TRM Implementation,

Communication Interface

Hybrid Compilation

4.3 BEHIND THE SCENES OF ACTIVE CELLS

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)

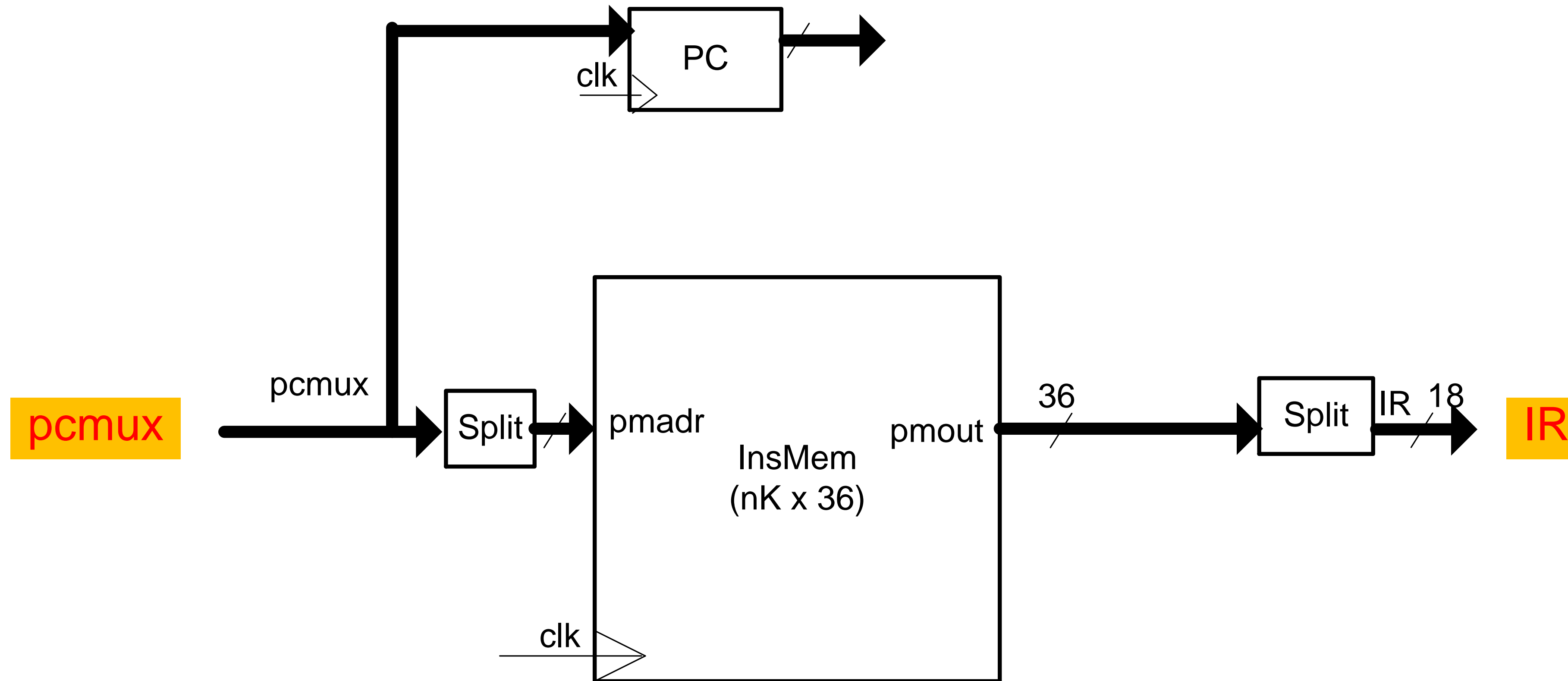
```
reg [31:0] a,b,c;

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

Single-Cycle Datapath: arithmetical logical instruction fetch



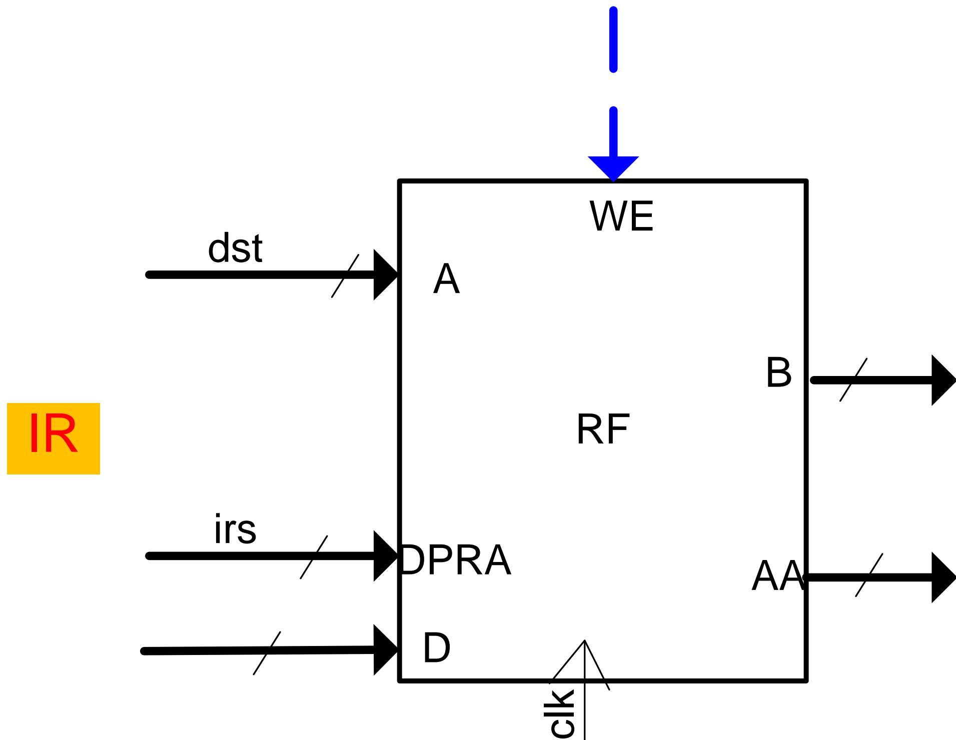
from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

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({{{32-PAW}{1'b0}},pcmux[PAW-1:1]}),
   .pmout(pmout));

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

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

Single-Cycle Datapath: register read

- **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 ird = IR[13:11];
assign dst = (BL)? 7: ird;
```

