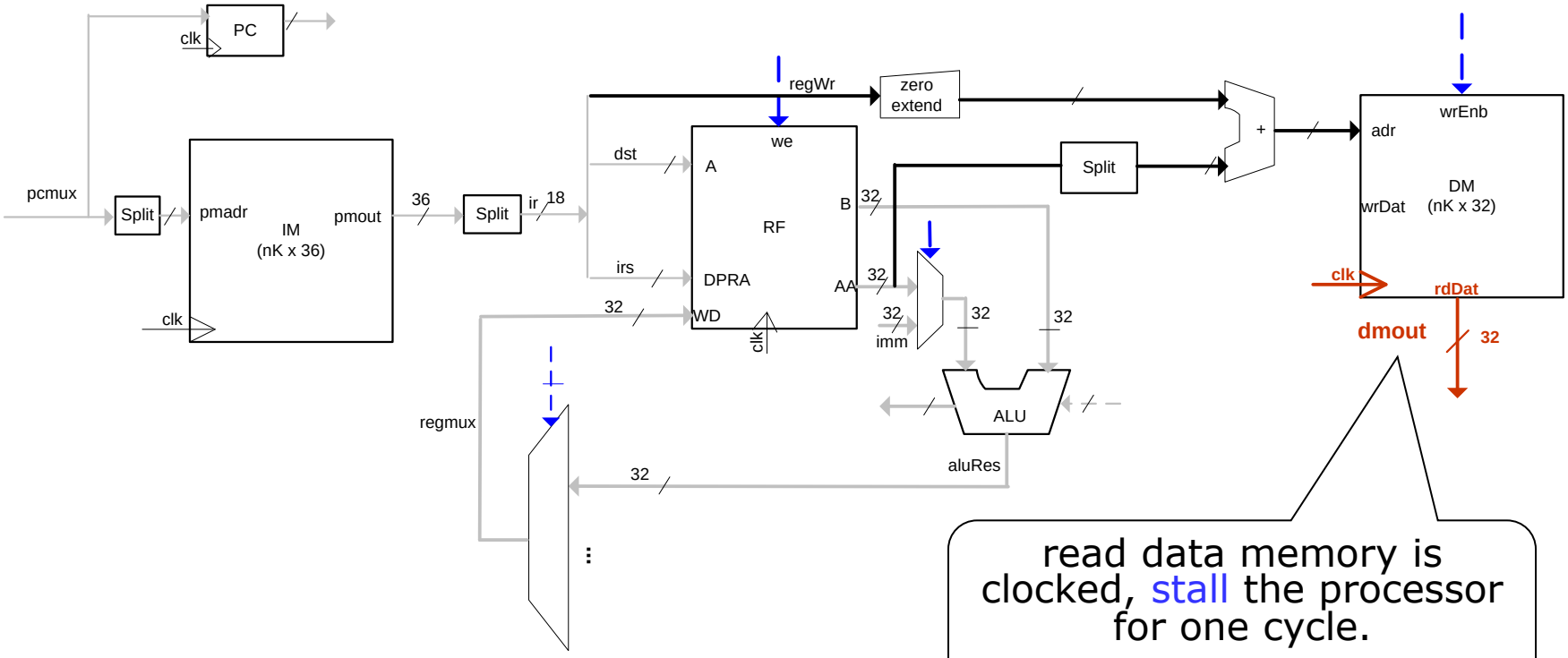
- **STEP 1:** Fetch instruction
- **STEP 2:** Read source operand from the register file
- **STEP 3:** Compute the memory address



from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

Single-Cycle Datapath: LD

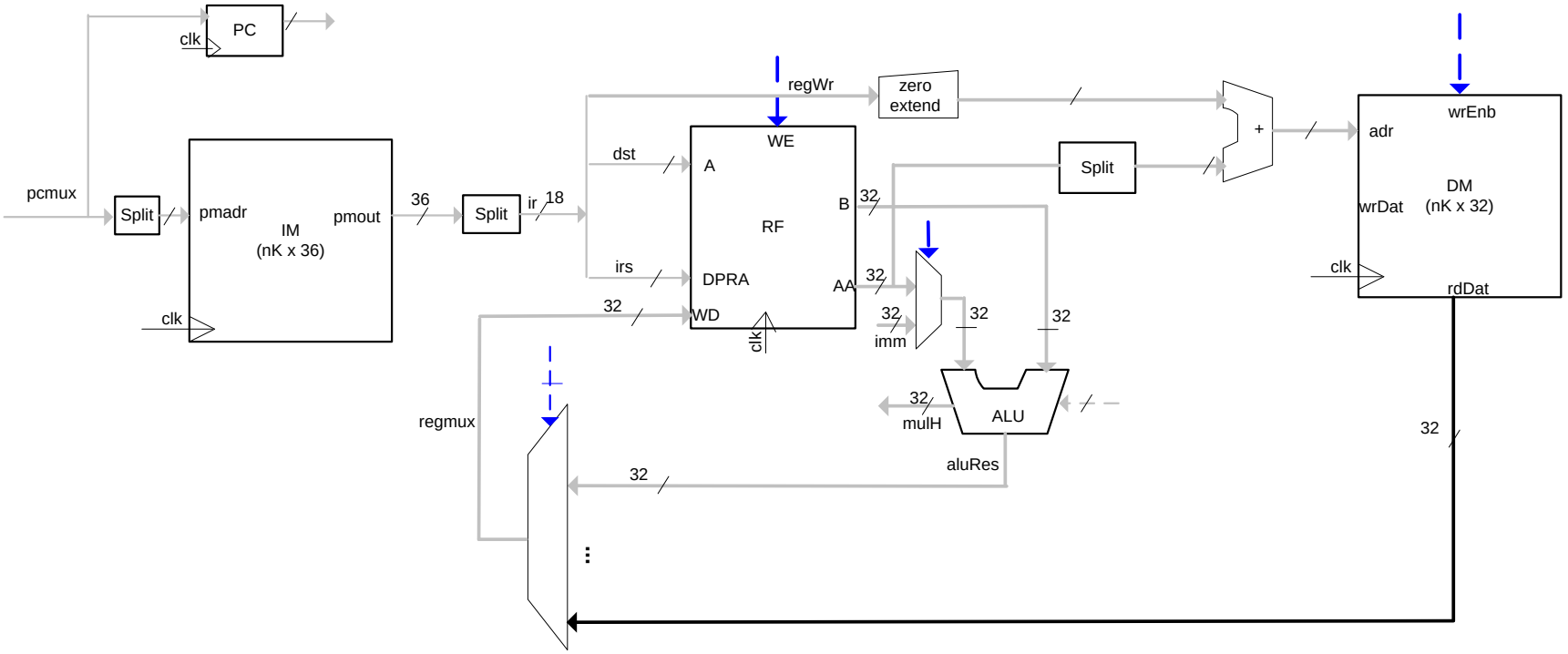
- **STEP 3:** Compute the memory address
- **STEP 4:** Read data from data memory



from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

Single-Cycle Datapath: LD

- **STEP 4:** Read data from data memory
- **STEP 5:** Write data back into the register file



from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

TRM: LD

Verilog code in TRM module

```
wire [31:0] dmout;
wire [DAW:0] dmadr;
wire [6:0] offset;
reg IoenbReg;

//register file
…

Assign dmadr = (irs == 7) ?
     {{{DAW-6}{1'b0}}, offset} :
     (AA[DAW:0] + {{{DAW-6}{1'b0}}, offset});

assign ioenb = &(dmadr[DAW:6]);
assign rfWd = ...
               (LDR & ~IoenbReg)? dmout:
               (LDR & IoenbReg)? InbusReg: //from IO
               ...;
always @(posedge clk)
     IoenbReg <= ioenb;
```

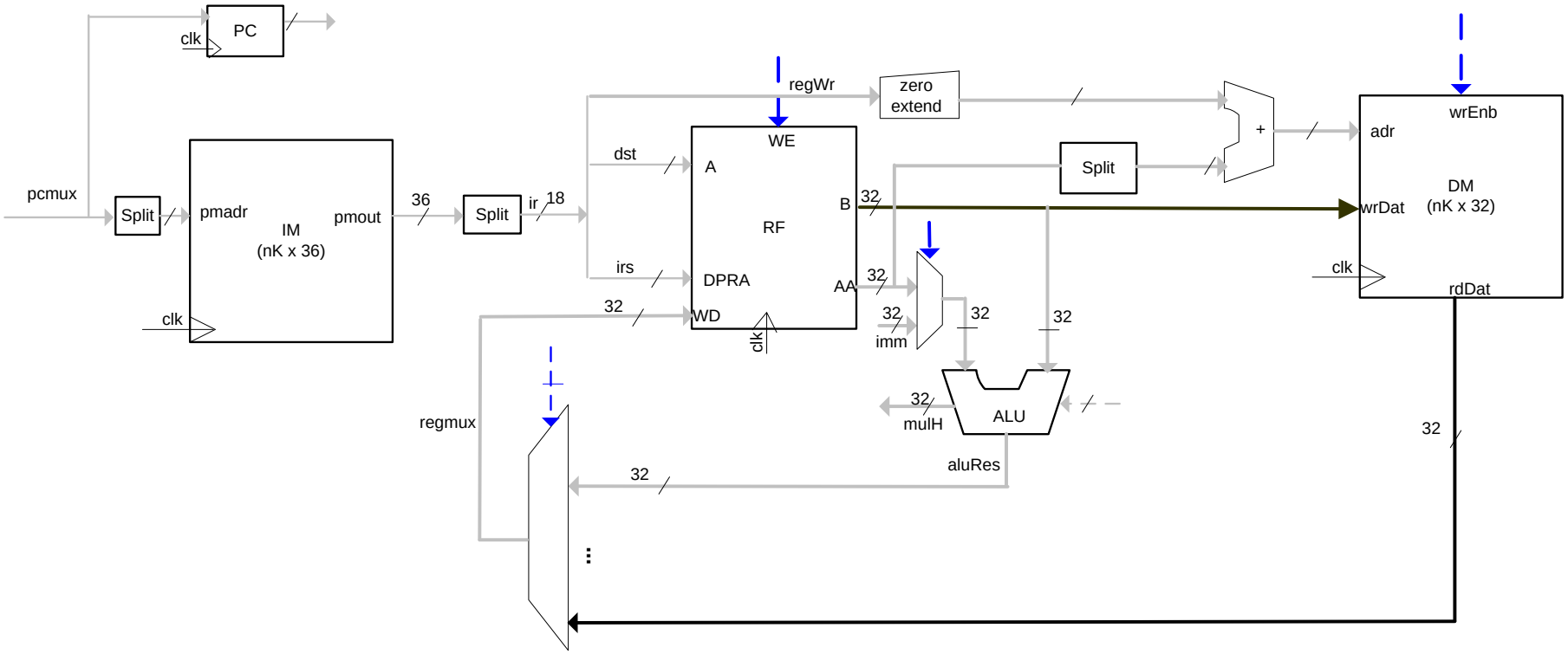
LD rd, [rs+offset]

Src register = Ir ignored (Harvard architecture!)

I/O space: Uppermost 2^6 bytes in data memory

Single-Cycle Datapath: ST

- **STEP 1:** Fetch instruction
- **STEP 2:** Read source operand from the register file
- **STEP 3:** Compute the memory address
- **STEP 4:** Write data into the data memory



from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

Single-Cycle Datapath: ST

- **STEP 3:** Compute the memory address
- **STEP 4:** Write data into the data memory

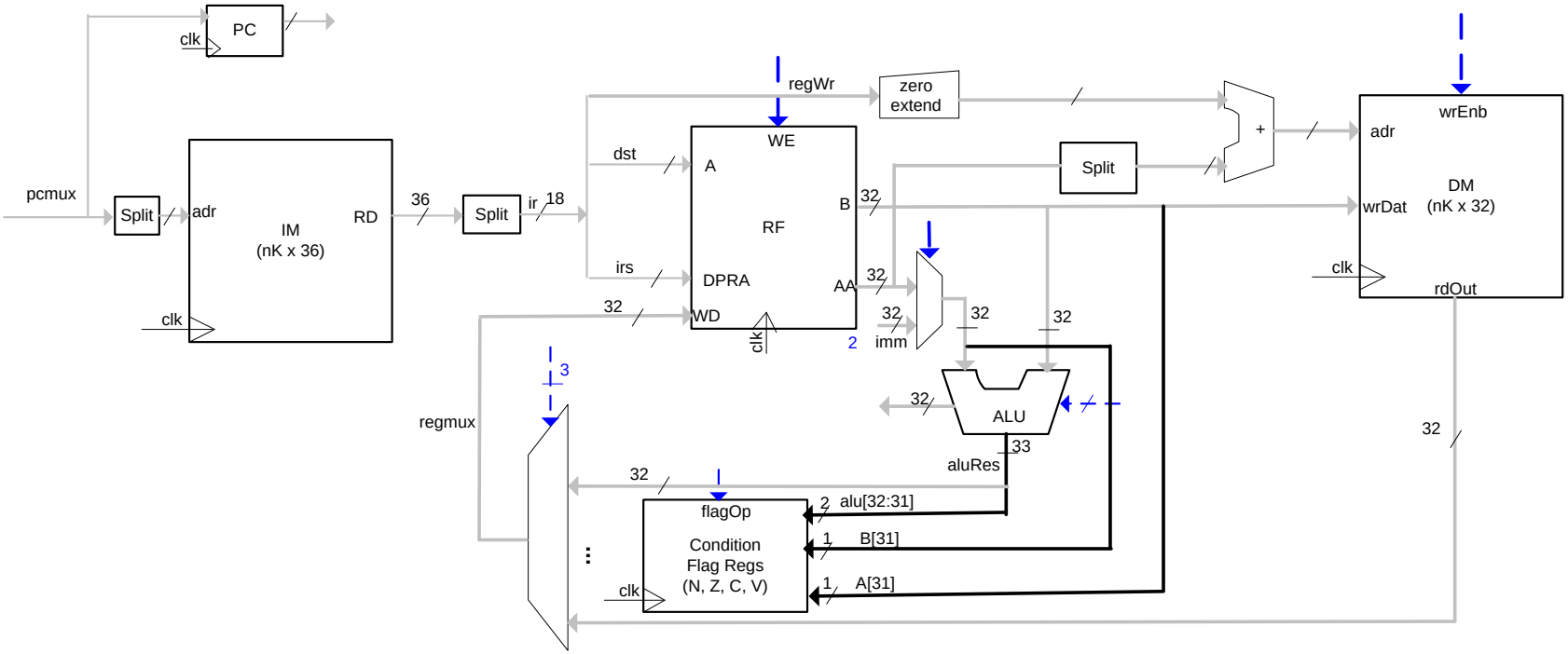
```
wire [31:0] dmin;
wire dmwr;

//register file
…
DM #(.BN(DMB)) dmx (.clk(clk),
   .wrDat(dmin),
        .wrAdr({{{31-DAW}{1'b0}},dmadr}),
        .rdAdr({{{31-DAW}{1'b0}},dmadr}),
        .wrEnb(dmwe),
        .rdDat(dmout));

Assign dmwe = ST & ~IR[10] & ~ioenb;
assign dmin = B;
```

Single-Cycle Datapath: set flag registers

```
always @ (posedge clk, negedge rst) begin  // flags handling
  if (~rst) begin N <= 0; Z <= 0; C <= 0; V <= 0; end
  else begin
       if (regwr) begin
            N <= aluRes[31];
            Z <= (aluRes[31:0] == 0);
            C <= (ROR & s3[0]) | (~ROR & aluRes[32]);
            V <= ADD & ((~A[31] & ~B[31] & aluRes[31])
                       | (A[31] & B[31] & ~aluRes[31]))
                       | SUB & ((~B[31] & A[31] & aluRes[31])
                       | (B[31] & ~A[31] & ~aluRes[31]));
       end
  end
end
```


from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

Single-Cycle Datapath: set flag registers



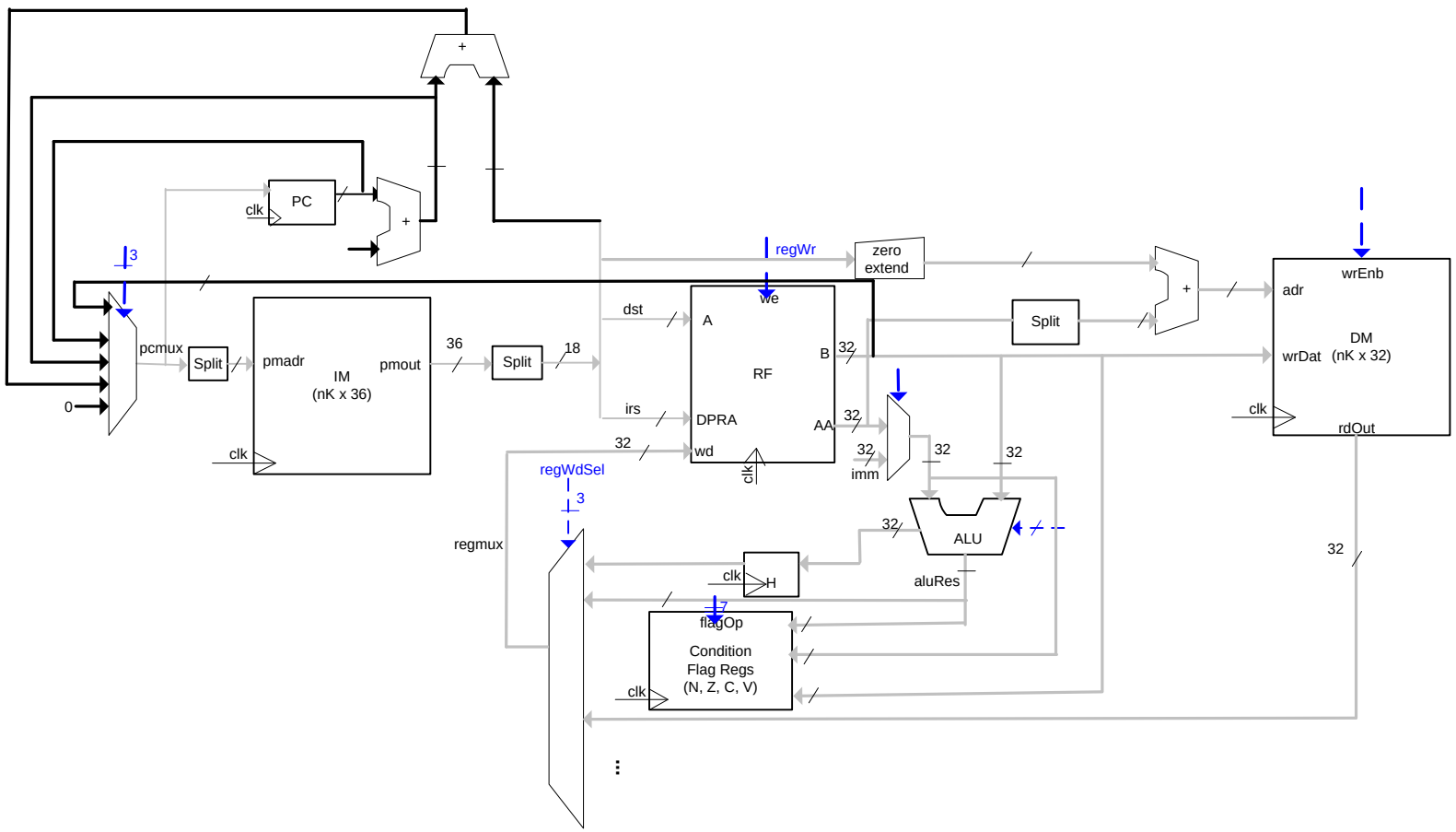
from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