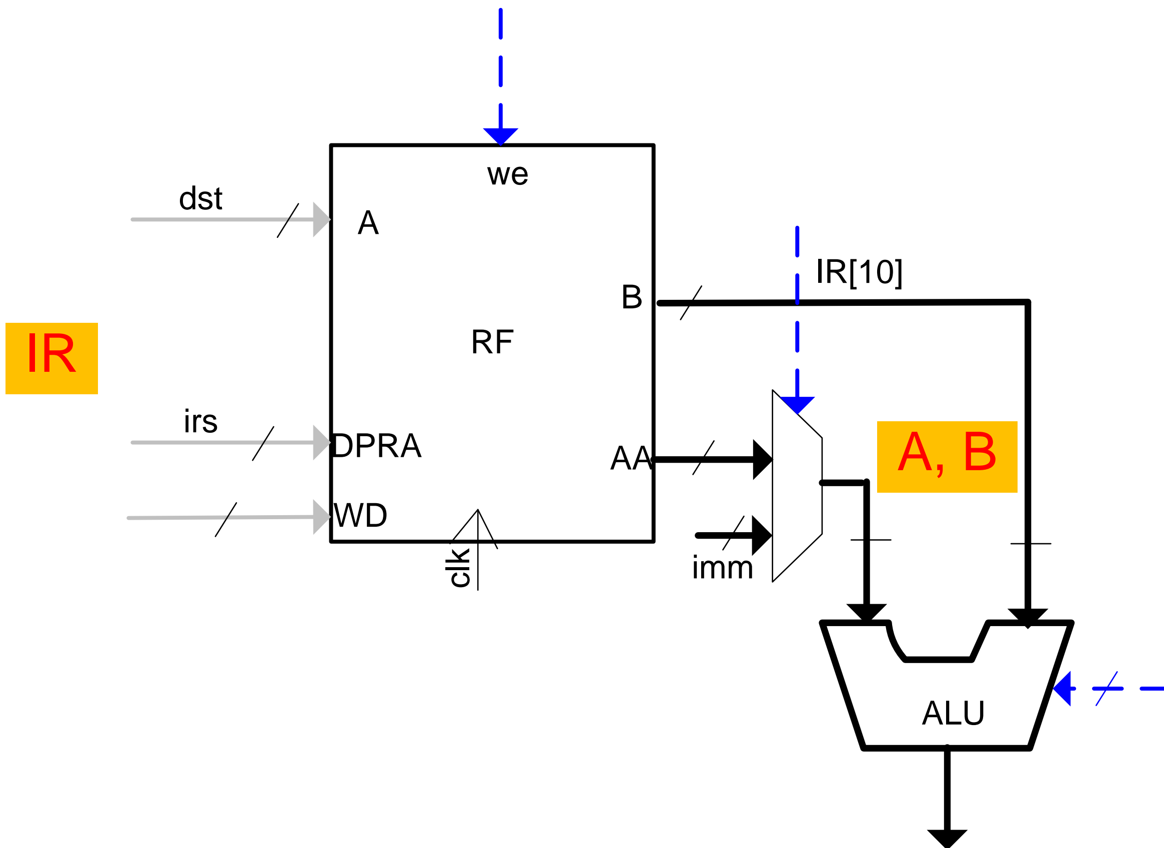
source register

destination register

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

Single-Cycle Datapath: ALU

- **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};

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;
```

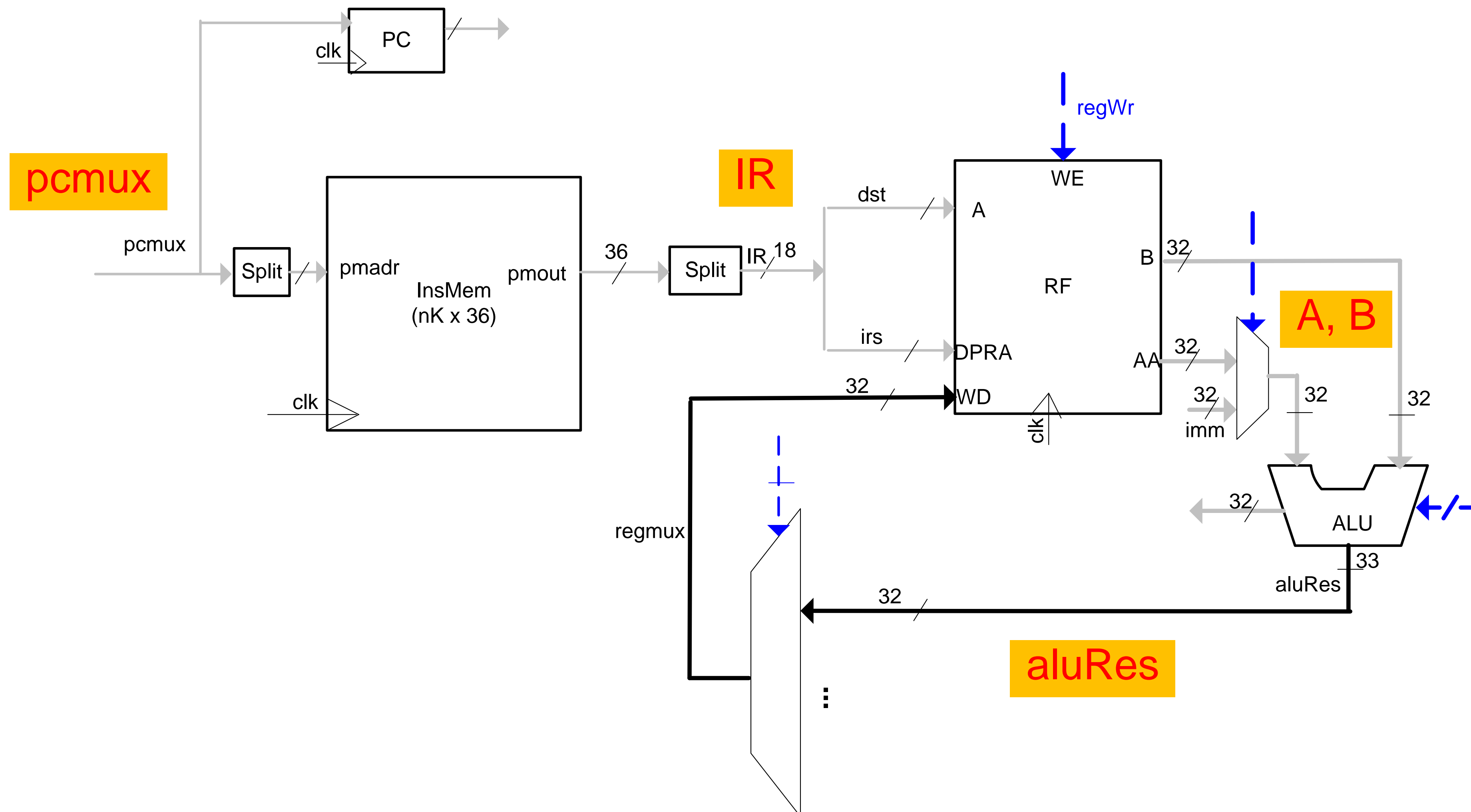
Control Path