Single-Cycle Datapath: Branch instructions

- Type c instructions, BR instruction, BL instruction
  - PC <= PC + 1 + off
  - PC < Rs
  - **PC <= PC + 1 (by default)**
  - **PC <= PC (if stall)**
  - **PC <= 0 (reset)**

from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

Single-Cycle Datapath: Branch instructions



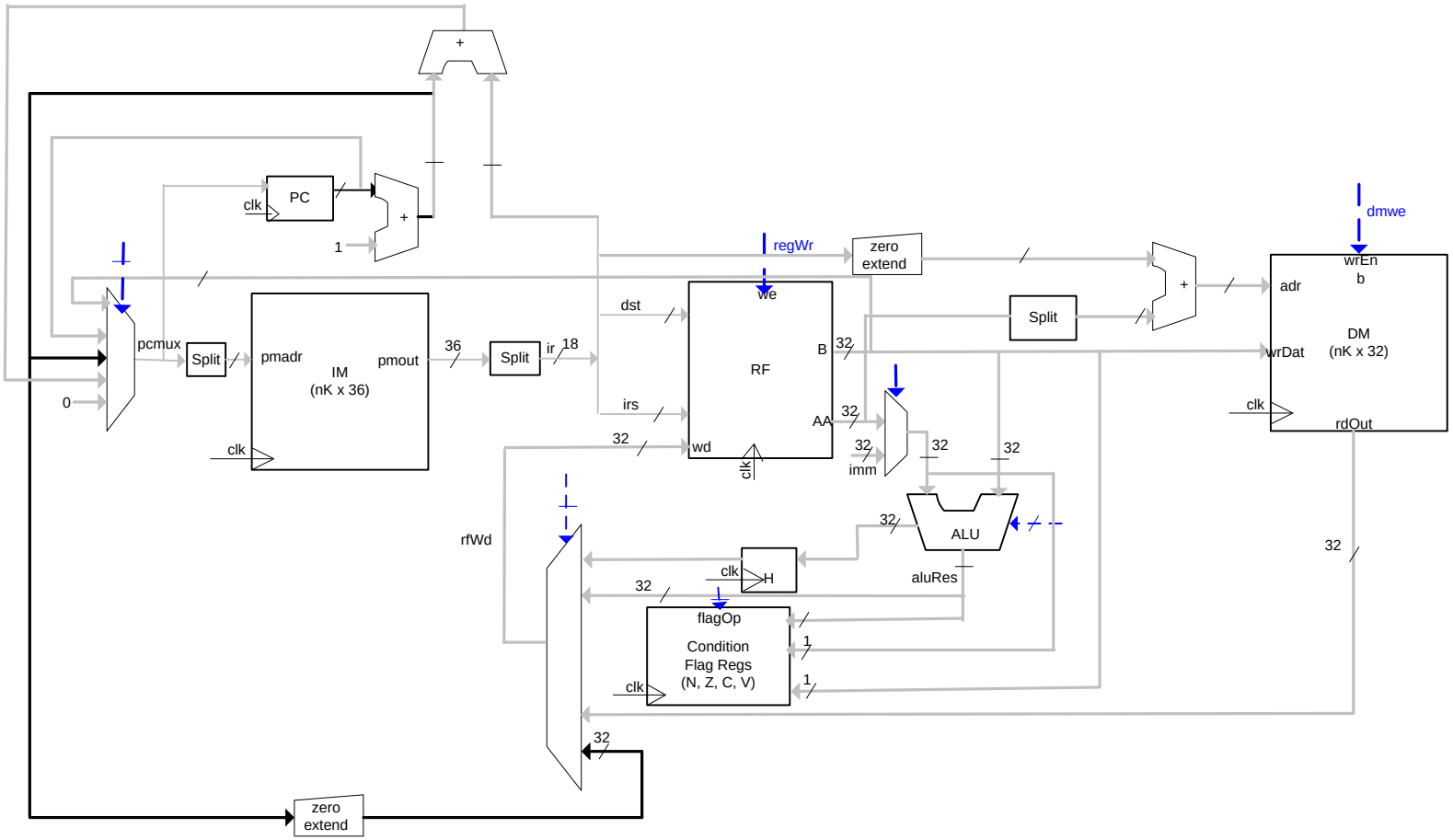
from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

Single-Cycle Datapath: Branch instructions

```
//pcmux logic
assign pcmux =
  (~rst) ? 0 :
  (stall0) ? PC:
  (BL)? {{10{IR[BLS-1]}},IR[BLS-1: 0]}+ nxpc :
  (Bc & cond) ? {{{PAW-10}{IR[9]}}, IR[9:0]} + nxpc :
  (BLR | BR ) ? A[PAW-1:0] :
  nxpc;
```

Complete single-cycle datapath

- Instruction BL n ; R7 <= PC + 1, PC <= PC + 1 +n



from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

Control Path

```
assign vector = IR[10] & IR[9] & ~IR[8] & ~IR[7];

assign MOV = (op == 0);
assign NOT = (op == 1);
assign ADD = (op == 2);
assign SUB = (op == 3);
assign AND = (op == 4);
assign BIC = (op == 5);
assign OR  = (op == 6);
assign XOR = (op == 7);
assign MUL = (op == 8) & (~IR[10] | ~IR[9]);
assign ROR = (op == 10);
assign BR  = (op == 11) & IR[10] & ~IR[9];
assign LDR = (op == 12);
assign ST  = (op == 13);
assign Bc  = (op == 14);
assign BL  = (op == 15);
assign LDH = MOV & IR[10] & IR[3];
assign BLR = (op == 11) & IR[10] & IR[9];
```


from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

For completenes: regmux

```
wire [PAW-1:0] pcmux, nxpc;

assign regmux =
    (BL | BLR) ? {{{32-PAW}{1'b0}}, nxpc} :
    (LDR & ~IoenbReg) ? dmout :
    (LDR & IoenbReg)? InbusReg: //from IO
    (MUL) ? mulRes[31:0] :
    (ROR) ? s3 :
    (LDH) ? H :
    aluRes;
```

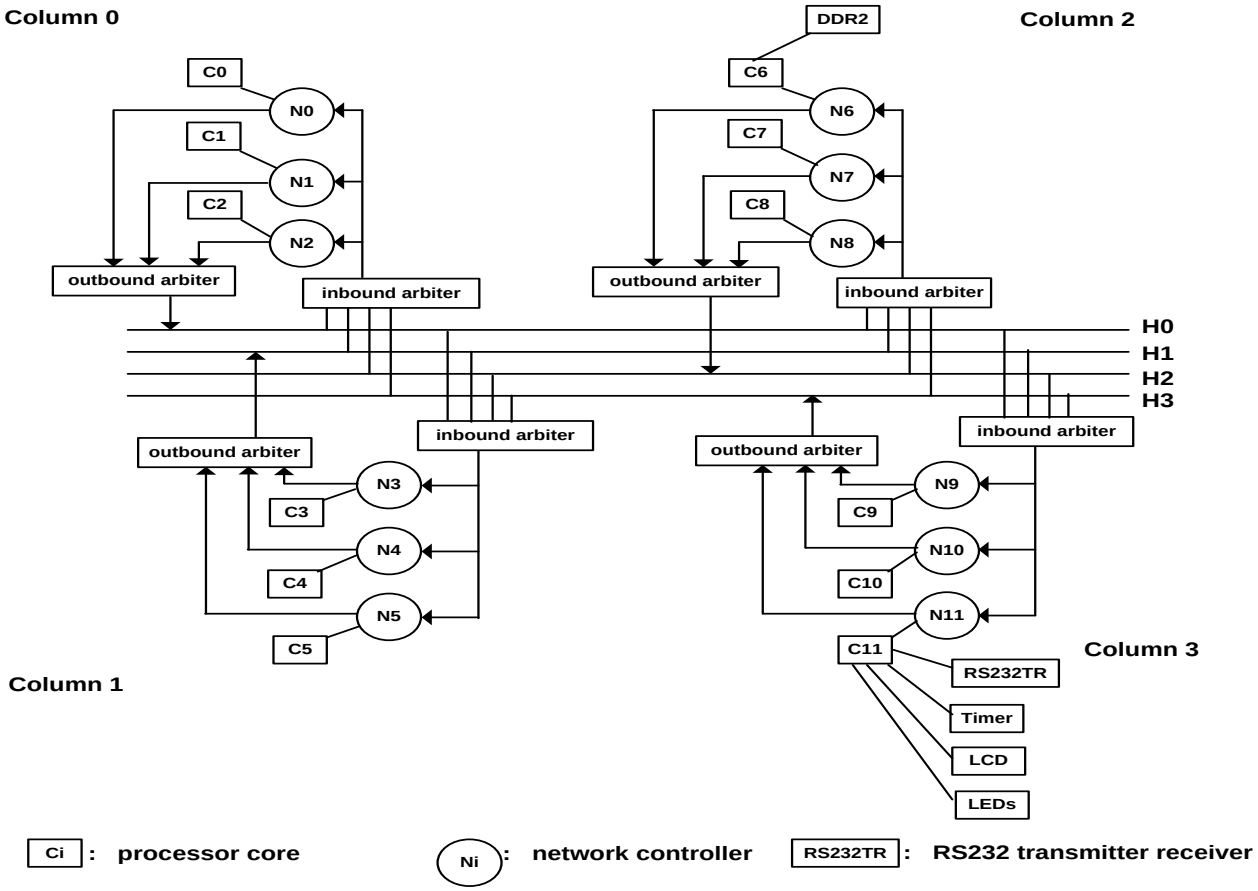
TRM Stalling

- stop fetching next instruction, pcmux keeps the current value
- disable register file write enable and memory write enable signals to avoid changing the state of the processor.
  - only LD and MUL instructions stall the processor.
  - dmwe signal is not affected.
  - regwr signal is affected.

TRM INTERCONNECTS

First Experiment: TRM12

A Multicore Processor Architecture on FPGA



- 12 RISC Cores (two stage pipelined at 116MHz)
- Message passing architecture
- Bus based on-chip interconnect
- On-chip Memory controller

Ling Liu, A 12-Core-Processor Implementation on FPGA, ETH Technical Report, October 2009
ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/6xx/646.pdf

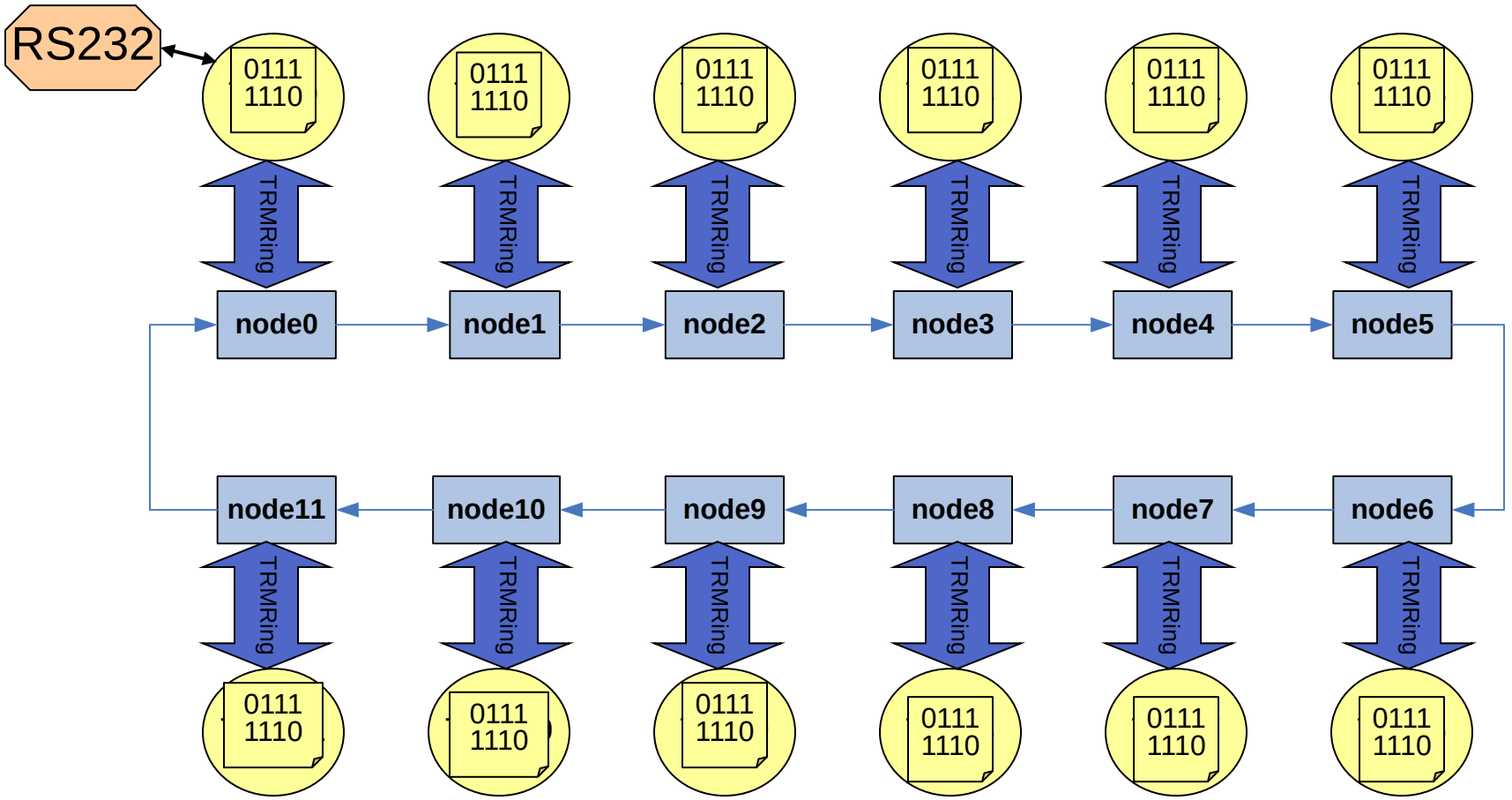
Interface to network and I/O

- TRM processor connected to a network controller („NetNode“)
- TRM core 11 connected to RS232 controller, a 2-line LCD controller, a timer and 8 LEDs
- TRM processor core 6 connected to 512 MB DDR2 controller
- Netnodes and RS232 controller treated as I/O port to the TRM processor, communication with TRM core through 32-bit I/O bus
- I/O accessed via memory mapped I/O at fixed addresses

Problems with this approach

- Not scalable
- Huge resource consumption
- Little but existing contention

Second Experiment: Ring of 12 TRMs



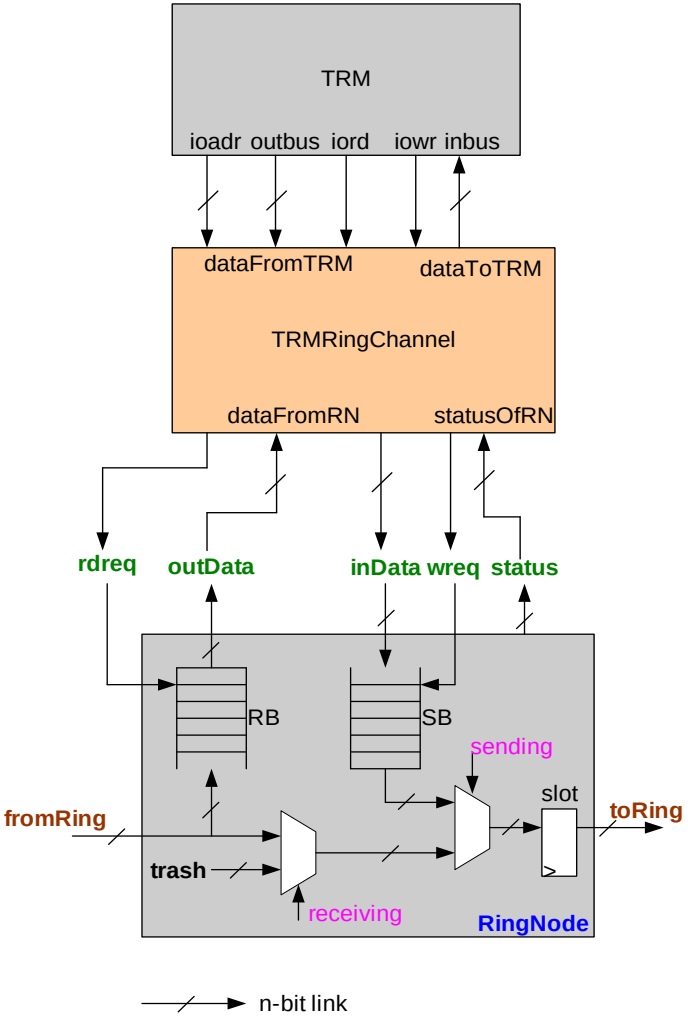
Slide from Dr. Ling Liu's Lecture Series on Reconfigurable Computing

Connection TRM / Ring

TRM

Adapter

Ring



- Ring interconnect very simple
- Small router
- Predictable latency

Slide from Dr. Ling Liu‘s Lecture Series on Reconfigurable Computing

Problems with this approach

- Not scalable without huge loss of performance
- Large delays

Next Experiment: Configurable Interconnect

infer knowlegde from software

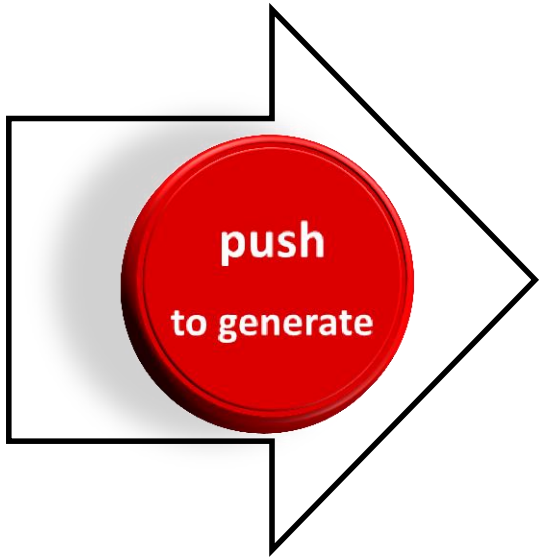
leading to the following visions

that have become reality

Software / Hardware Co-design

Vision: Custom System on Button Push

System design as high-level program code



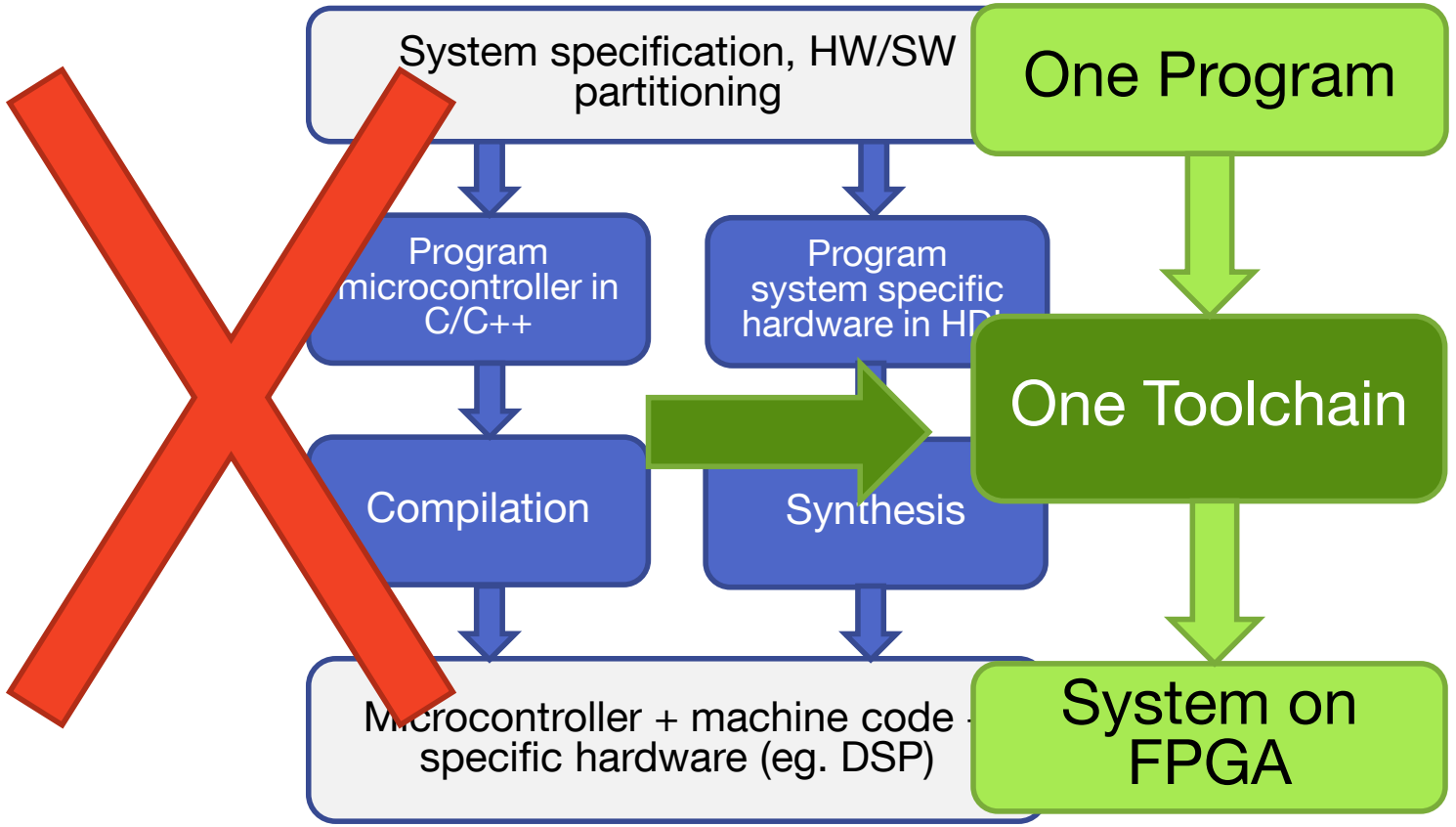
Electronic circuits

Computing model
Programming Language

Compiler, Synthesizer, Hardware Library, Simulator

Programmable Hardware (FPGA)