```
assign vector = IR[10] & IR[9] & ~IR[8] & ~IR[7];
assign op = IR[17:14];

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];
```

$IR_{17:14}$	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

Single-Cycle Datapath: write back to Rd

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



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

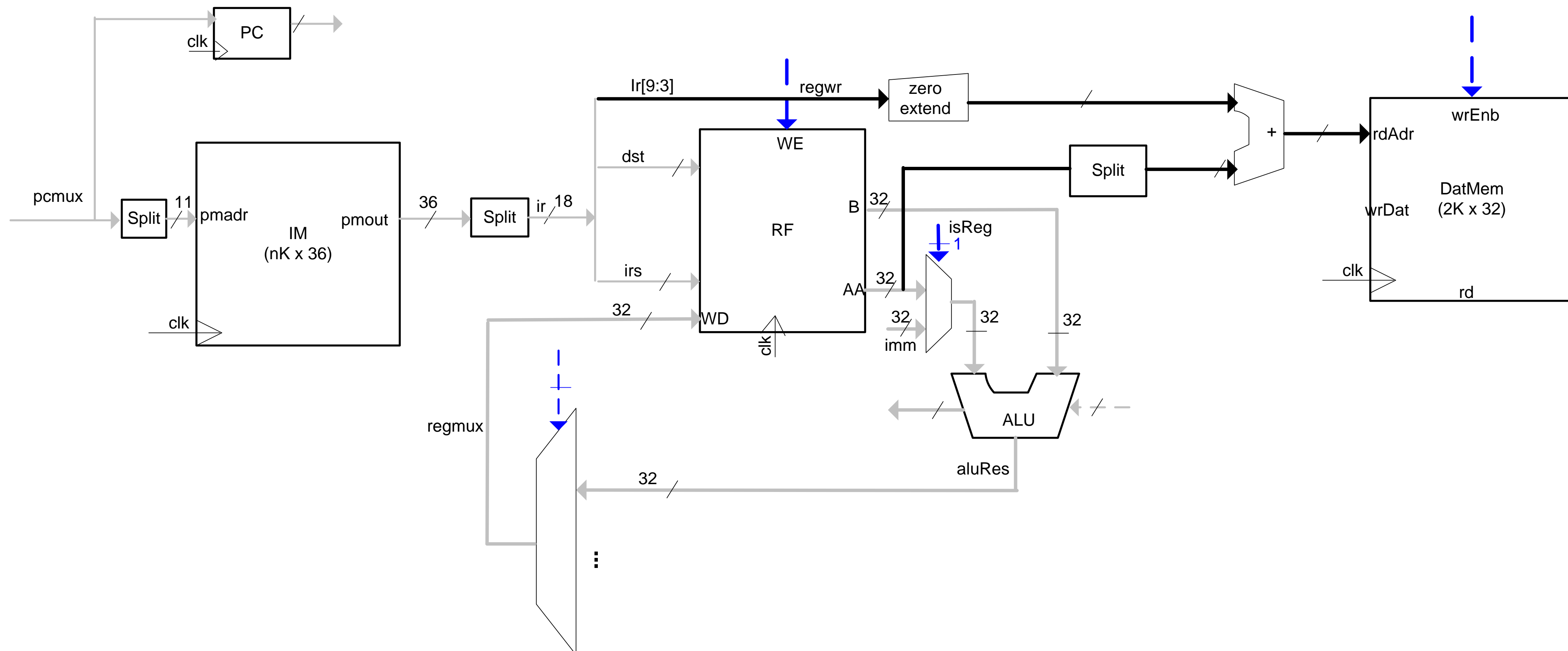
Single-Cycle Datapath: write back to Rd

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

...

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

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;
```