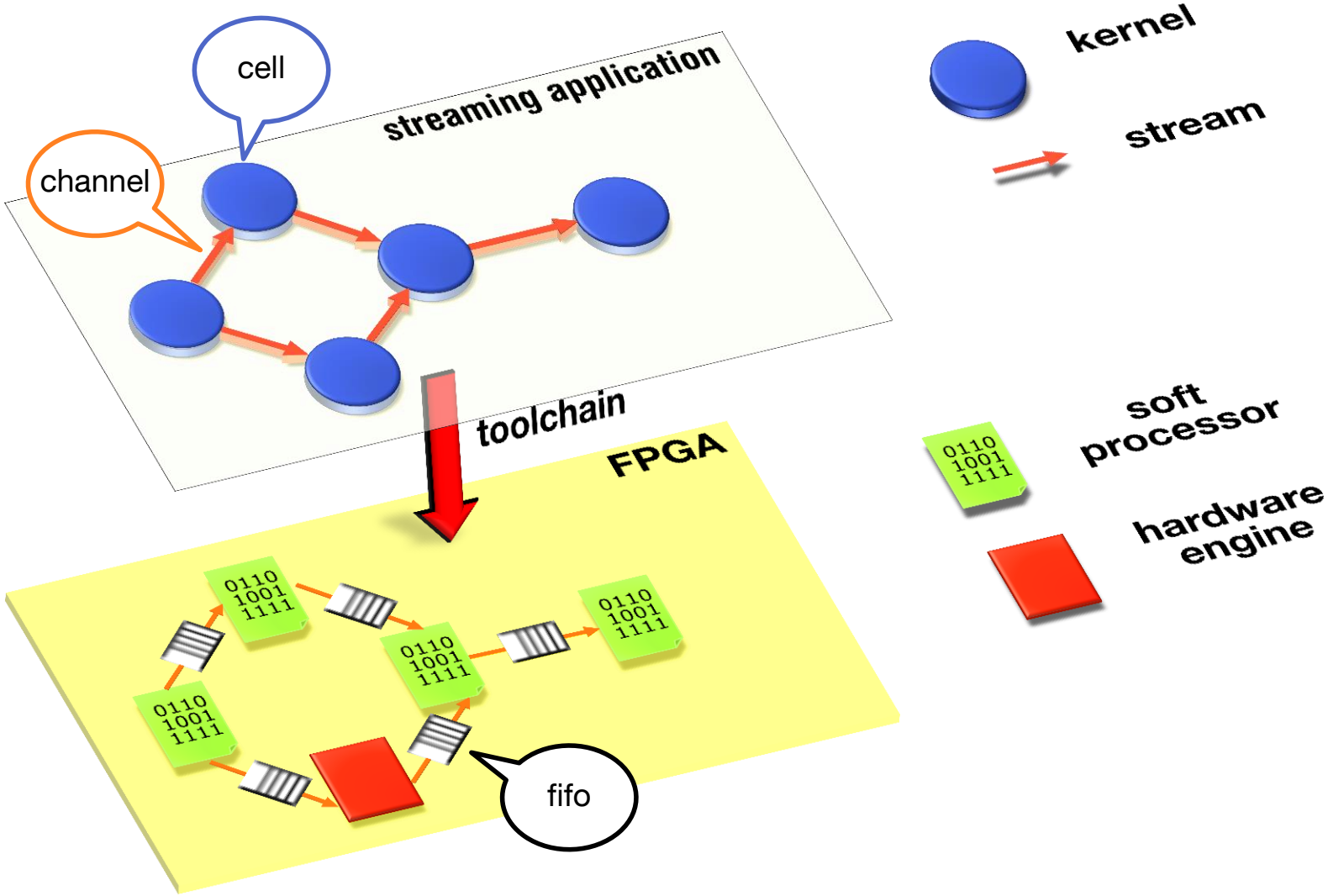
Traditional HW/SW co-design



Traditional HW/SW co-design for embedded systems

Active Cells approach for embedded systems development

Software → Hardware Map



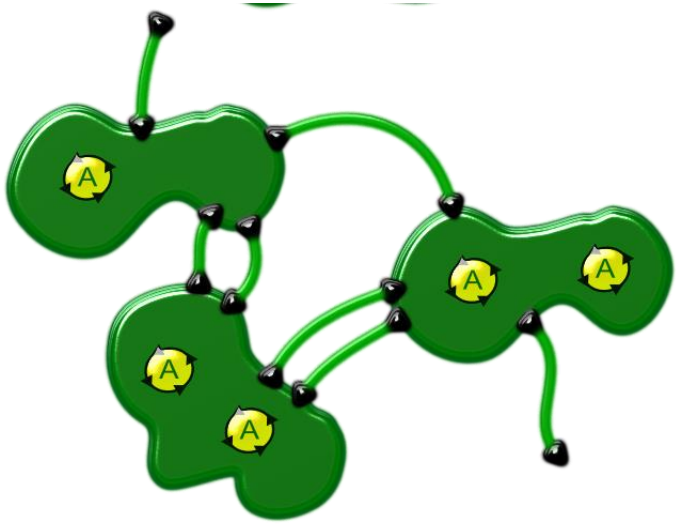
Consequences of the approach

- No global memory
- No processor sharing
- No pecularities of specific processor
- No predefined topology (NoC)
- No interrupts

➔ **No operating system**

Active Cells Computing Model

- Distributed system in the small
- Computation units: "Cells"
- Different parallelism levels addressed by
  - Communication Structure (Pipelining, Parallel Execution)
  - Cell Capabilities (Vector Computing, Simultaneous Execution)
- Inspired by
  - Kahn Process Networks
  - Dataflow Programming
  - CSP
  - ..



Active Cell Components

- Active Cell
  - Object with private state space
  - Integrated control thread(s)
  - Connected via channels
- Cell Net
  - Network of communication cells

Active Cells

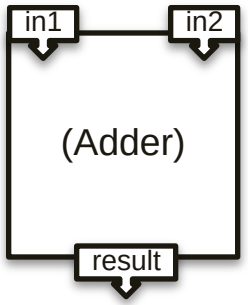
- Scope and environment for a running isolated process.
- Cells do not immediately share memory
- Defined as types with port parameters

communication ports

```
type
    Adder = cell (in1, in2: port in; result: port out);
    var summand1, summand2: integer;
    begin
        in1 ? summand1;
        in2 ? summand2;
        result ! summand1 + summand2
    end Adder;
```

blocking receive

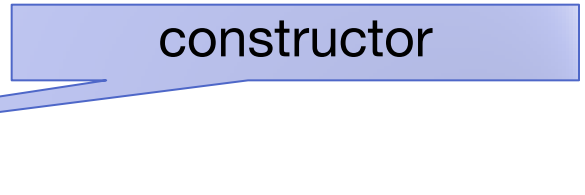
non-blocking send



Cell Constructors

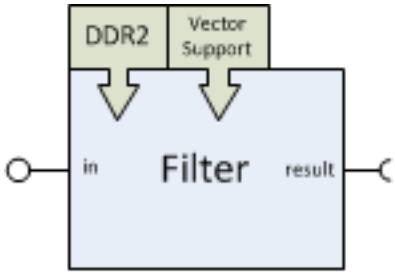
- Constructors to parametrize cells during allocation time

```
type
    Filter = cell (in: port in; result: port out);
    var ...; filterLength: integer;

        procedure & Init(filterLength: integer)
        begin self.filterLength := filterLength
        end Init;

    begin
        (* ... filter action ... *)
    end Filter;

var filter: Filter;
begin

.... new(filter, 32); (* initialization parameter filterlength = 32 *)
```



Further Configurations:Cell Capabilities

- Cells can be parametrized further, being provided with further capabilities or non-default values.



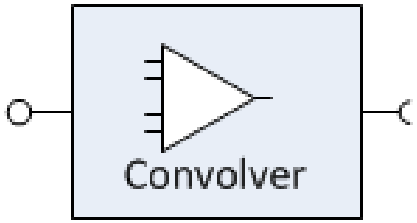

```
type
    Filter = cell {Vector, DataMemory(2048), DDR2}
                  (in: port in (64); result: port out);
    var ...

    begin
        (* ... filter action ... *)
    end Filter;
....
```

Cell is a VectorTRM with 2k of Data Memory and has access to DDR2 memory

This port is implemented with a (bit-)width of 64

Engine Cell Made From Hardware

- Special cells are provided as prefabricated hardware components (***Engines***).




```
type
    Convolver2d= cell {Engine}
                    (in: port in (64); result: port out);

    end Convolver2d;
```

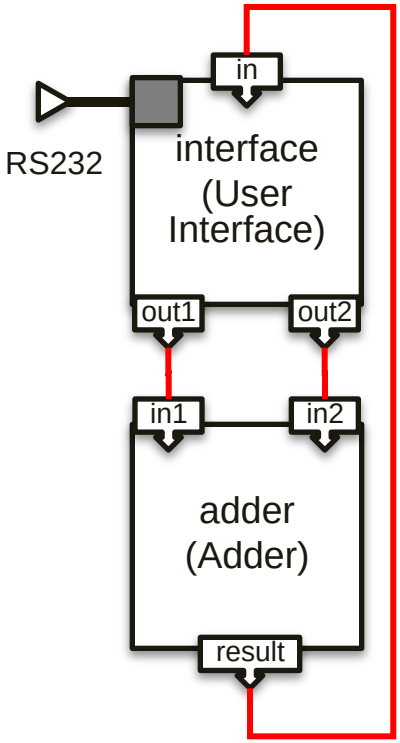
Hierarchic Composition: Cell Nets

- Cellnets consist of a set of cells that can be connected over their ports.
  - Allocation of cells: **new** statement
  - Connection of cells: **connect** statement
- Cellnets can **provide ports**, ports of cells can be **delegated** to the ports of the net
  - Delegation of cells: **delegate** statement
- *Terminal* (or *closed*) Cellnets* can be deployed to hardware

*i.e. Cellnets without ports

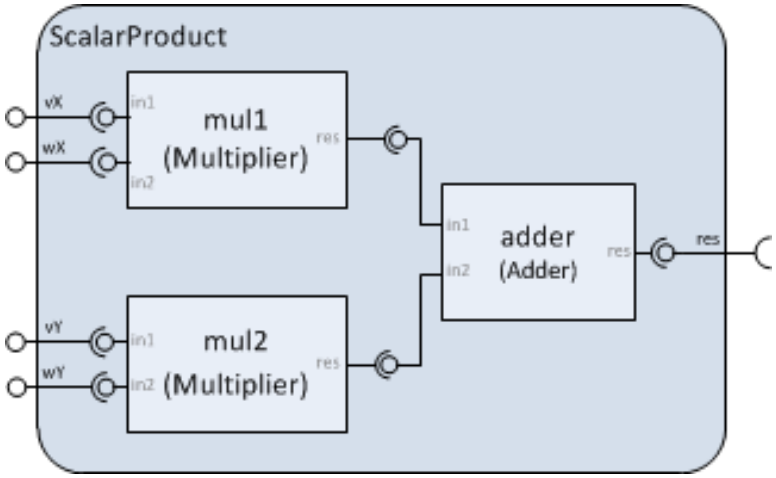
Terminal Cellnet Example

```
cellnet Example;
import RS232;
type
    UserInterface = cell {RS232}(out1, out2: port out; in:
    port in)
    (*...*) end UserInterface;

    Adder = cell(in1, in2: port in; out: port out)
    (* ... *) end Adder;

var interface: UserInterface; adder: Adder
begin
    new(interface);
    new(adder);
    connect(interface.out1, adder.in1);
    connect(interface.out2, adder.in2);
    connect(adder.result, interface.in);
end Example.
```



Hierarchic Composition Example

```
module SimpleCells
import RS232;
type
    Adder = cell (in1, in2: port in; result: port out)
    (* ... *) end Adder;

    Multiplier = cell (in1, in2: port in; result: port out)
    (* ... *) end Adder;
```

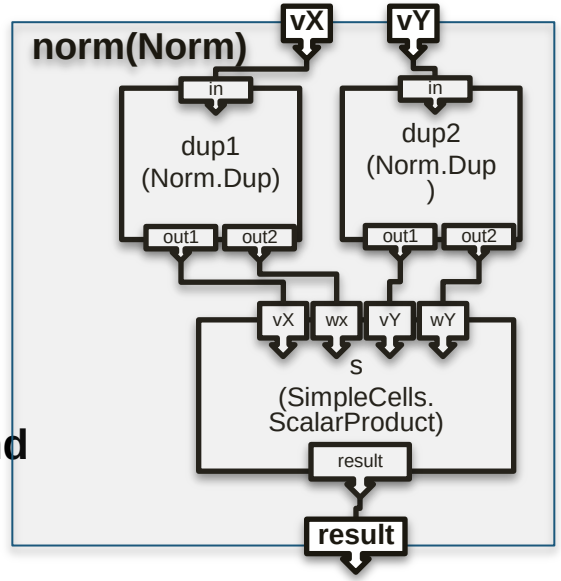



```
    ScalarProduct*= cellnet (vx,vy,xw,xy: port in; result: port out)
    var adder: Adder; multiplier1, multiplier2: Multiplier;
    begin
        new(mul1); new(mul2); new(adder);
        delegate(vx, mul.in1); delegate(wx, mul1.in2);
        delegate(vy, mul2.in1); delegate(wy, mul2.in2);
        connect(mul1.result, adder.in1); connect(mul2.result, adder.in2);
        delegate(result, adder.result)
    end ScalarProduct;

end SimpleCells
```

port delegation

Example of a wired Cellnet

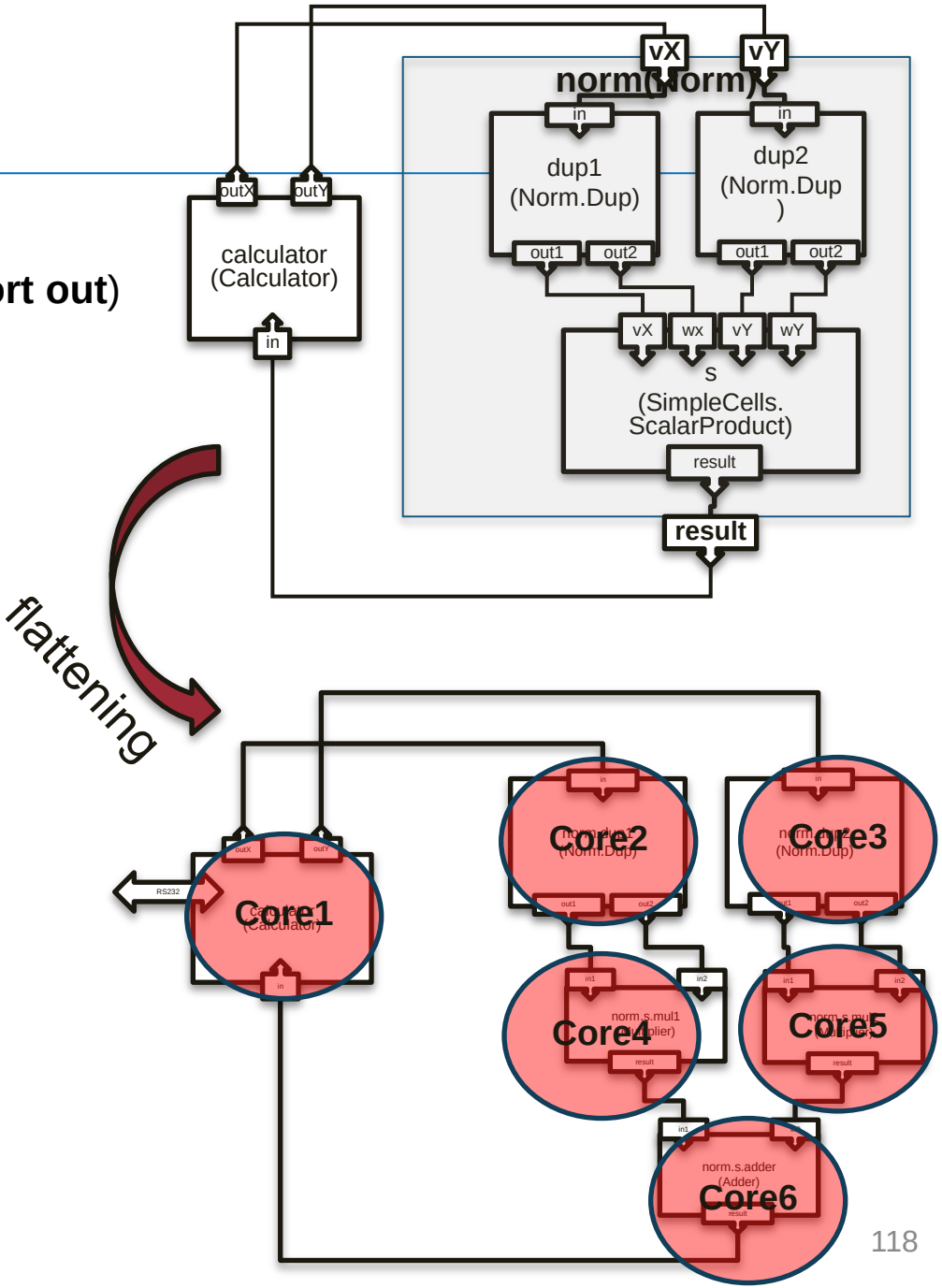
```
cellnet Test;
import SimpleCells, RS232;
type
   Norm*=cellnet (vX,vY: port in; result: port out)
   type
      Dup*=cell(in: port in; out1,out2: port out)
      var val: LONGINT;
      begin
         loop receive(in, val); send(out1, val); send(out2, val) end
      end Dup;

   var s: SimpleCells.ScalarProduct2d; dup1, dup2: Dup;
   begin
      new(s); new(dup1); new (dup2);
      connect (dup1.out1,s.vX); connect(dup1.out2,s.wX);
      connect(dup2.out1,s.vY); connect(dup2.out2,s.wY);
      delegate(vX,dup1.in);delegate(vY,dup2.in);
      delegate(result,s.result);
   end Norm;
```



Flattening

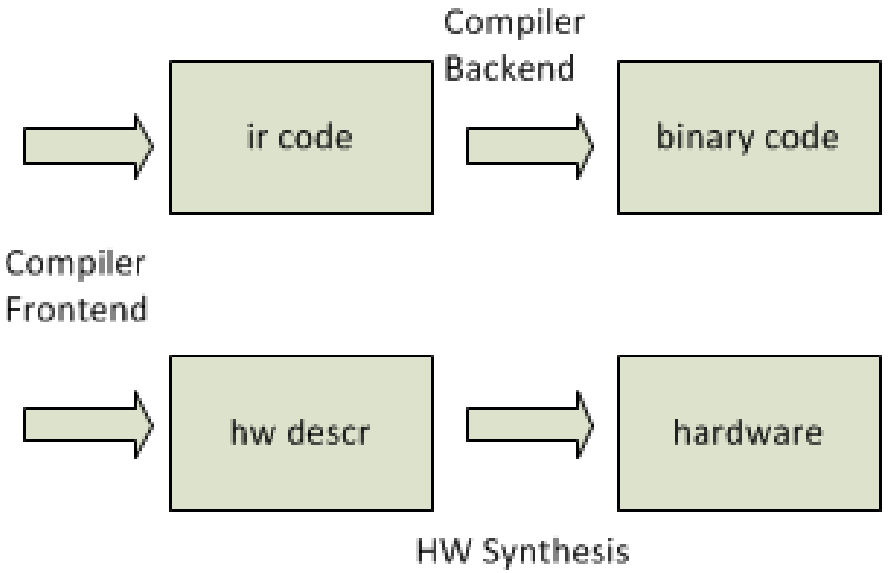
```
Calculator*=cell {RS232} (in: port in;
                                  outX,outY: port out)
var result: longint; vX,vY,wX,wY: longint;
begin
    loop
          RS232.ReceiveInteger(vX);
          RS232.ReceiveInteger(vY);
          send (outX,vX); send(outY,vY);
          receive (in,result);
          RS232.SendInteger(result);
    end;
 end Calculator;
```

```
var calculator: Calculator; norm:Norm;
begin
    new(calculator); new(norm);
    connect(calculator.outX,norm.vX);
    connect(calculator.outY,norm.vY);
    connect(norm.result,calculator.in);
end Test.
```



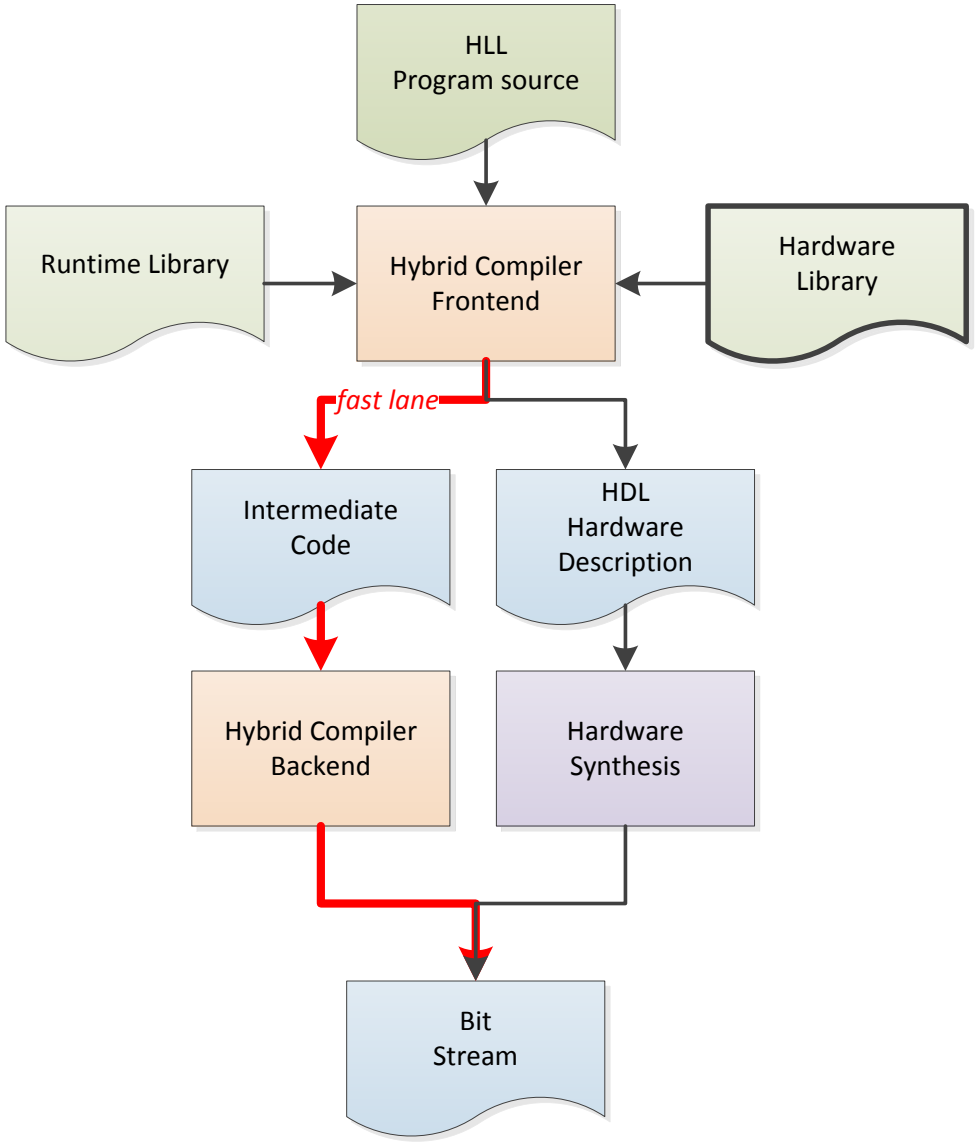
Hybrid Compilation

```
cellnet N;

type A=cell(pi: port in; po: port out);
var x: integer;
begin
… pi ? x; … po ! x; …
end A;

var a,b: A;
begin
… connect(a.po, b.pi)
end N.
```



Code body	Role	Compilation method
Cell (Softcore)	Program logic	Software Compilation
Cell (Engine)	Computation unit	Hardware Generation
Cell Net	Architecture	Hardware Compilation

Automated Mapping to FPGA



Hardware Library

Computation Components

- General purpose minimal machine: TRM, FTRM
- Vector machine: VTRM
- MAC, Filters etc.

Storage Components

- DDR2 controller
- configurable BRAMs
- CF controller

Communication Components

- FIFOs
  - 32 * 128
  - 512 * 128
  - 32, 64, 128, 1k * 32

I/O Components

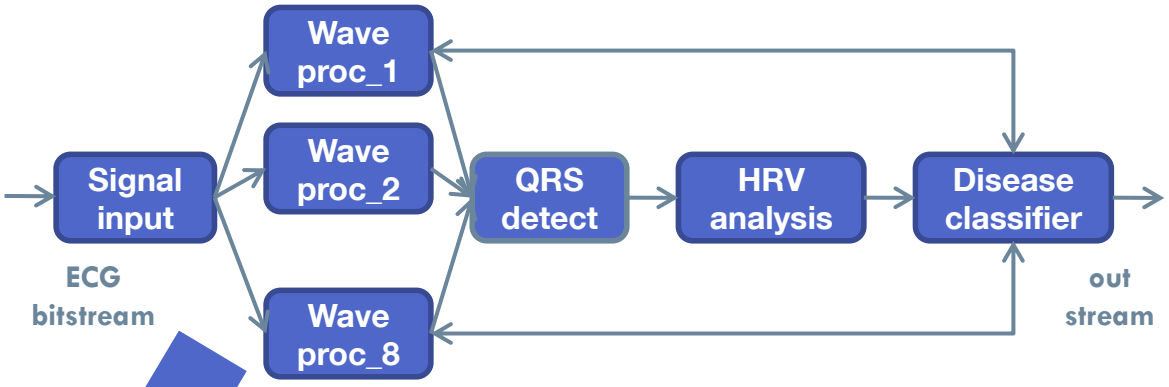
- UART controller
- LCD, LED controller
- SPI, I2C controller
- VGA, DVI controller

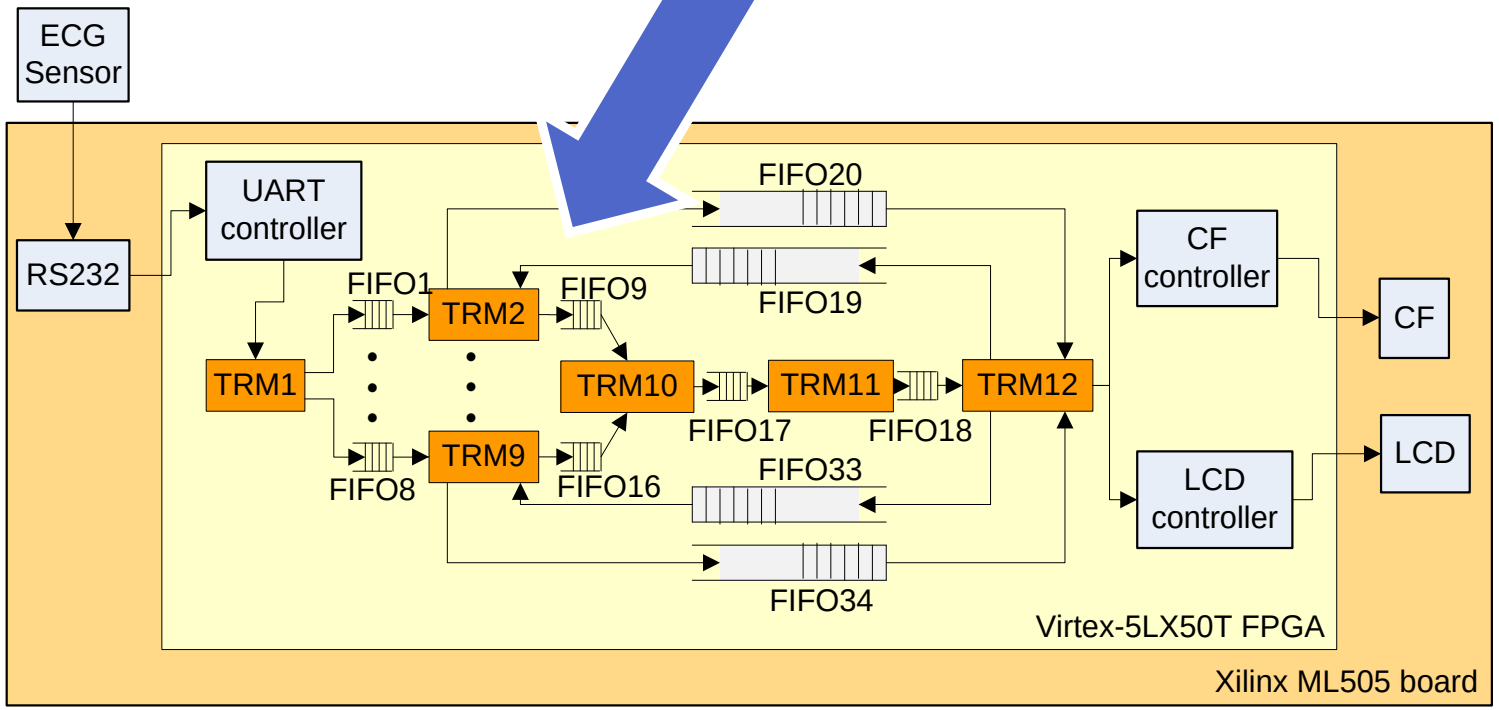
Case Study 1: ECG

Focus: Resources and Power

Real-time

ECG Monitor





Resources

- ECG Monitor*

#TRMs	#LUTs	#BRAMs	#DSPs	TRM load
12	13859 (48%)	52 (86%)	12 (25%)	<5% @116 MHz

- Maximum number of TRMs in communication chain

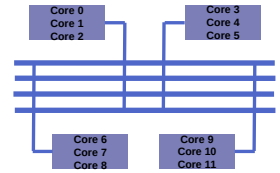
FPGA	#TRMs	#LUTs	#BRAMs	#DSPs
Virtex-5	30	27692 (96%)	60 (100%)	30 (62%)
Virtex 6	500			

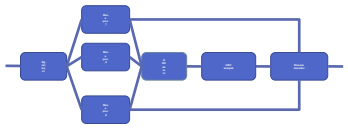
*8 physical channels @ 500 Hz sampling frequency implemented on Virtex 5

123

Comparative Power Usage

- Preconfigured FPGA (#TRMs, IM/DM, I/O, Interconnect fixed)
  versus fully configurable FPGA (Active Cells)

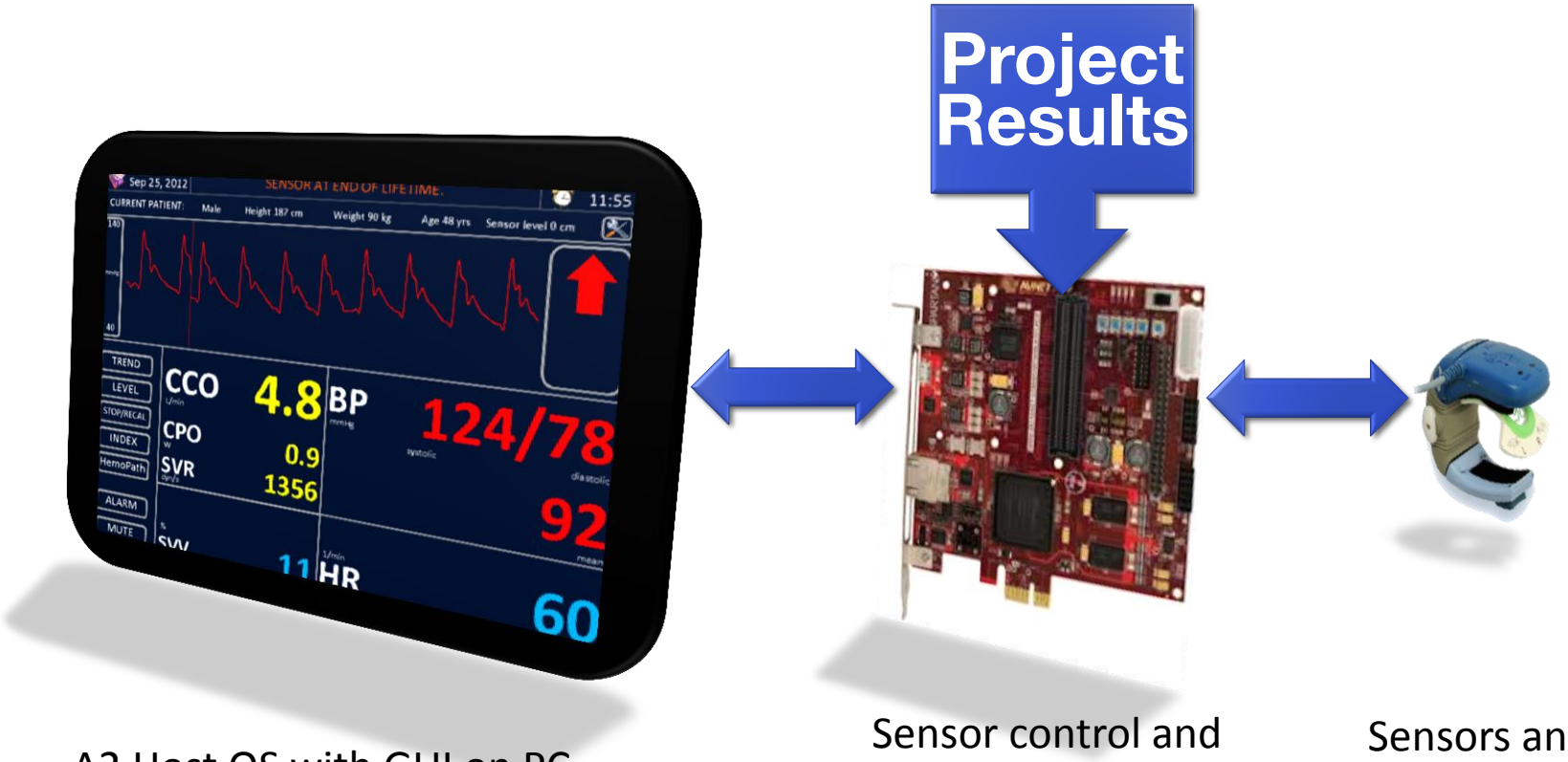




System	Static Power (W)	Dynamic Power (W)
Preconfigured ("TRM12")	**3.44**	**0.59**
Dynamically configured	**0.5**	**0.58**

86% saving!

Case Study 2: Non-Invasive Continuous Blood Pressure Monitor
Focus: Development Cycle Time

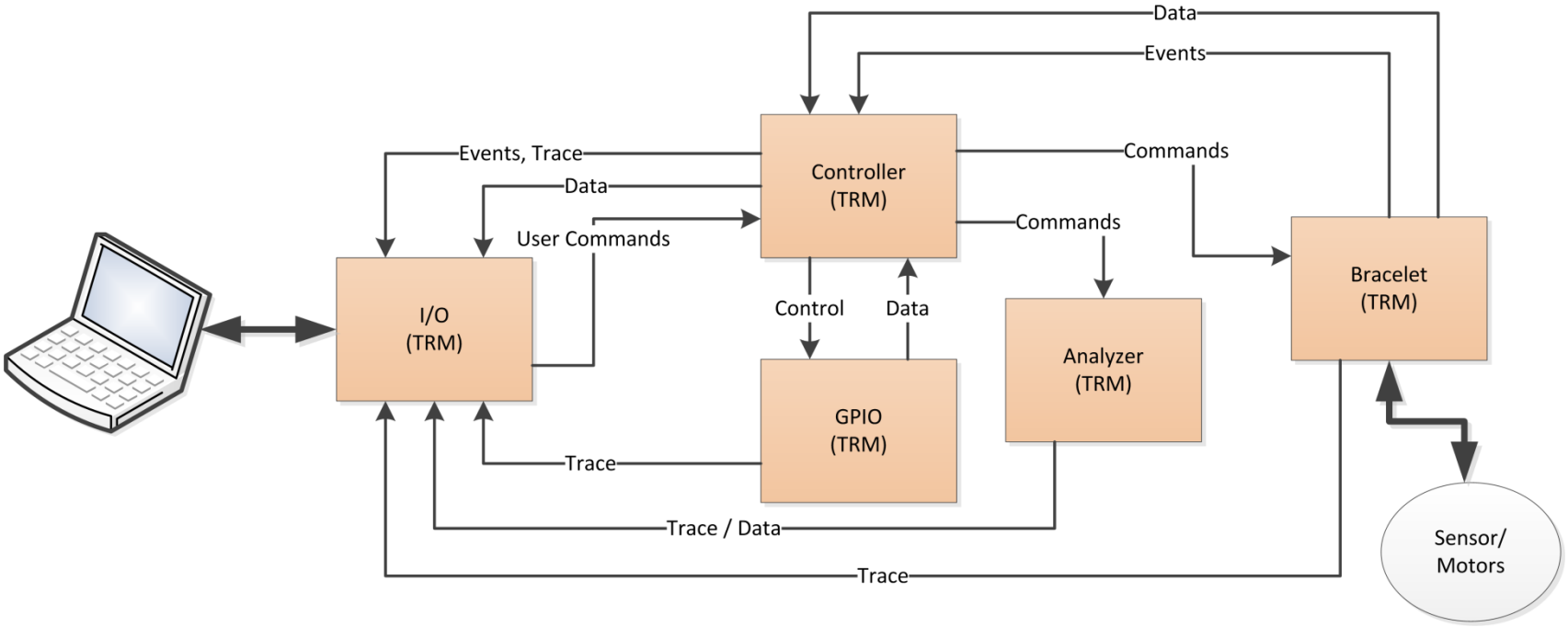


A2 Host OS with GUI on PC

Sensor control and medical algorithms on Spartan 6 FPGA

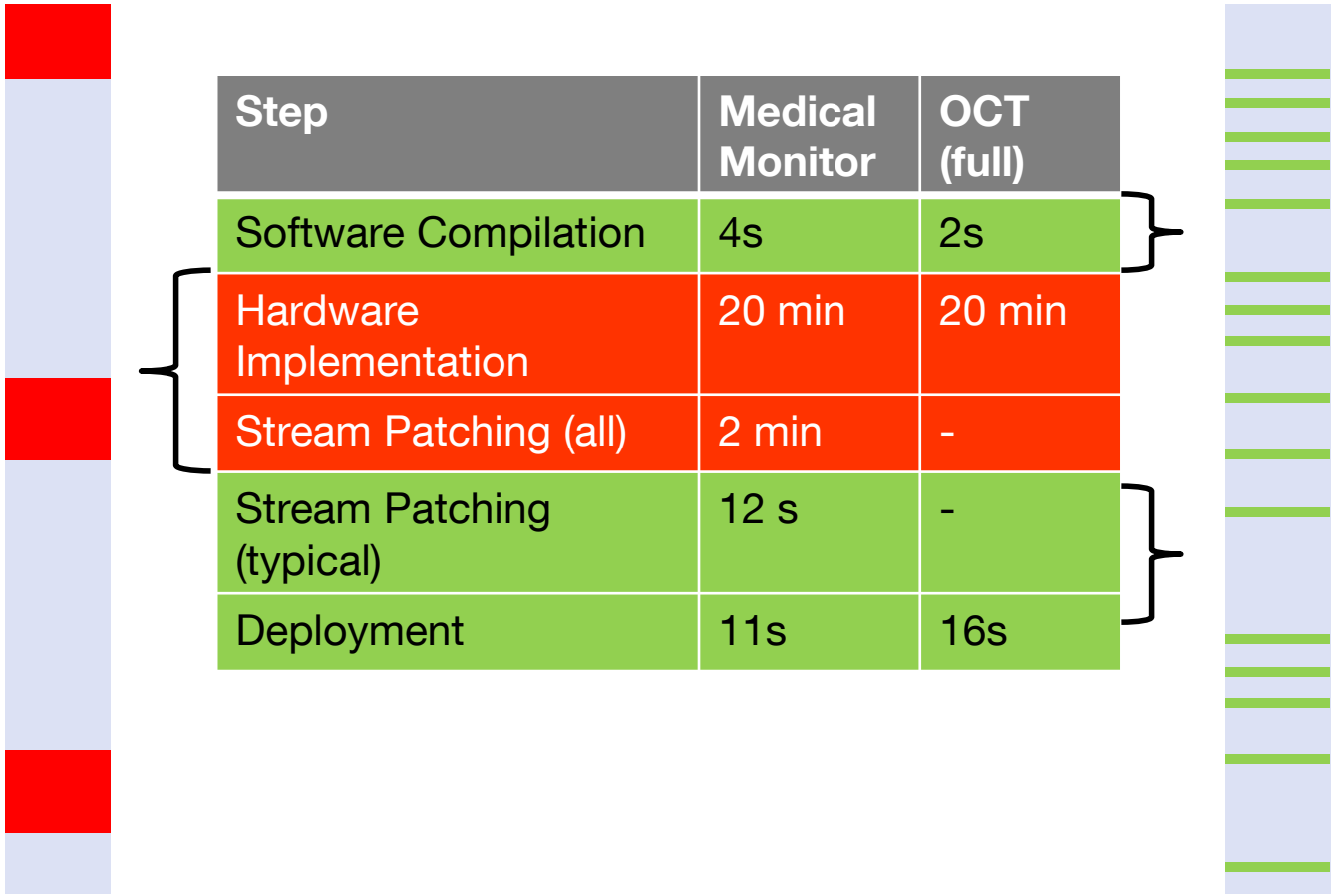
Sensors and Motors on Bracelet

Medical Monitor Network On Chip



Dominated by TRM processors. Feedback driven.