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

Single-Cycle Datapath: `LD`

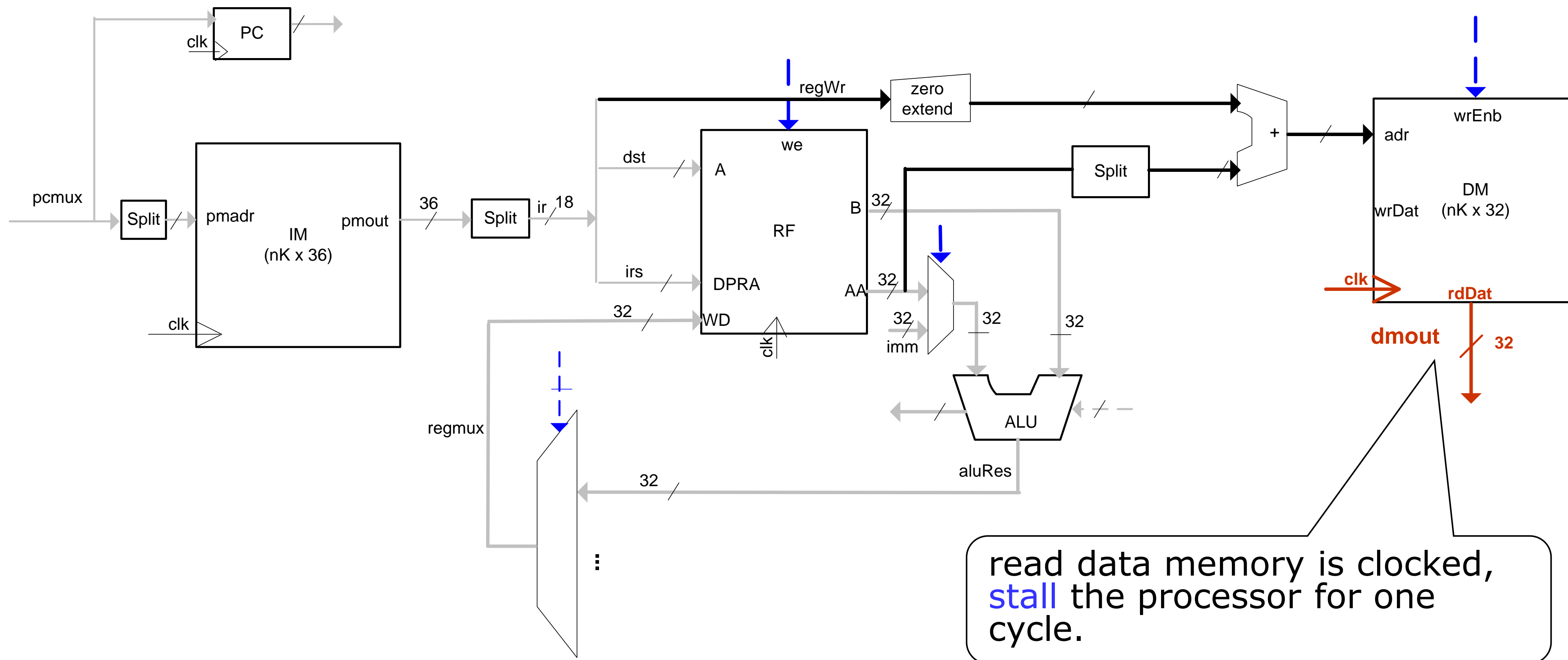
- **STEP 1:** Fetch instruction
- **STEP 2:** Read source operand from the register file
- **STEP 3:** Compute the memory address



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



read data memory is clocked, stall the processor for one cycle.

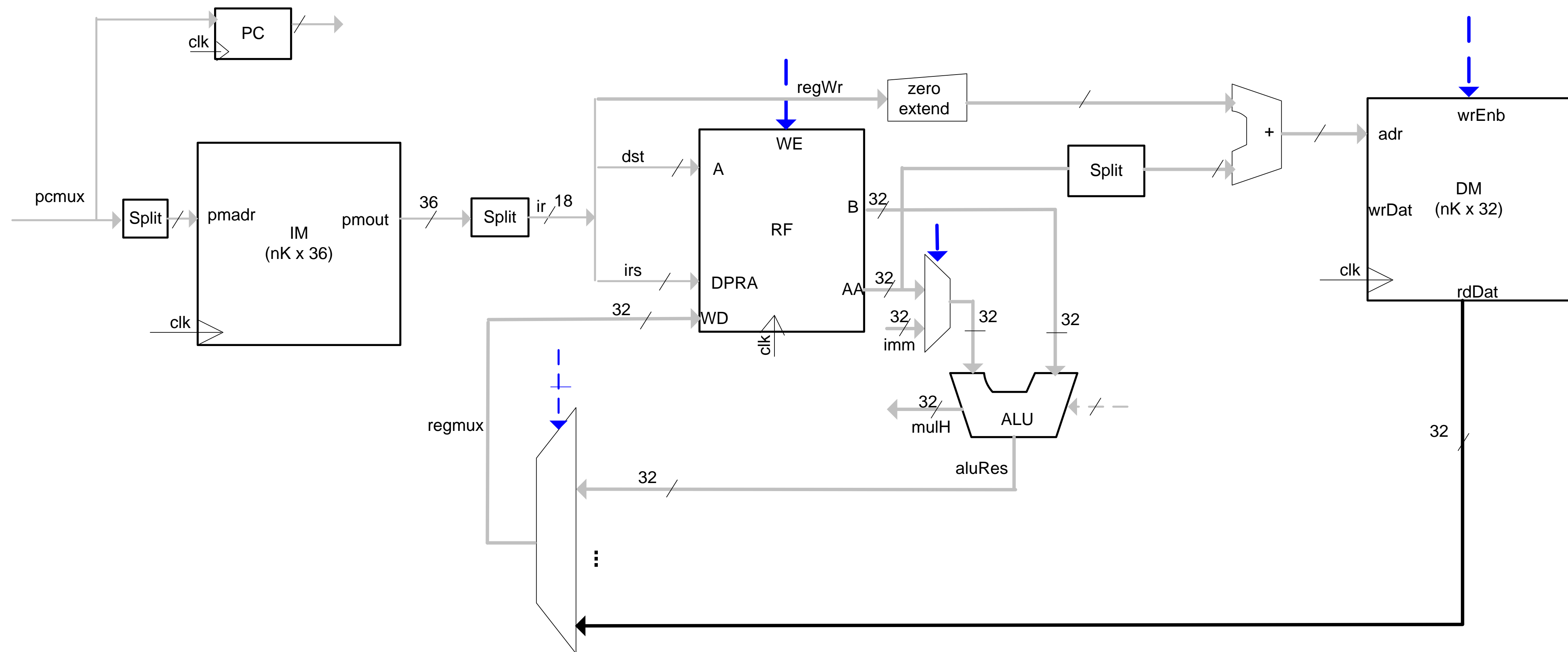
from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

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.