Development Cycle Times



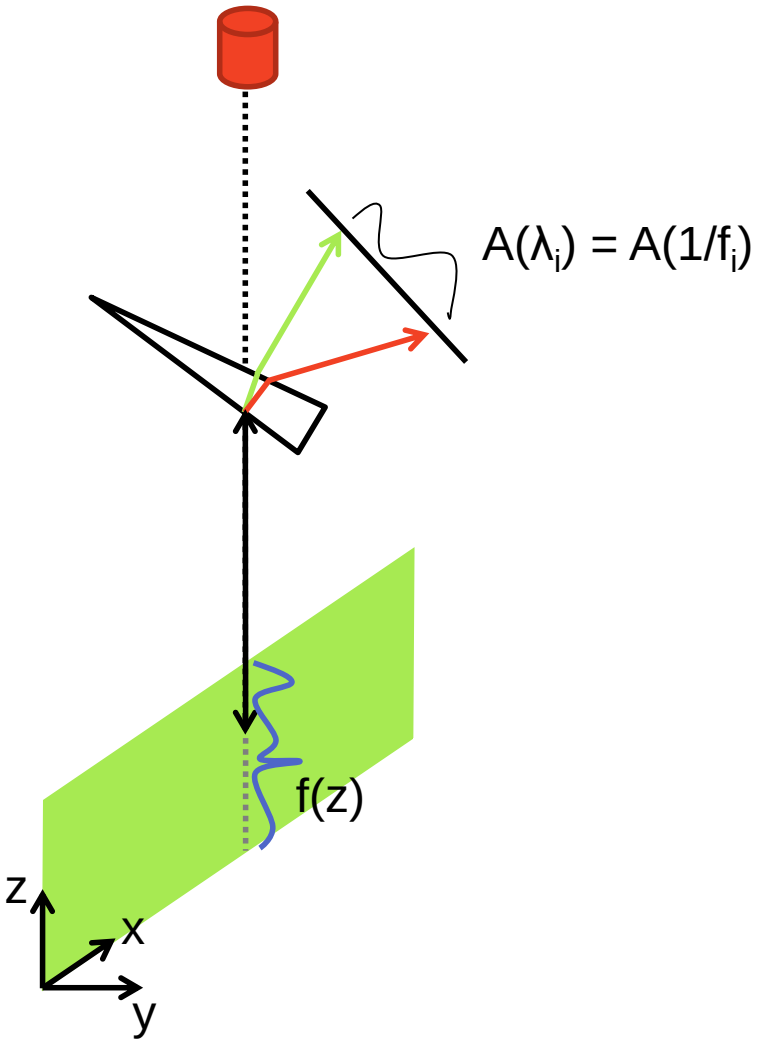
Step	Medical Monitor	OCT (full)
Software Compilation	4s	2s
Hardware Implementation	20 min	20 min
Stream Patching (all)	2 min	-
Stream Patching (typical)	12 s	-
Deployment	11s	16s

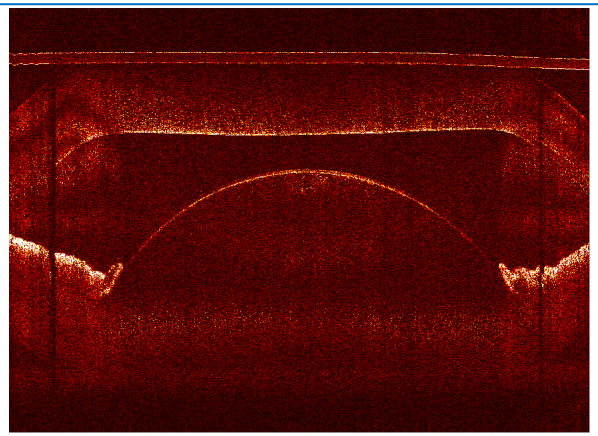
sporadic

often

Case Study 3: Optical Coherence Tomography

Focus: Performance





z-Axis Processing

1. Non uniform sampling

   $A(\lambda_i) \rightarrow \tilde{A}(f_i)$

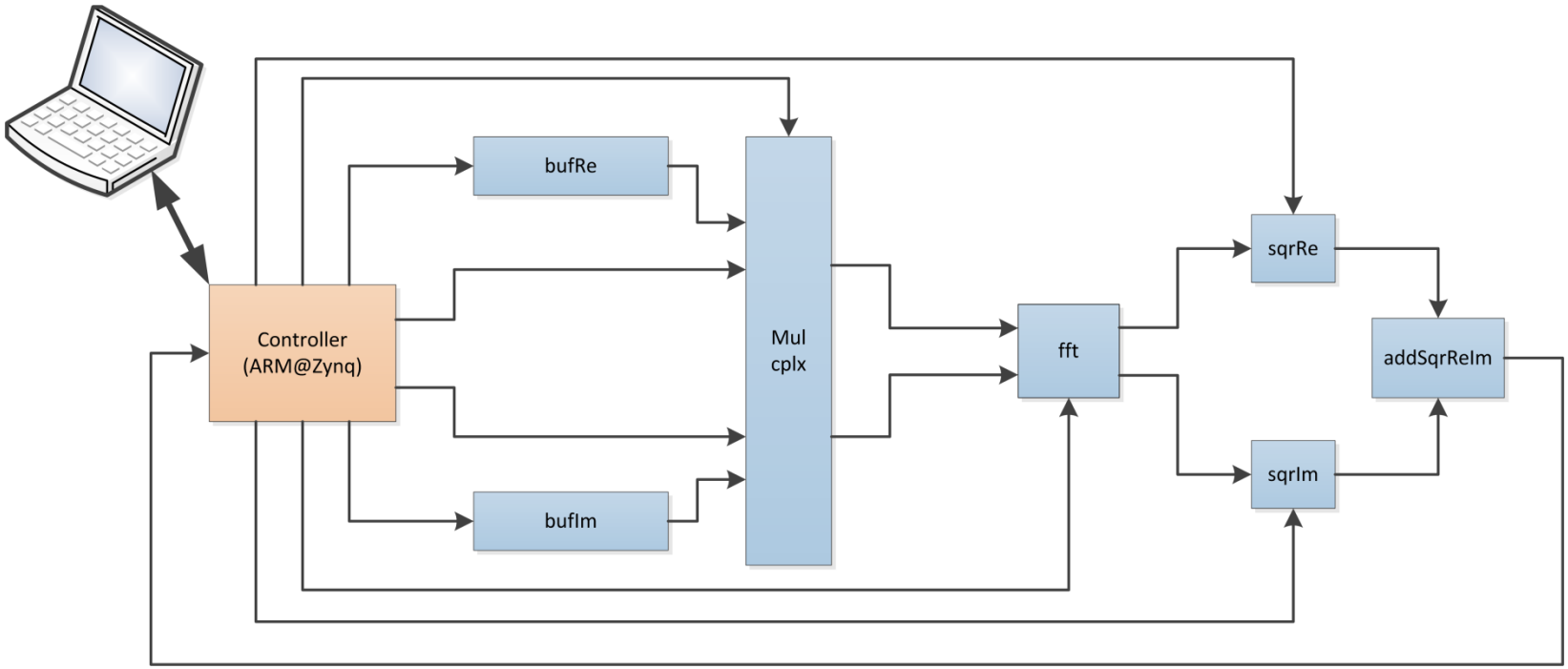
2. Dispersion compensation

3. (Inverse) FFT

… for many lines x in a row (2d)
… and many rows y in a column (3d)

A component of OCT image processing

Dispersion Compensation



Dominated by Engines. Dataflow driven.

Performance and Resource Usage

	Medical Monitor	Dispersion Compensation	OCT
Architecture	Spartan 6 XC6SLX75	Zynq 7000 XC7Z020	Zynq 7000 XC7Z020
Resources	28% Slice LUTs, 4% Slice Registers 80% BRAMs 24% DSPs	11% Slice LUTs, 6% Slice Registers 7% BRAMs 15% DSPs 1 ARM Cortex A9	17% Slice LUTS 8% Slice Registers 22% BRAMs 31% DSPs 1 ARM Cortex A9
Clock Rate	58 MHz	118 MHz	50 MHz
Performance	--	**8.3 GFPOps*** *up to 32 GFPops***	**4.3 GFPOps***
Data Bandwidth	1.25 Mbit /s (in) 23 kB/s (out)	236 MWords/s (in) 118 MWords/s (out)	50 MWords/s (in) 50 MWords/s (out)
Power	~2W	~5W	~5W

** Fixed point operations, 32bit
* when instantiated 4 times

Conclusion

ActiveCells: Computing model and tool-chain for emerging configurable computing

- Configurable interconnect → Simple Computing, Power Saving
- Hybrid compilation→ Decreased Time to Market
- Embedding of task engines → High Performance

Spartan3e Board used in the labs

Resources

4-input LUTS	9312
Flip Flops	9312
BRAMs	20
DSPs	-

Resource Usage Scenarios

1 TRM

```
CELLNET Simple;

IMPORT LED;

TYPE

  Trm=CELL {LED,
            CodeMemorySize=1024,
            DataMemorySize=512};
  BEGIN
     LED.Show({1,3,5});
    LOOP
    END
  END Trm;

VAR trm: Trm;
BEGIN
   NEW(trm);
END Simple.
```

Resources

4-input LUTS	12%
Flip Flops	2%
BRAMs	10%

2 TRMs

```
TYPE
  Trm1=CELL {
            CodeMemorySize=1024,
            DataMemorySize=512}
            (out: PORT OUT);
  ...
  END Trm1;

  Trm2=CELL {LED,
            CodeMemorySize=1024,
            DataMemorySize=512}
            (in: PORT IN);
  ...
  END Trm2;

VAR trm1: Trm1; trm2: Trm2;
BEGIN
  NEW(trm1);
  NEW(trm2);
  CONNECT(trm1.out, trm2.in);
```

Resources

4-input LUTS	24%
Flip Flops	5%
BRAMs	20%

Ring of 7 TRMs

Resources

4-input LUTS	92%
Flip Flops	19%
BRAMs	70%

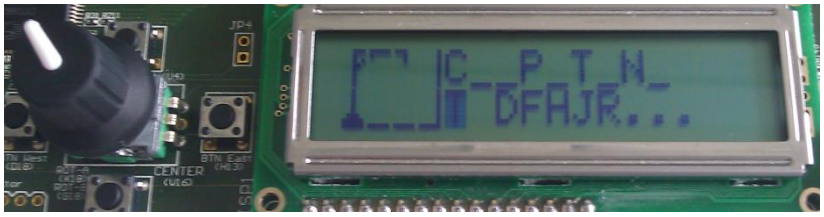
2 TRMs with 10 Fifos

Resources

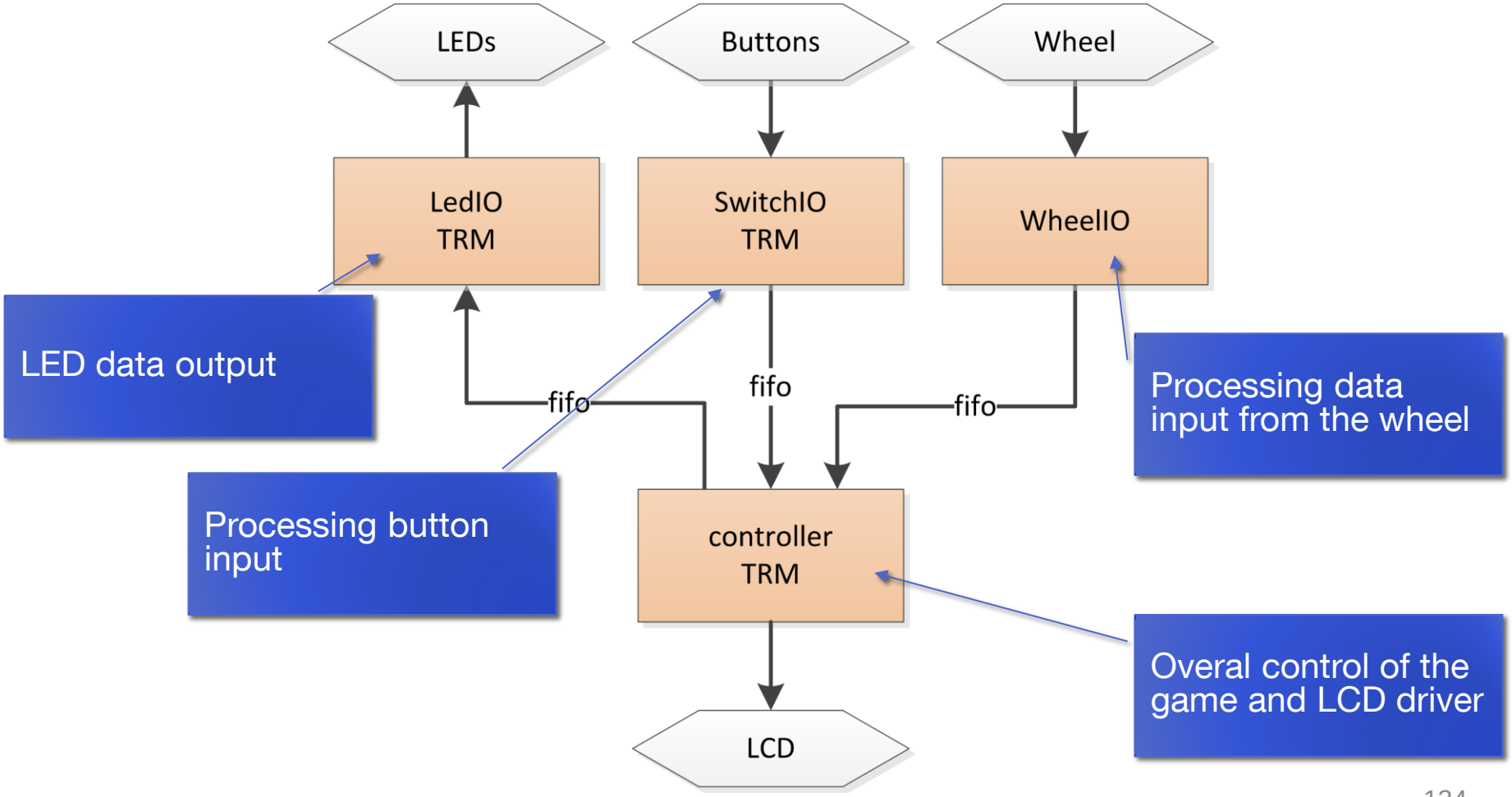
4-input LUTS	45%
Flip Flops	6%
BRAMs	20%

➔ TRM ≈ 12% (LUTs), 10% (BRAMs); Fifo ≈ 2% (LUTs)

Hangman Game



- Components



Hangman Game CellNet

```
CELLNET Game;

...

VAR controller: Controller; switches: SwitchIO;
wheel: WheelIO; leds: LedIO;
BEGIN
      NEW(controller);
      NEW(switches);
      NEW(leds);
      NEW(wheel);
      CONNECT(controller.leds, leds.in);
      CONNECT(switches.out, controller.switches);
      CONNECT(wheel.out, controller.wheel);
END Game.
```

LED cell

```
(* Leds output *)
LedIO*=CELL {LED} (in: PORT IN);
VAR value: SET;
BEGIN
  LED.Show({});
  LOOP
    in ? value;
    LED.Show(value);
  END;
END LedIO;
```

Obviously too simple to waste a full TRM

Switch IO

```
(* Switches Input  *)
SwitchIO*=CELL {Switch} (out: PORT OUT);
VAR recent, on: SET;
BEGIN
  recent := {};
  LOOP
    (*! there is chatter on the buttons -- repair this *)
    REPEAT
      Switch.Get(on);
      on := on * {0,1,2,3}; (* mask out undefined bits *)
    UNTIL recent # on;
    recent := on;
    out ! on;
  END;
END SwitchIO;
```

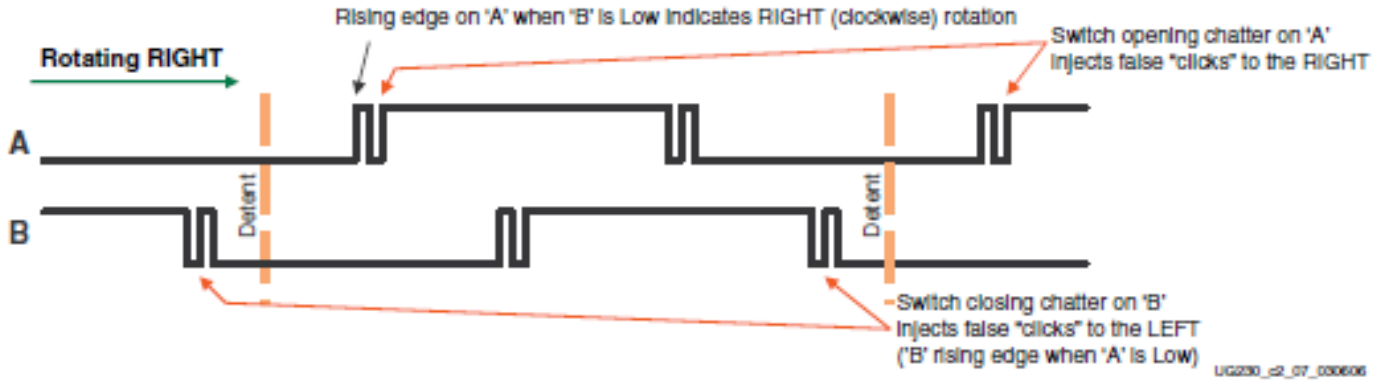
Still quite simple, but helps to avoid polling loop on other cores

Wheel IO

```
WheelIO*= CELL {Wheel} (out: PORT OUT);
VAR wheel, recent:SET;
BEGIN
  LOOP
     (*! this code is wrong -- replace by correct code *)
     REPEAT Wheel.Get(wheel) UNTIL wheel # recent;
     recent := wheel;
     IF 0 IN wheel THEN out ! 1
     ELSIF 1 IN wheel THEN out ! 0
     END;
  END;
END WheelIO;
```

Substantial complexity,

still typical driver that might be better done in hardware …



Controller

```
Controller*=CELL {LCD} (switches, wheel: PORT IN; leds: PORT OUT)
VAR
  badCount, offset: LONGINT;
  (* ... *)
BEGIN
  (* ... *)
  LOOP
    IF wheel ?? int THEN (* wheel input from wheel component *)
       PrintLetter;
    ELSIF switches ?? set THEN (* button input from io component *)
      IF 3 IN set THEN (* Wheel pushed -- enter character  *)
        Enter(letter)
      ELSIF 1 IN set THEN (*South button pressed -> toggle cursor *)
      (* ... *)
    END
  END
END Controller;
```

non-blocking receive

Control Logic and LCD Protocol – quite complex and definitely a task for a TRM

Workshop on System-on-Chip Design

Part III : Hybrid Compilation and Future Directions

National Chiao Tung University, 21.-23.10.2013

Felix Friedrich, ETH Zürich

Objectives of Part II

- The compiler toolchain: Hybrid compilation
- Software-/Hardware Map
- Active Cells #
- Active Cells **2**

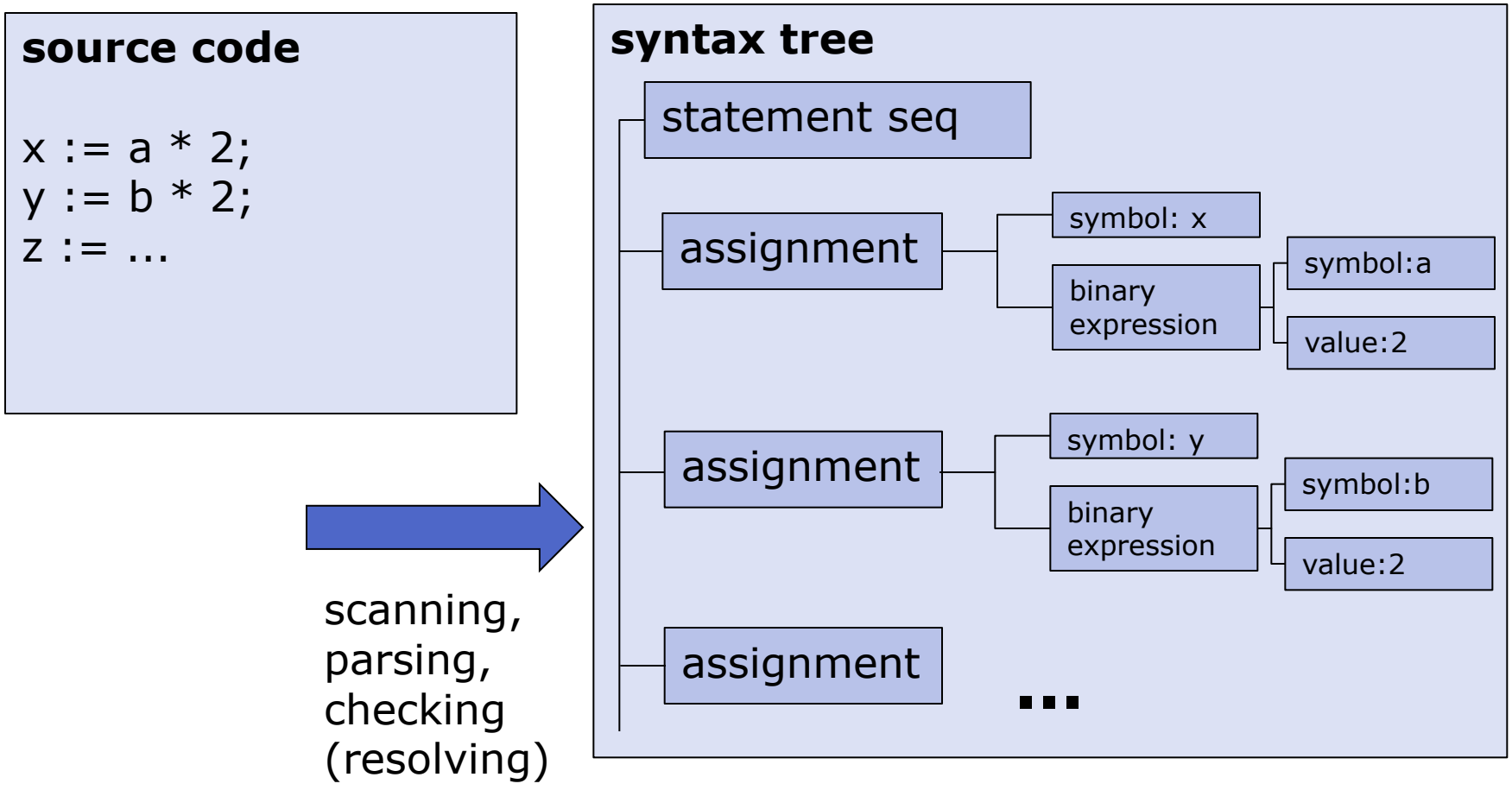
Behind the Scenes of Hybrid Compilation

TRM: Separate Compilation

- Development environment with separate compilation is
  - important for commercial products
  - state of the art
- Very limited architecture (18bit instructions, 10-bit immediates, 7-bit memory offset, 10bit (signed) conditional branch offset, 14-bit (+/- 8k) Branch&Link
  - long jumps (chained)
  - immediates -> expressed by several instructions or put to memory (-> fixups)
  - fixups, problematic if large global variables are allocated
  - far procedure calls
- Solution: Compilation to Intermediate Code, backend application at link time.

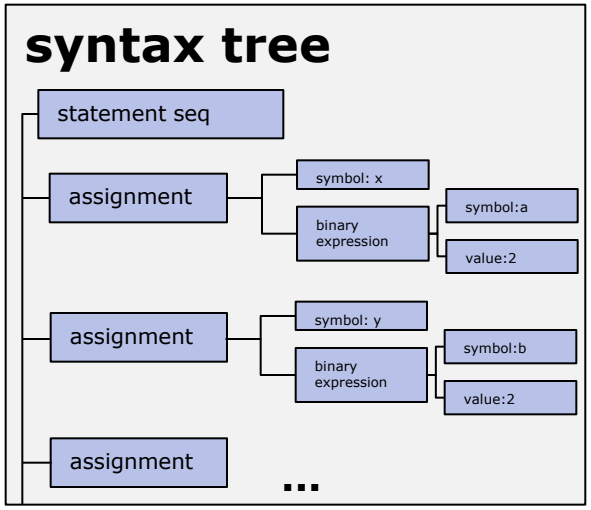
The Compilation Process

Syntax Tree Generation



The Compilation Process

Intermediate Code Generation



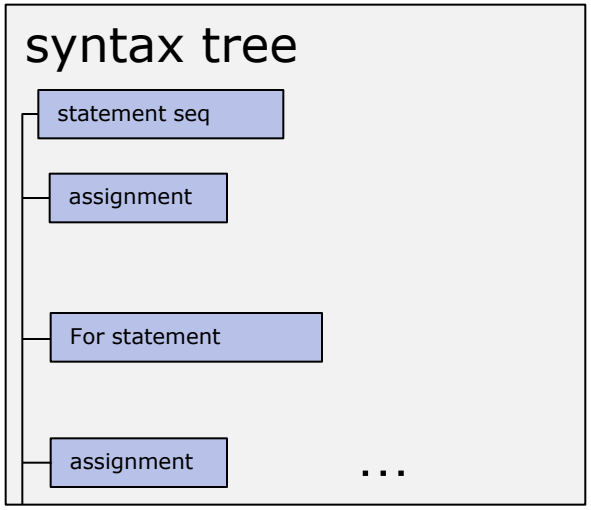
syntax tree traversal

intermediate code

```
.module A
 ....

.code A.P
  ....
  1:   mul s32 [A.x:0], [A.a:0], 2
  2:   mul s32 [A.y:0], [A.y:0], 2
   ....
```

The Compilation Process

Interpretation

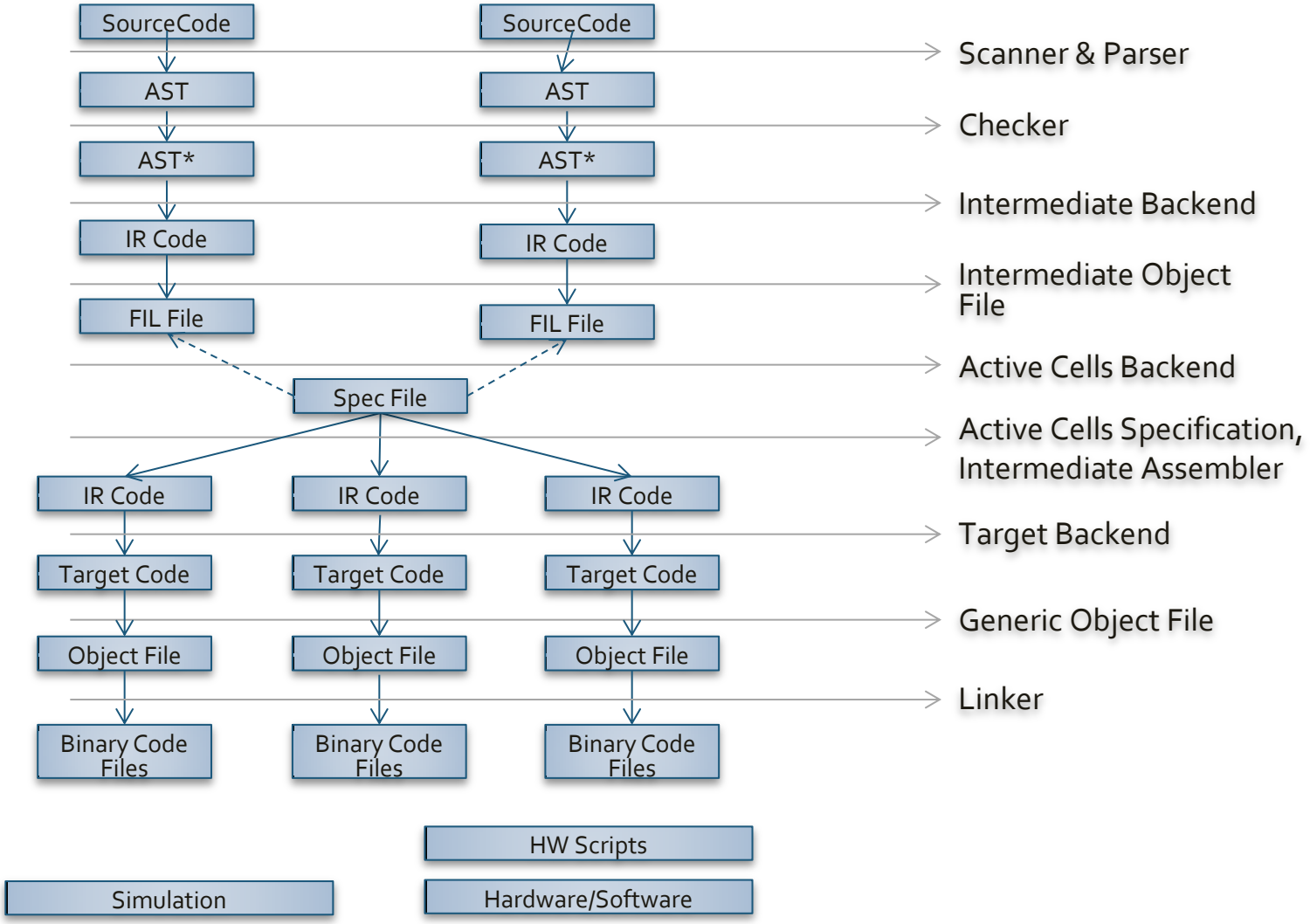


syntax tree traversal

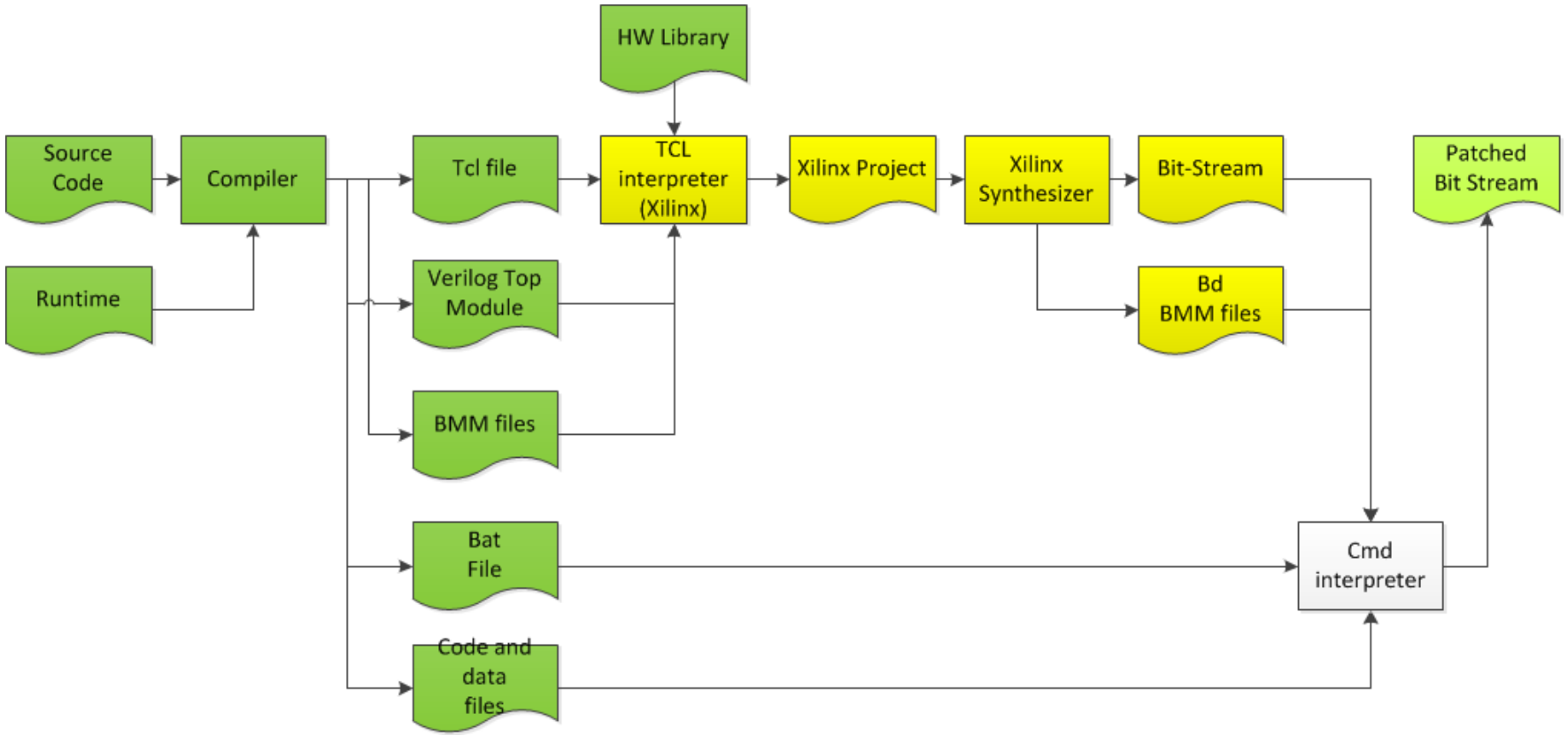
e.g. Active Cells specification

```
name=Game
  instructionSet=TRM
types=2
    0 name=Display
modules=6
        0 name=Timer filename=''
        1 name=Button filename=''
        2 name=LED filename=''
        3 name=TRMRuntime filename='
....
```

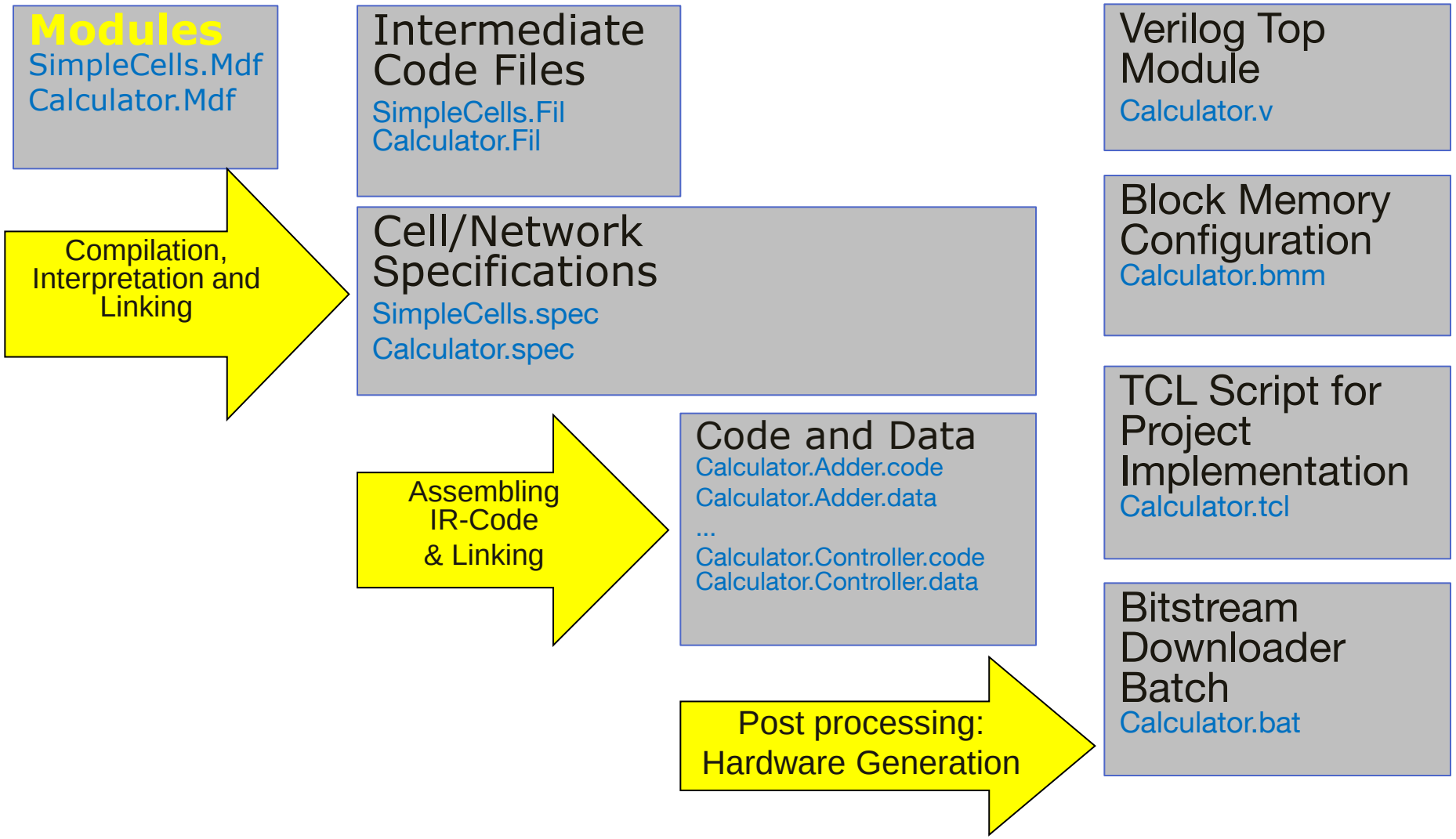
Active Cells Code Generation



Tool-Chain Steps in Detail



Perspective of the compiler



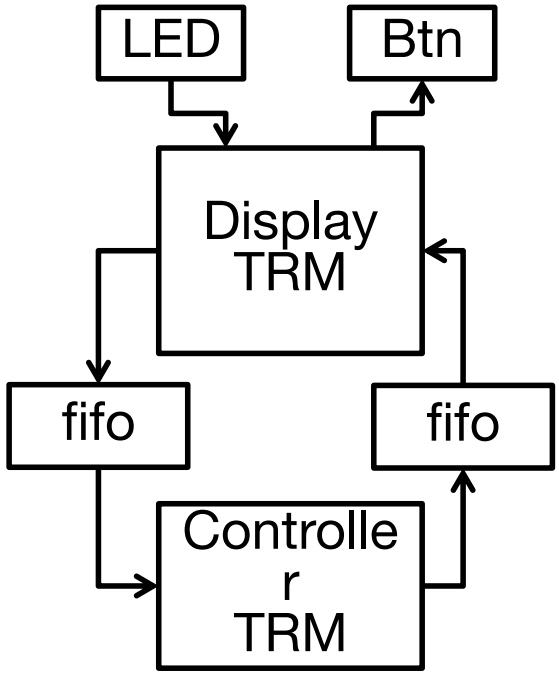
Mapping to Hardware

Simple Example Code

Display*=CELL {LED, Button} (in: PORT IN; out: PORT OUT);
BEGIN
END Display;

Controller*=CELL (in: PORT IN; out: PORT OUT)
BEGIN
END Controller;

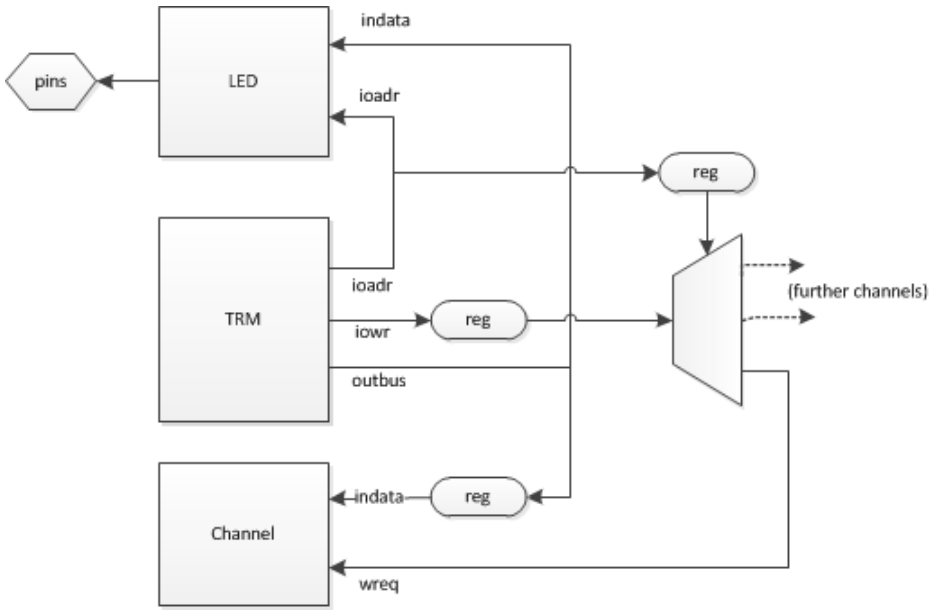
VAR controller: Controller; display: Display;
BEGIN
NEW(controller); NEW(display);
CONNECT(controller.out, display.in);
CONNECT(display.out, controller.in);
END Game.