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

LD rd, [rs+offset]

Verilog code in TRM module

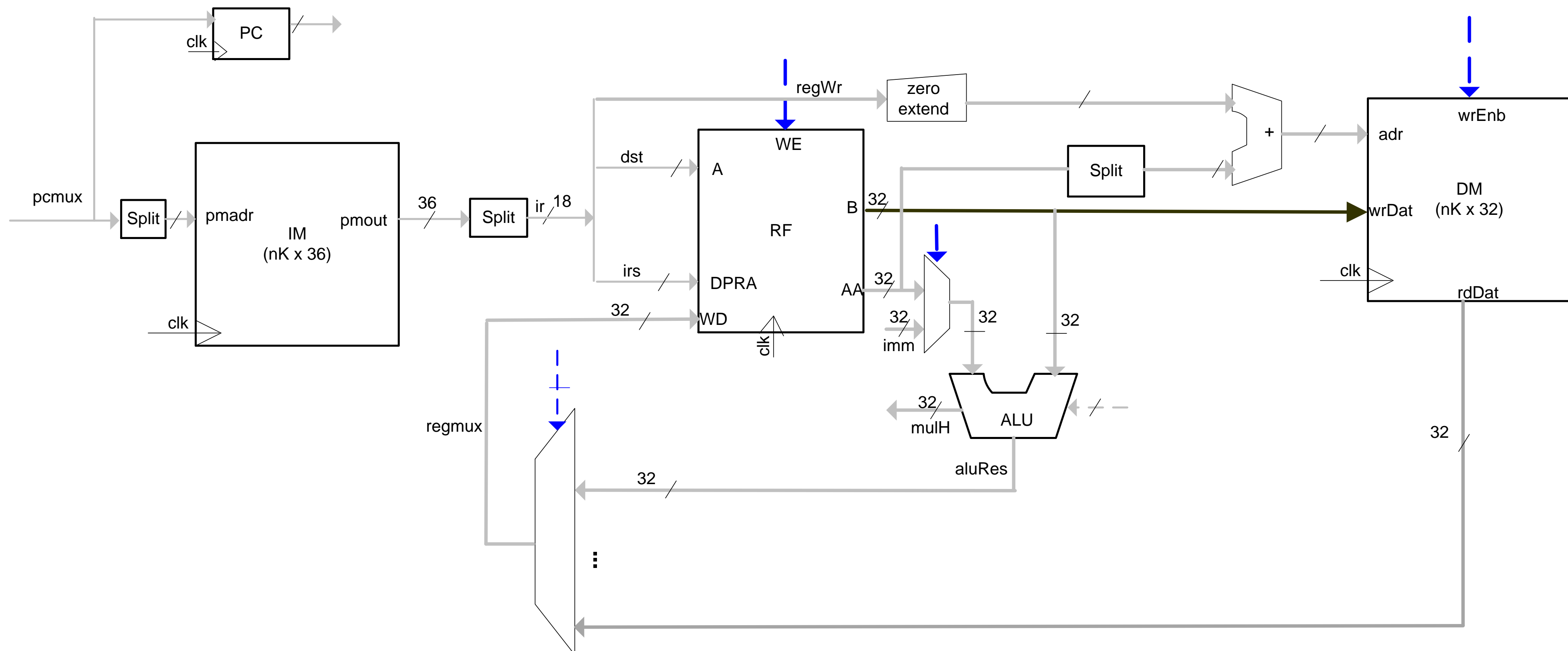
```
wire [31:0] dmout;
wire [DAW:0] dmadr;
wire [6:0] offset;
reg IoenbReg;

//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;
```

Src register = lr 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

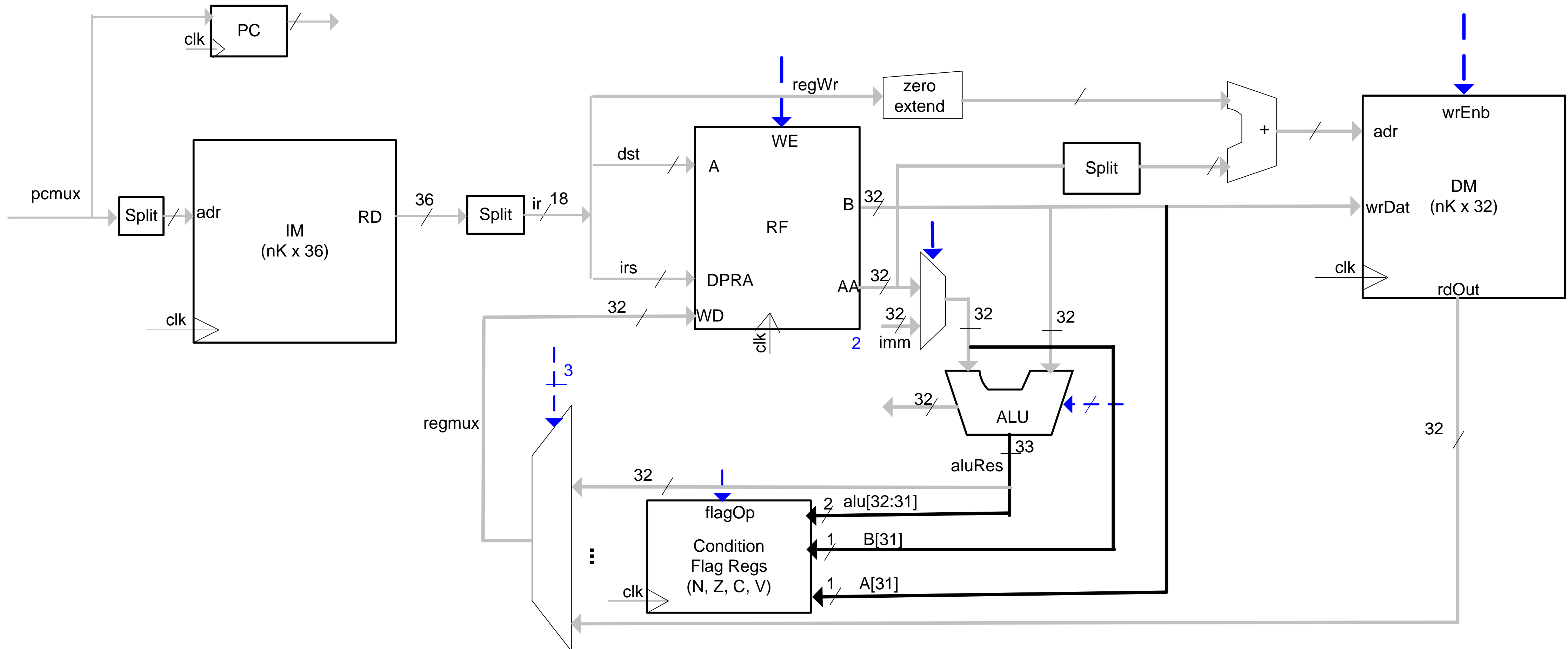
- **STEP 3:** Compute the memory address
- **STEP 4:** Write data into the data memory

```
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
```

Single-Cycle Datapath: set flag registers

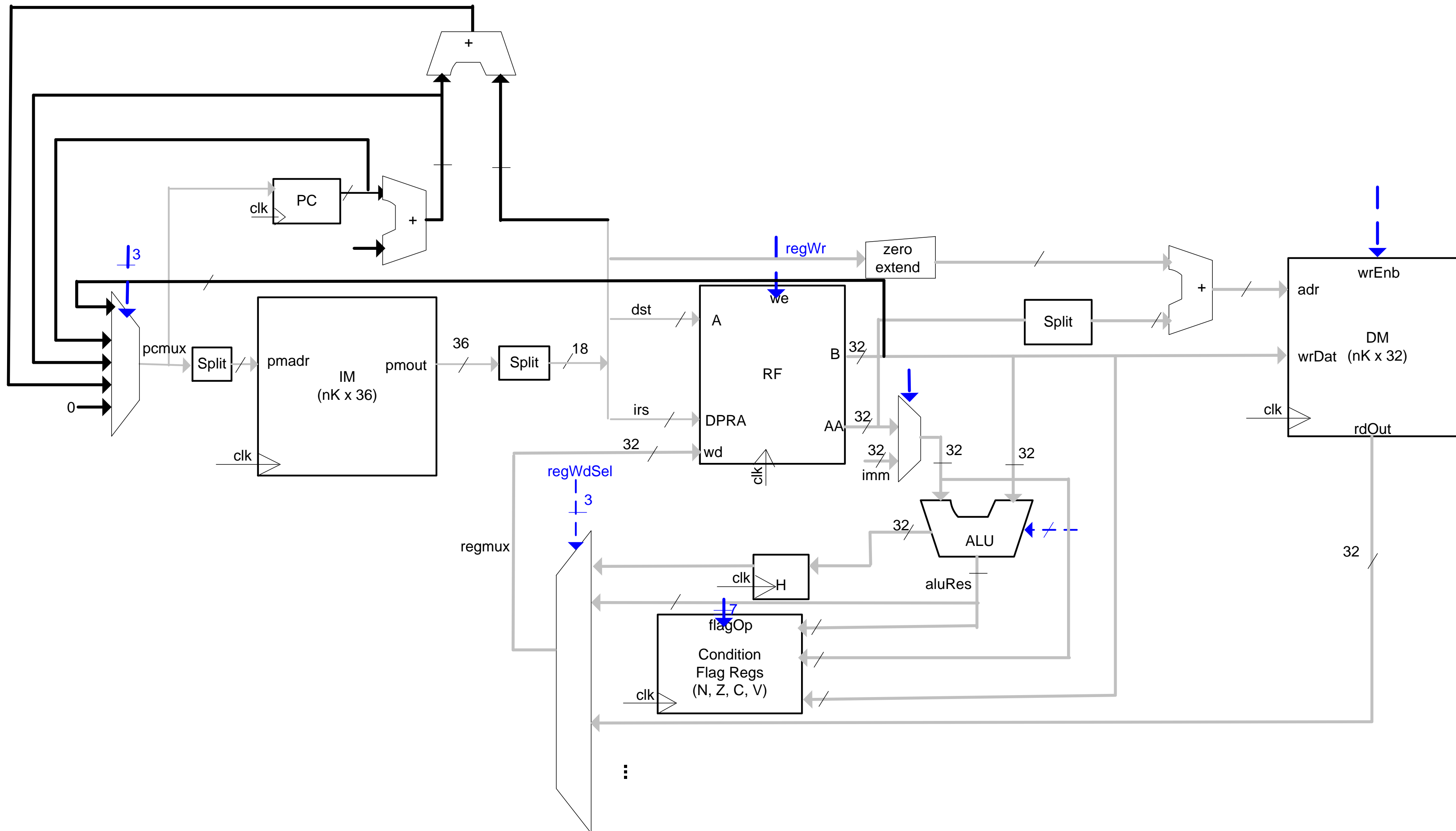


from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

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)**

Single-Cycle Datapath: Branch instructions

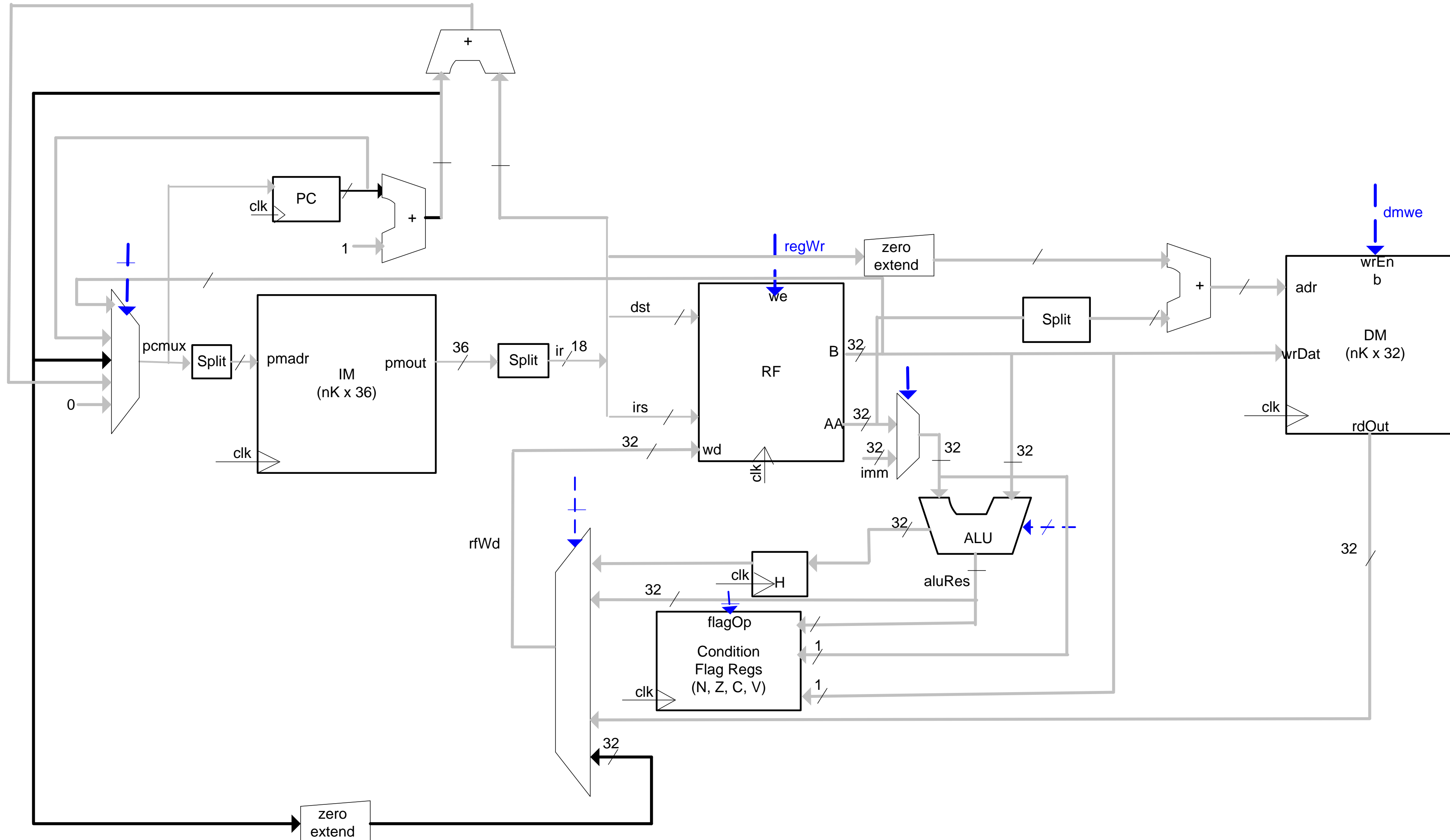


from Lectures on Reconfigurable Computing, *Dr. Ling Liu,* ETH Zürich

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;