Mapping to hardware
TRM outbound communication

Parts of the generated top module (some signals omitted)

```
TRM  #(.IMB(1), .DMB(4)) Game0display
(.clk(clk), .rst(rst), .irq0(1'b0), .irq1(1'b0), .stall(1'b0), .inbus(Game0display_inbus), .ioadr(Game0display_ioadr),
.iowr(Game0display_iowr), .iord(Game0display_iord), .outbus(Game0display_outbus));

LED instGame0Display0LED(.clk(clk), .rst(rst), .inData(Game0Display0LED_inData), .ioadr(Game0display_ioadr),
.iowr(Game0display_iowr), .outData(Game0Display0LED_outData));

assign leds = Game0Display0LED_outData;

ParChannel    #(.Size(32)) Game00Channel1(.clk(clk), .rst(rst), .wreq(Game00Channel1_wreq), .rdreq(Game00Channel1_rdreq),
.inData(Game00Channel1_inData), .outData(Game00Channel1_outData), .status(Game00Channel1_status));

assign Game00Channel1_wreq =
(Game0display_ioadr_reg == 34) & Game0display_iowr_reg;

assign
Game00Channel1_inData = Game0display_outbus_reg;

always @(posedge clk)
begin
  Game0display_iowr_reg <= Game0display_iowr;
  Game0display_ioadr_reg <= Game0display_ioadr;
  Game0display_outbus_reg <= Game0display_outbus;
End

assign
 Game0Display0LED_inData = Game0display_outbus[7:0];
```



Mapping to hardware
TRM inbound communication

Parts of the generated top module

TRM #(.IMB(1), .DMB(4)) Game0display(.clk(clk), .rst(rst), .irq0(1'b0), .irq1(1'b0), .stall(1'b0), .inbus(Game0display_inbus), .ioadr(Game0display_ioadr), .iowr(Game0display_iowr), .iord(Game0display_iord), .outbus(Game0display_outbus));

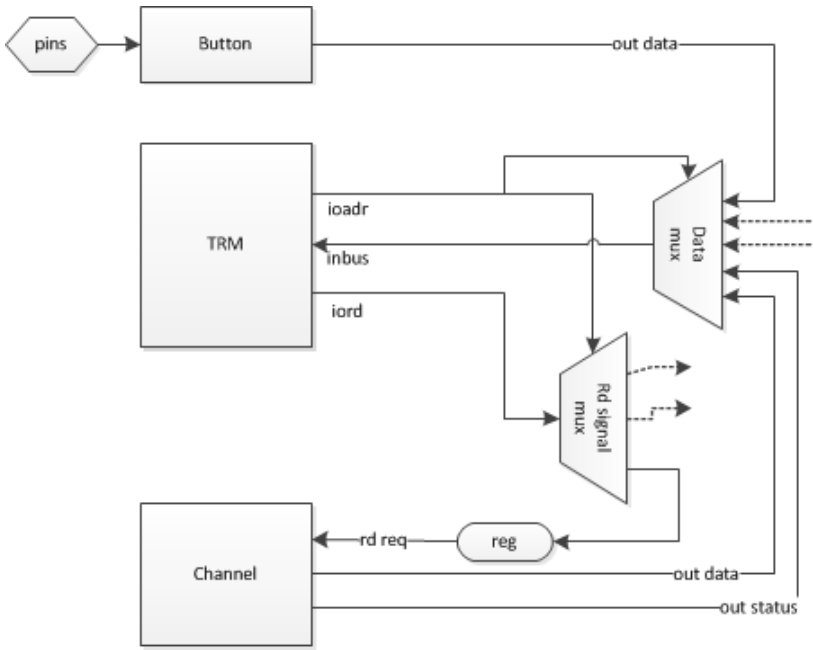
Button instGame0Display0Button(.clk(clk), .rst(rst), .inData(Game0Display0Button_inData), .outData(Game0Display0Button_outData));

ParChannel #(.Size(32)) Game00Channel0(.clk(clk), .rst(rst), .wreq(Game00Channel0_wreq), .rdreq(Game00Channel0_rdreq), .inData(Game00Channel0_inData), .outData(Game00Channel0_outData), .status(Game00Channel0_status));

```
always @(posedge clk)
begin
Game00Channel0_rdreq <= ((Game0display_ioadr == 32) & ~Game00Channel0_rdreq & Game0display_iord);
end
```

```
assign
Game0Display0Button_inData = button;
```

```
assign
Game0display_inbus =
(Game0display_ioadr == 32)? Game00Channel0_outData:
(Game0display_ioadr == 33)? Game00Channel0_status:
(Game0display_ioadr ==  7)? Game0Display0Button_outData:
Game0display_time;
```



Some signals (clk, rst etc.) and Timer component not shown

Mapping to hardware: TCL script

1. Define project name and files

```
set compile_directory  ./
set top_name Top
set hdl_files [ list \
Test.v \
../FIFO/FIFO32.v \
….
../FIFO/ParChannel.v \
../TRM/TRM.v \
…
../IO/Button.v \
../IO/Button.ucf \
]
```

2. Create new project

```
set prj $top_name
cd $compile_directory
file delete -force $prj.ise
file delete -force $prj.xise
puts "Creating a new project ..."
project new $prj.xise
project set family Spartan3
project set device XC3S200
project set package FT256
project set speed -4

foreach filename $hdl_files {
xfile add $filename
 }
```

3. Set compilation parameters

```
if {![catch {set source_directory
$source_directory}]} {
project set "Macro Search Path"
$source_directory -process Translate
}
project set "top" Test
project set "Optimization Goal" Speed
project set "Optimization Effort" High
project set "Keep Hierarchy" Soft
project set "Placer Effort Level" High
project set "Placer Extra Effort" Normal
project set "Register Duplication" true
project set "Optimization Strategy (Cover
Mode)" Speed
project set "Place And Route Mode" "Normal
Place and Route"
project set "Place & Route Effort Level
(Overall)" High
project set "Extra Effort(Highest PAR level
only)" Normal
project set "Other Ngdbuild Command Line
Options" "-bm ./Test.bmm "
```

4. Generate

```
process run "Generate Programming File"
project close
```

Mapping to hardware: Patch file (.bat or .sh)

1. Copy bit stream from recent synthesis

```
del .\df.bit
copy Test.bit .\df0.bit
```

2. Patch generated code and data files into bit stream (using BMM file)

```
data2mem -bm Test_bd.bmm  -bt  .\df0.bit  -bd
Test0controller0code0.mem tag Test0controller_ins0 -o b .\df1.bit
data2mem -bm Test_bd.bmm  -bt  .\df1.bit  -bd
Test0controller0data0.mem tag Test0controller_dat0 -o b .\df2.bit
data2mem -bm Test_bd.bmm  -bt  .\df2.bit  -bd Test0io0code0.mem tag
Test0io_ins0 -o b .\df3.bit
data2mem -bm Test_bd.bmm  -bt  .\df3.bit  -bd Test0io0data0.mem tag
Test0io_dat0 -o b .\df4.bit
copy .\df4.bit .\df.bit
copy ..\download.cmd  .\download.cmd
```

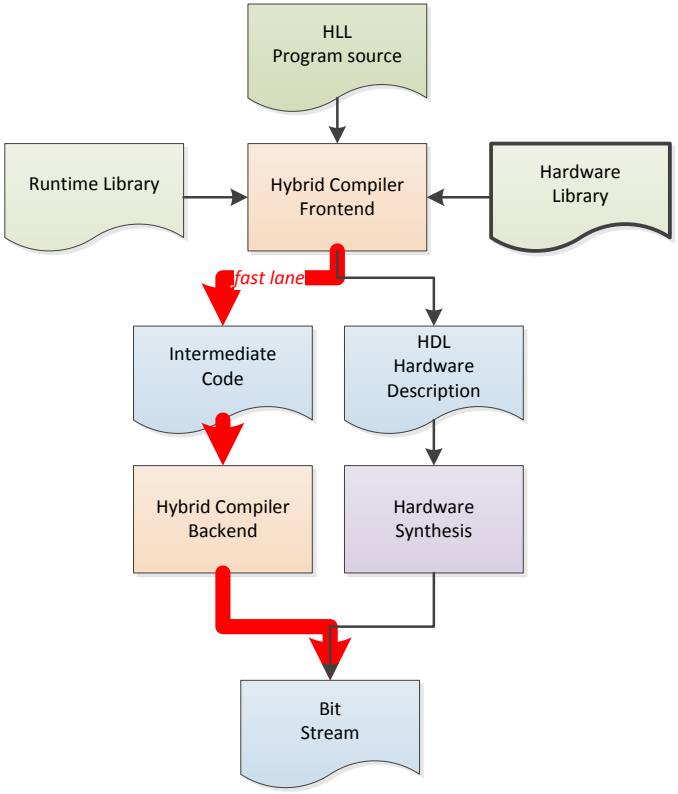
3. Call impact to upload to hardware

```
impact -batch .\download.cmd
```

Automatic Reduction of synthesis / patching tasks

TRMTools.BuildHardware

- Check for existing built-<projname>.spec file.
  If exists check for content equality with current spec file.
- If not existing or not equal (topology modification, modified capabilities, new memory sizes etc.)
  resynthesize hardware by calling tcl file
- Check for existing built-<projname>.code and data files.
  If existing check for equality with current code and data files
- For all non equal or non existing code or data files
  generate data2mem commands in command script
- Add impact command to command script and
  call command script



Today's Lab

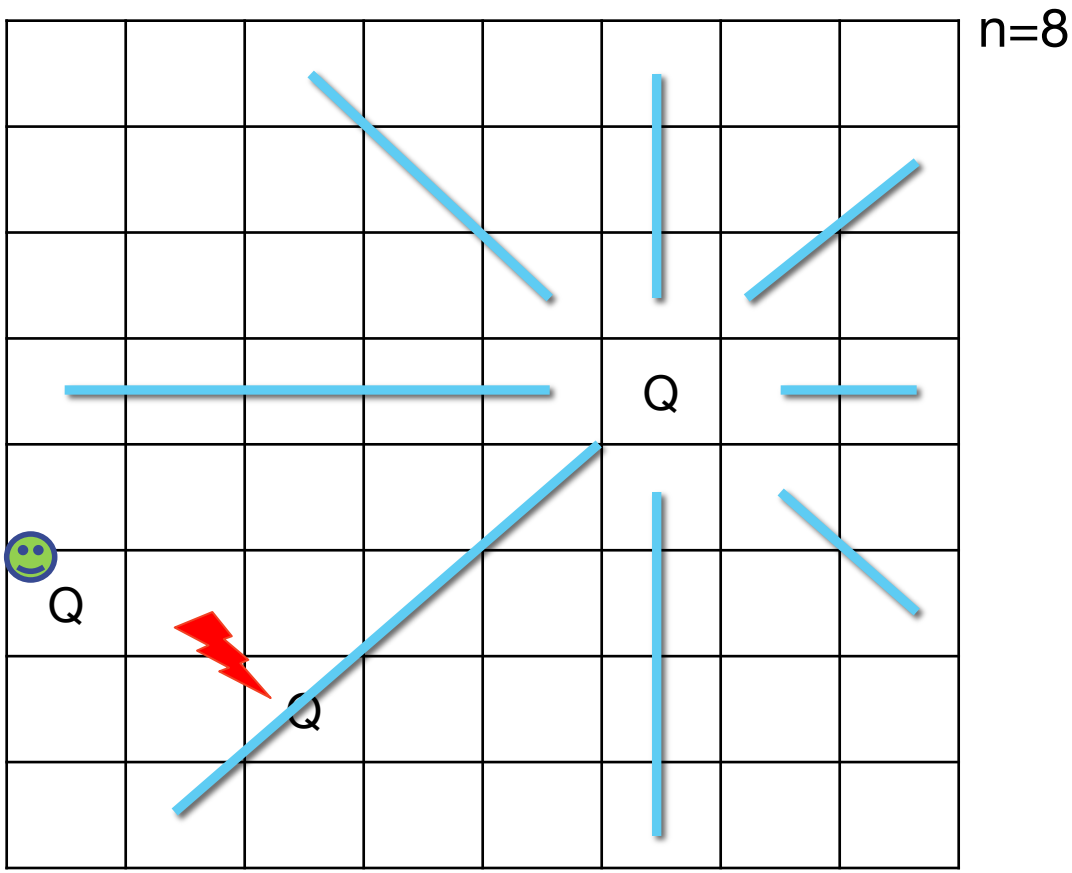
What? [Objectives]

- Understand the concept of hybrid compilation.
- Get insight into the compiler tool-chain and learn how to add capabilities to components.
- Experiment with scheduling algorithms.

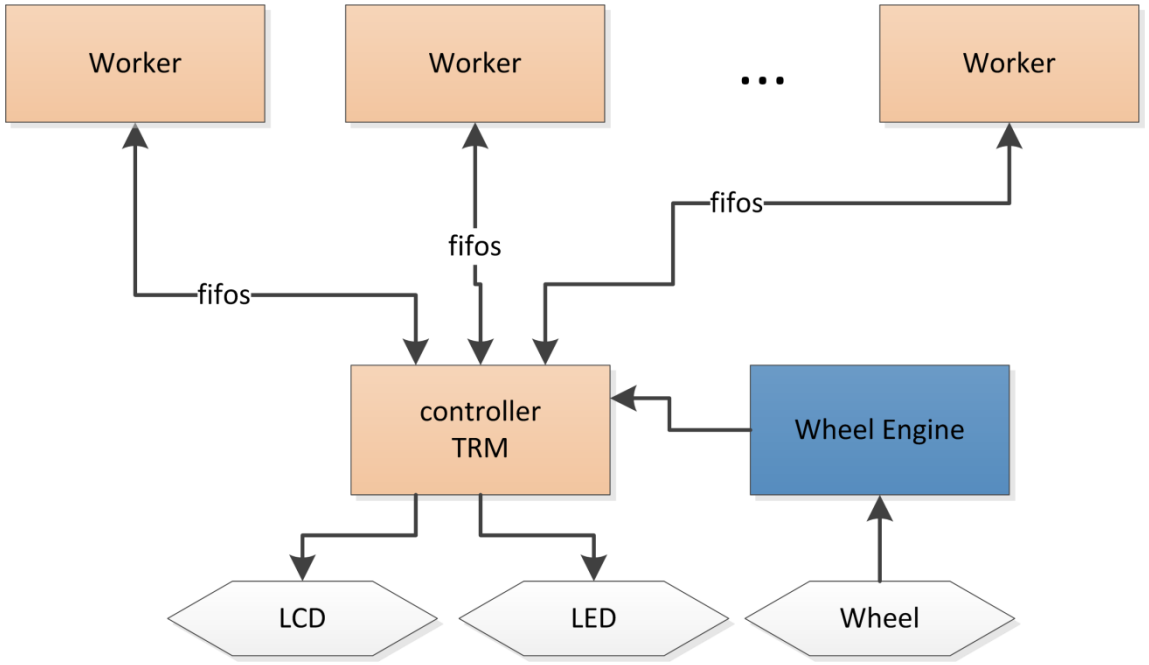
How? [Tasks]

- Add a new device to the Active Cells Compiler tool-chain.
- Improve the *n-Queen* Application

the N-Queen Problem



N-Queen Solution on FPGA

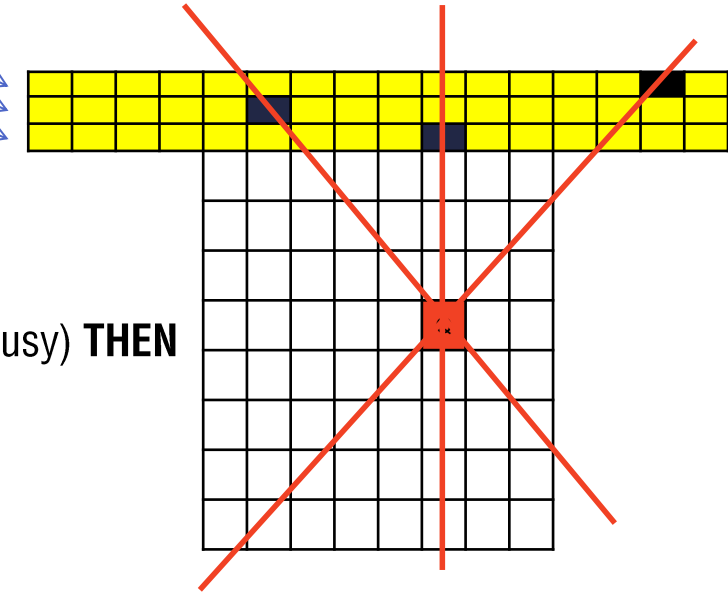


N-Queen Cellnet

```
CELLNET Queens;

...
CONST NumWorkers = 4;

VAR
  controller: Controller;
  workers: ARRAY NumWorkers OF Worker;
  i: LONGINT;
BEGIN
  NEW(controller);
  FOR i := 0 TO NumWorkers-1 DO
    NEW(workers[i]);
    CONNECT(workers[i].out, controller.in[i]);
    CONNECT(workers[i].dbgOut, controller.dbgIn[i]);
    CONNECT(controller.out[i], workers[i].in);
    CONNECT(controller.dbgOut[i], workers[i].dbgIn);
  END;
END Queens.
```

Counting solutions

```
PROCEDURE Count(xS, xMinusy, xPlusy: SET; y: INTEGER): INTEGER;
VAR
  result, x: INTEGER;
  s1, s2, s3: SET;
BEGIN
  IF (y = boardSize) THEN
    result := 1
  ELSE
    result := 0;
    FOR x := 0 TO boardSize - 1 DO
      IF ~(x IN xS) & ~(x-y+boardSize IN xMinusy) & ~(x+y IN xPlusy) THEN
        s1 := xS + {x};
        s2 := xMinusy + {x - y + boardSize};
        s3 := xPlusy + {x + y};
        result := result + Count(s1, s2, s3, y + 1)
      END
    END
  END;
  RETURN result
END Count;
```



Worker Cell

```
Worker*= CELL {DataMemorySize(1024), CodeMemorySize(1024)}
            (in: PORT IN; out: PORT OUT);
VAR boardSize, position, result: LONGINT;
BEGIN
  LOOP
    in ? boardSize;
    in ? position;
    result := Count({position}, {position + boardSize},  {position}, 1);
    out ! result;
  END;
END Worker;
```

Controller: Static Scheduling

```
(* Scheduling *)
i := 0;
WHILE i < boardSize DO
   j := 0;
  WHILE (j < NumWorkers) & (i<boardSize) DO
    IF j IN cores THEN
      out[j] ! boardSize;
      out[j] ! i;
      INC(i);
    END;
    INC(j);
  END;
END;
```

```
(* wait for results and display *)
i := 0; sum := 0;
WHILE i < boardSize DO
  j := 0;
  WHILE (j < NumWorkers) &
              (i < boardSize) DO
    IF j IN cores THEN
      in[j] ? result;
      INC(sum, result);
      INC(i);
      LCD.SetDDRamAddress(1,0);
      LCD.String("(Sum= ");
      LCD.WriteInt(sum);
      LCD.String(")");
    END;
    INC(j);
  END;
END;
```

Active Cells in C# - ActiveCells#

Motivation

- Embed Active Cells in an industrial standard language.
- Prove orthogonality of the Active Cells model to a hosting language.
- Prove flexibility of the Fox compiler and tool-chain.

Slides from Alexey Gokhberg, main designer of ActiveCells#

Basic constructs of ActiveCells#

- Cell nets

```
cellnet FinanceSystem { ... }
```

- Cells and ports

```
cell OrderBook(in pin, out pout) { ... }
```

- Send to a port

```
pout ! stockNum, min, price, type;
```

- Receive from a port (blocking)

```
pin ? stockNum, count, price, type;
```

- Receive from a port (non-blocking)

```
bool b = pin ?? stockNum;
```

Program Structure

```
cellnet FinanceSystem {
    import RS232;
    import LCD;
    import Timer;

    const int OrderBooks = 8;

    // cell definitions
    cell Demultiplexer(in pin, out[OrderBooks] pout) { ... }
    cell Multiplexer(in[OrderBooks] pin, out pout) { ... }
    cell OrderBook(in pin, out pout) { ... }
    cell Sink(in pin) { ... }
    cell Source(out pout) {

    // cell instances
    Source source;
    Demultiplexer demux;
    Multiplexer mux;
    Sink sink;
    OrderBook[] orderBooks;

    // cellnet constructor
    FinanceSystem() { ... }
}
```

Topology Instantiation

```
Source source;
Demultiplexer demux;
Multiplexer mux;
Sink sink;
OrderBook[] orderBooks;

// cell net constructor
FinanceSystem() {

    // instantiate cells
    source = new Source();
    demux = new Demultiplexer();
    mux = new Multiplexer();
    sink = new Sink();
    orderBooks = new OrderBook[OrderBooks];

    // connect cells
    source(pout) >> demux(pin);
    mux(pout) >> sink(pin);

    for (int i = 0; i < OrderBooks; i++) {
        orderBooks[i] = new OrderBook();
        demux(pout[i]) >> orderBooks[i](pin);
        orderBooks[i](pout) >> mux(pin[i]);
    }
}
```

Multi-Cast

- Motivation

```
// Variant 1: non-blocking receive
for (int i = 0; i < OrderBooks; i++) {
    if (pin[i] ?? stockNum) {
        pin[i] ? count, price, type;
        pout ! stockNum, count, price, type;
    }
}


// Variant 2: multi-cast receive
for ( ; ; ) {
    select {
    case (int i in 0 : OrderBooks-1) pin[i] ? stockNum:
        pin[i] ? count, price, type;
        pout ! stockNum, count, price, type;
        break;
    }
}
```

Multi Cast - Implementation

Source representation

```
select {
case c ? x:
    S1;
case (int i in m : n) d[i] ? y:
    S2;
}
```

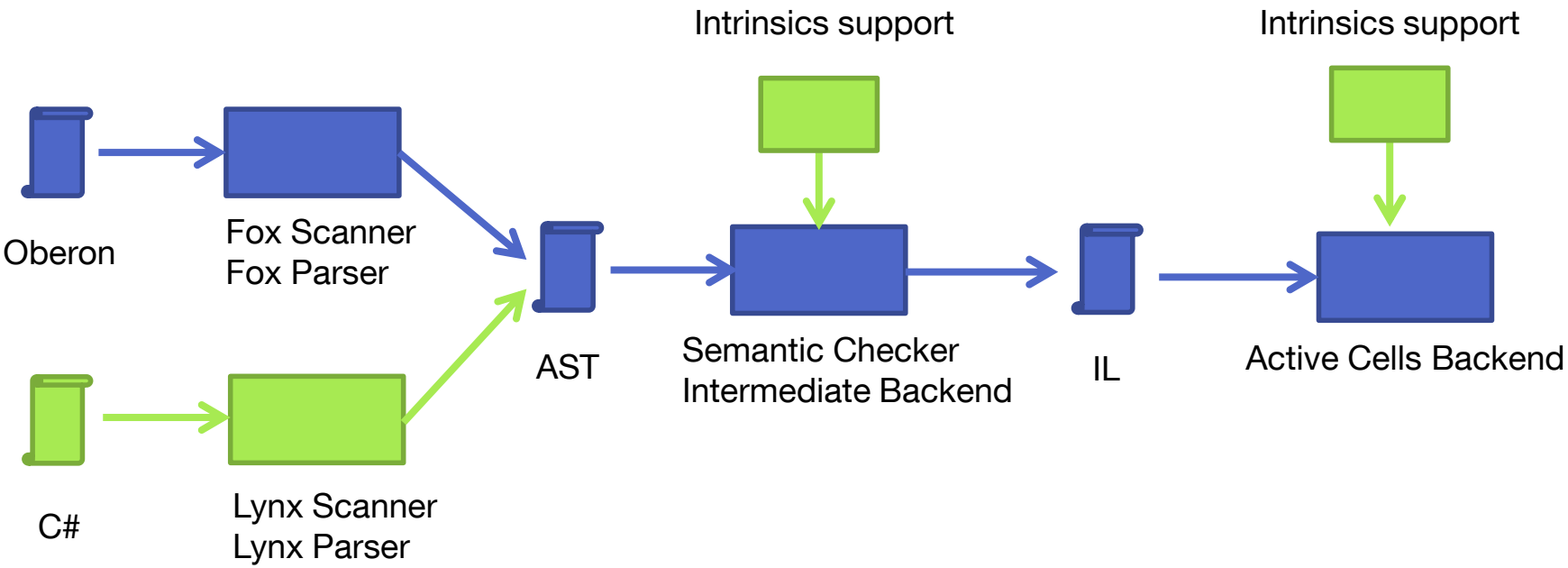
Intermediate representation

```
lynx@newsel;
lynx@addsel(0, 0, c);
FOR i := m TO n DO
    lynx@addsel(1, i, d[i])
END;
CASE lynx@select() OF
0:
    c ? x;
    S1;
| 1:
    i := lynx@selidx();
    d[i] ? y;
    S2;
END;
```

Capabilities in ActiveCells#

```
import RS232;
import LED;
import LCD;

[RS232, LED, LCD]
cell Controller {

  void main() {
    sbyte ch;

    LED.SetLED(12);
    LCD.Init();
    LCD.Clear();
    for ( ; ; ) {
      RS232.Receive(ch);
      LED.SetLED(ch);
      RS232.Send(ch+1);
      LCD.WriteInt(ch);
      LCD.Ln();
    }
  }
}
```

ActiveCells# Compiler Implementation



Compiler design principles:

- independent front-end is provided for Lynx
- back-end FOX compiler infrastructure is completely re-used
- extended C# functionality is mapped to intrinsic functions (lynx@func) in AST and IL
- back-end components are extended to support intrinsics

Active Cells Revisited

- ActiveCells was successfully used
  - implemented a range of development projects
  - 2 industrial applications
- Despite the success we met some problems which
  - hinder product development productivity
  - hinder ActiveCells development productivity
- Problems are *mild* in comparison with problems of standard commercial tools!

Challenges

- Expansion of the HW component library with Engines
  - Necessity to change the compiler with every new supported Engine
  - A lot of compiler code replication
  - Complications in the structure of the HW library repository (a lot of directories, replication of HDL code etc.)
  - Far not enough dynamicity and flexibility for productive HW library development
- SoftCore peripherals
  - Many peripherals share SoftCore buses in the same way as Engines
  - Same productivity drawbacks as mentioned above

Challenges

- Component interconnects
  - Non-standard, non-generic connectivity interface
    - not possible to use components with standard interfaces
- Parameterization of the HW components
  - No possibility to flexibly parameterize the HW components from the software code
- Specification of the target device
  - Necessity to change the compiler

ActiveCells 2

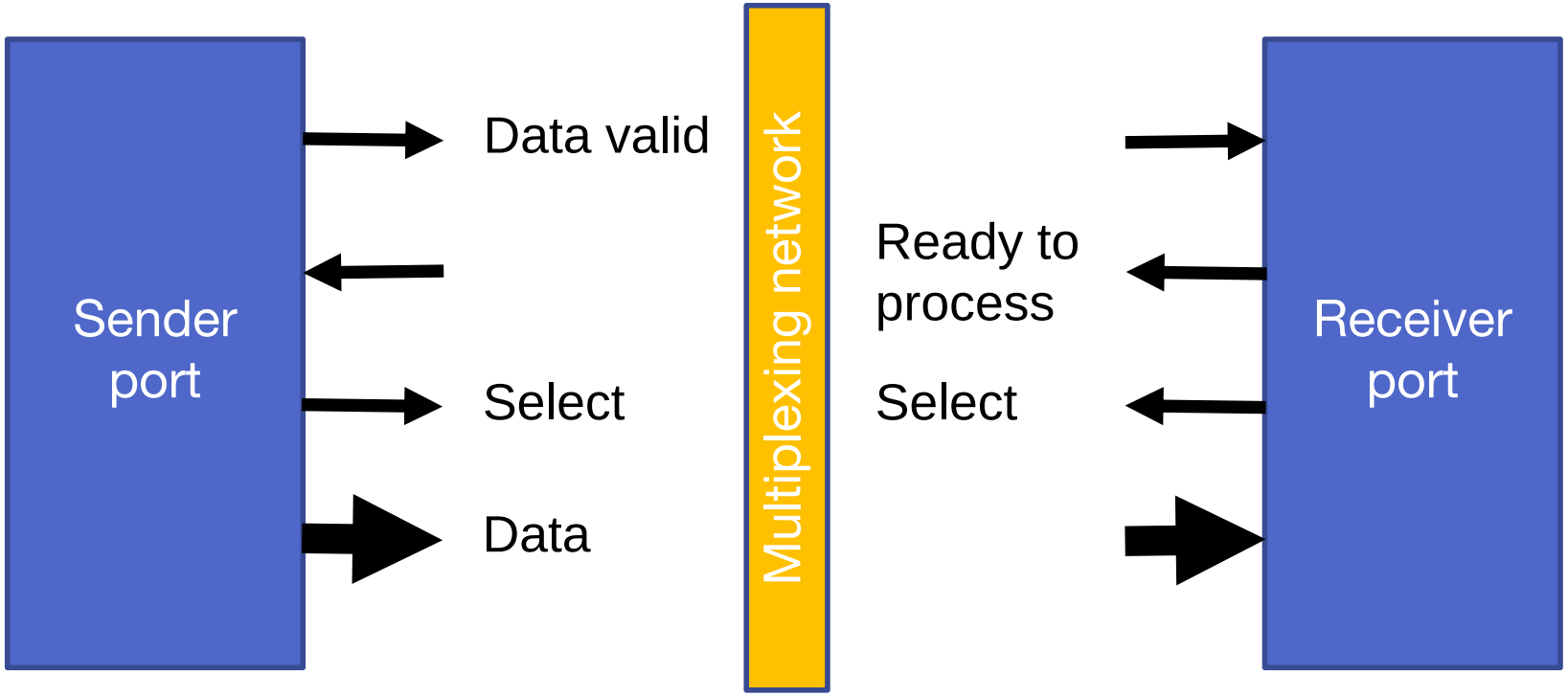
- Generic and flexible representation of HW components
- XML-based runtime reloadable specification

<component type="HdlComponent" name="Gpo" description="general purpose output (GPO) component" isa="none">

```
<component type="HdlComponent" name="Gpo" description="general purpose output (GPO) component" isa="none">
  <supported>
    <element type="StringValue" value="*"/>
  </supported>
  <ports>
    <element type="ClockHdlPort" name="aclk" mapped="systemClock" direction="in" width="1">
      <clockFrequencyConstraints>
        <element type="IntegerRangeValue" value="1:any:1"/>
      </clockFrequencyConstraints>
      <clockDutyCycleConstraints>
        <element type="RealValue" value="0.5"/>
      </clockDutyCycleConstraints>
    </element>
    <element type="HdlPort" name="aresetn" mapped="systemReset" direction="in" width="1" signalPolarity="false"/>
    <element type="AXI4StreamPort" name="inp" description="GPO channel data input (AXI4 Stream)" direction="in" width="?{parameters['DW'].value}?">
      <tvalid name="inp-tvalid" mapped="tvalid" description="AXI4 Stream TVALID" direction="in" width="1"/>
      <tready name="inp-tready" mapped="tready" description="AXI4 Stream TREADY" direction="out" width="1"/>
      <tdata name="inp-tdata" mapped="tdata" description="AXI4 Stream TDATA" direction="in" width="?{parameters['DW'].value}?"/>
    </element>
    <element type="TerminalHdlPort" name="gpo" mapped="Gpo?{instance.capabilityParameters['Id'].integer:0}?" direction="out" width="?{parameters['DW'].value}?"/>
  </ports>
  <parameters>
    <element type="HdlParameter" name="DW" mapped="DataWidth" description="number of GPO bits">
      <value type="IntegerValue" value="?{instance.capabilityParameters['DataWidth'].integer:8}?"/>
      <constraints>
        <element type="IntegerRangeValue" value="1:"/>
      </constraints>
    </element>
  </parameters>
  <dependencies>
    <element value="Gpo.v"/>
  </dependencies>
</component>
```

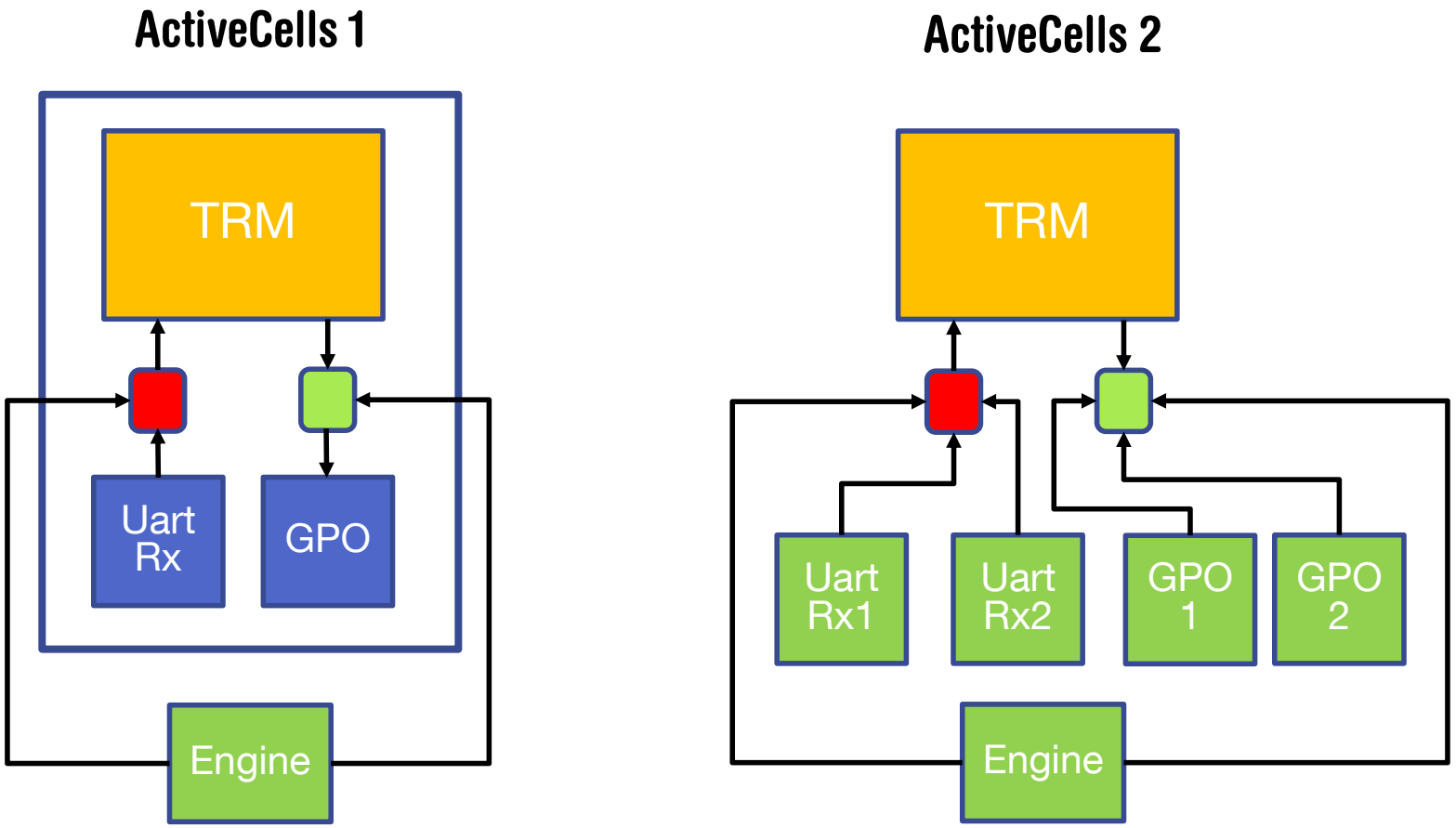
Generic Peer-to-Peer Communication Interface

- Use of AXI4 Stream interconnect standard from ARM
  - Generic, flexible
  - Non-redundant
  - Well suited for high performance



ActiveCells 2

- Many peripherals can be flexibly connected to SoftCores, in the same way as Engines and in multiple instances



ActiveCells 2

- Flexible parameterization of the components using interpreted code in the component specification
  - XML-based specification (→ object persistency)

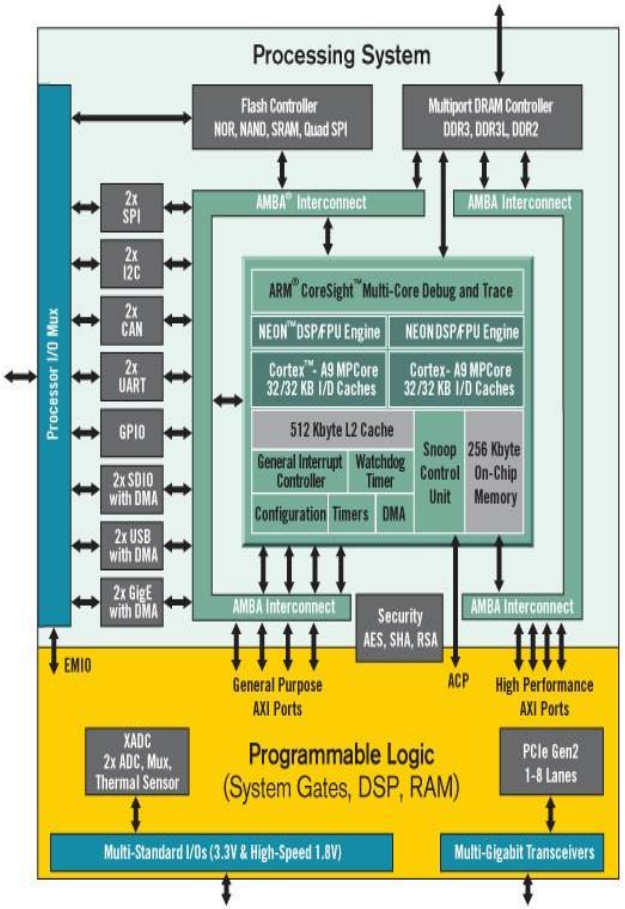
```
<element type="HdlParameter"  name="KernelLength"
                 description="filter kernel length">
   <value type="IntegerValue"
  value="?{instance.capabilityParameters['KernelLength'].integer:7}?"/>
  <constraint>
       <element type="IntegerRangeValue" value="1:"/>
  </constraints>
</element>
```

  - Instance-wise parameterization is possible

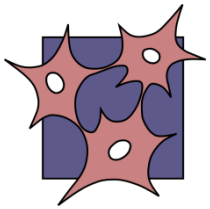
```
var
controller{Arch="TRM"}: Controller;

new(fir[k]{CoeffWidth=16,InpWidth=16,OutWidth=16,KernelLe
ngth=12,InitShift=15});
```

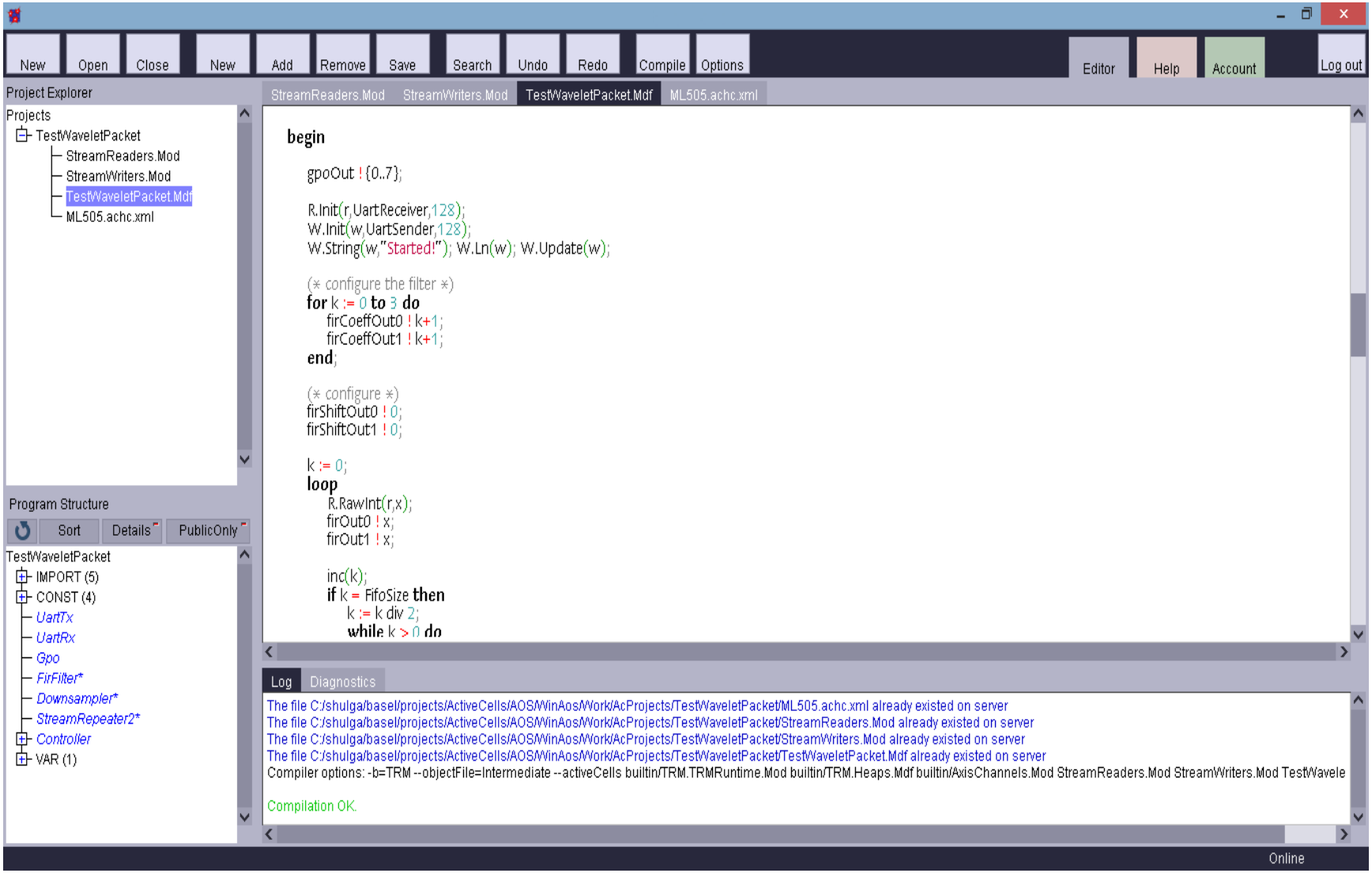
ActiveCells 2

- Support of Xilinx Zynq SOC (Dual ARM + FPGA logic)

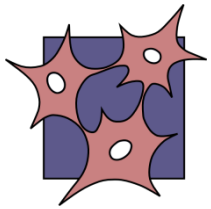


ActiveCells 2 IDE





ActiveCells 2



- Automated design flow by one button press
- Remote compilation and implementation using a dedicated WEB service
- Multiple project management
- Open source repository of HW components
- Help and tutorials for easy and convenient start

A preview version coming soon!