```

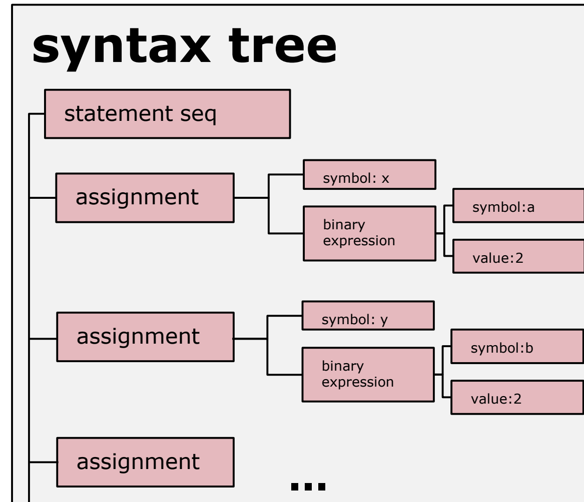
Complete single-cycle datapath



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

Hybrid Compilation: Code Generation

Intermediate Code Generation



syntax tree traversal

intermediate code

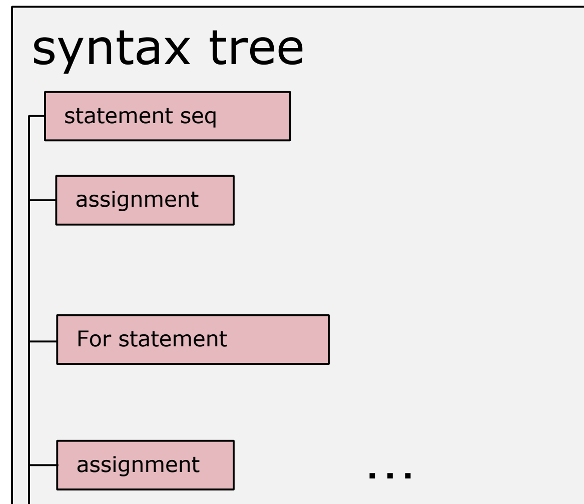
```
.module A
 ....

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

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.

Hybrid Compilation: Architecture Generation

Interpretation

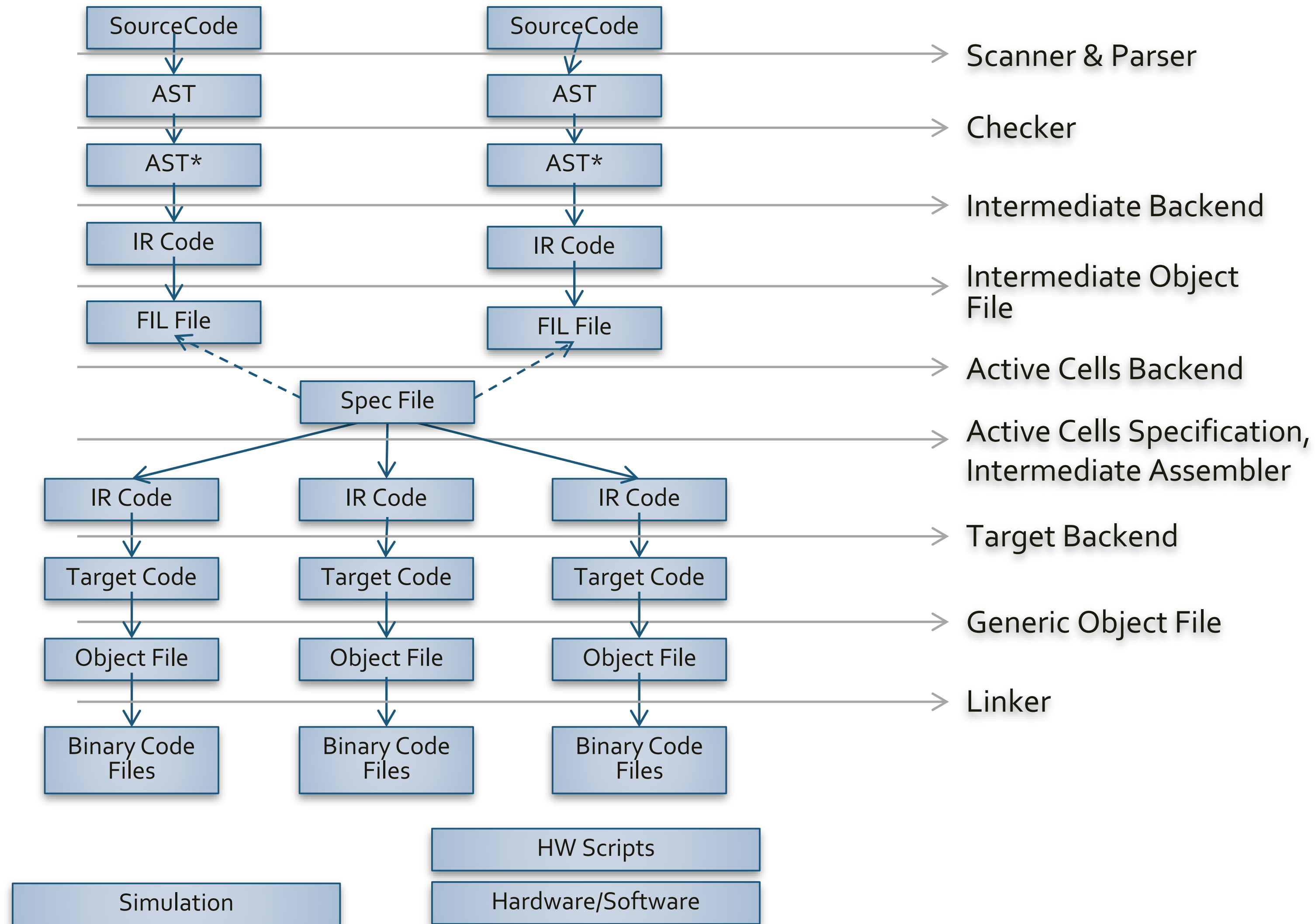


syntax tree traversal

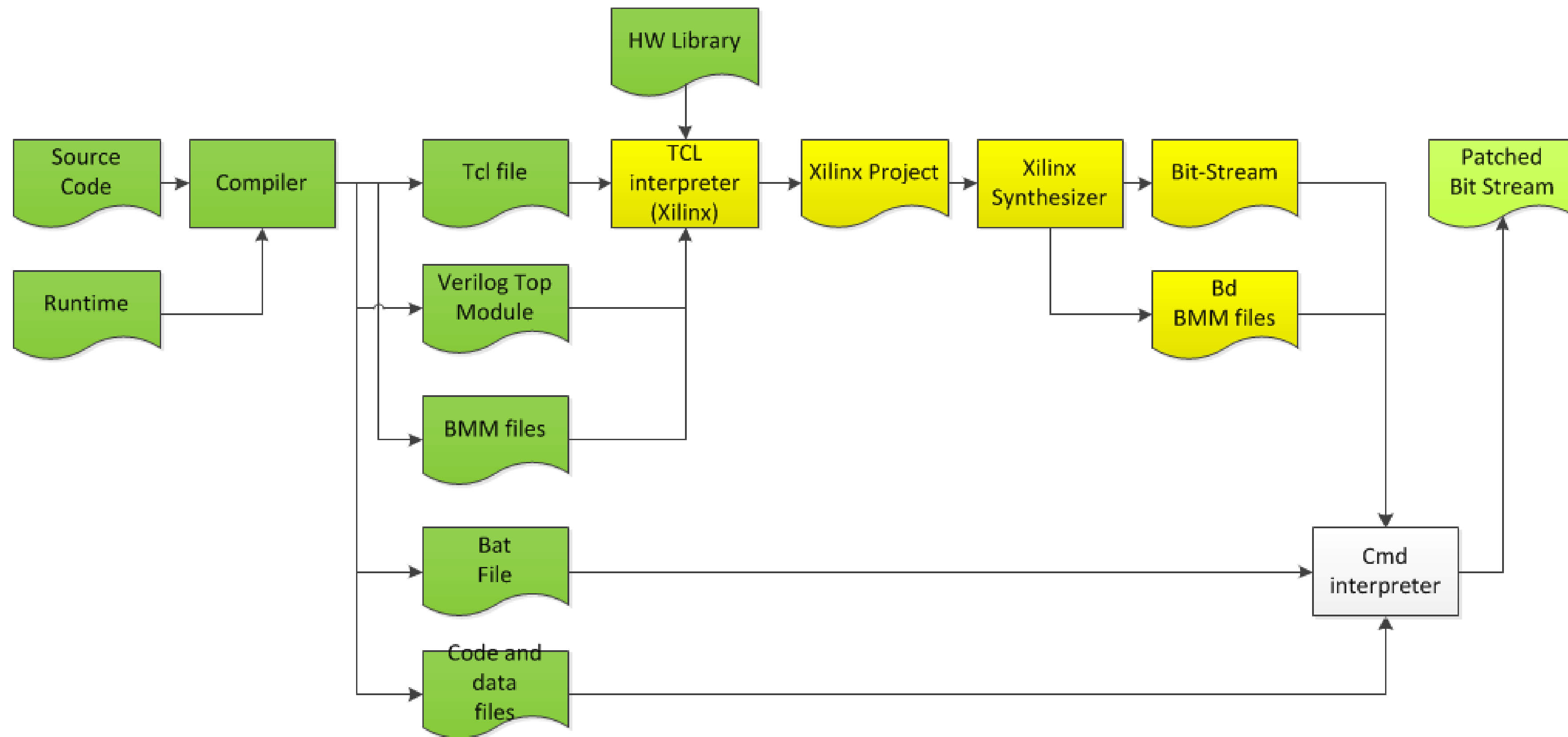
e.g. Active Cells specification

```
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='
....
```

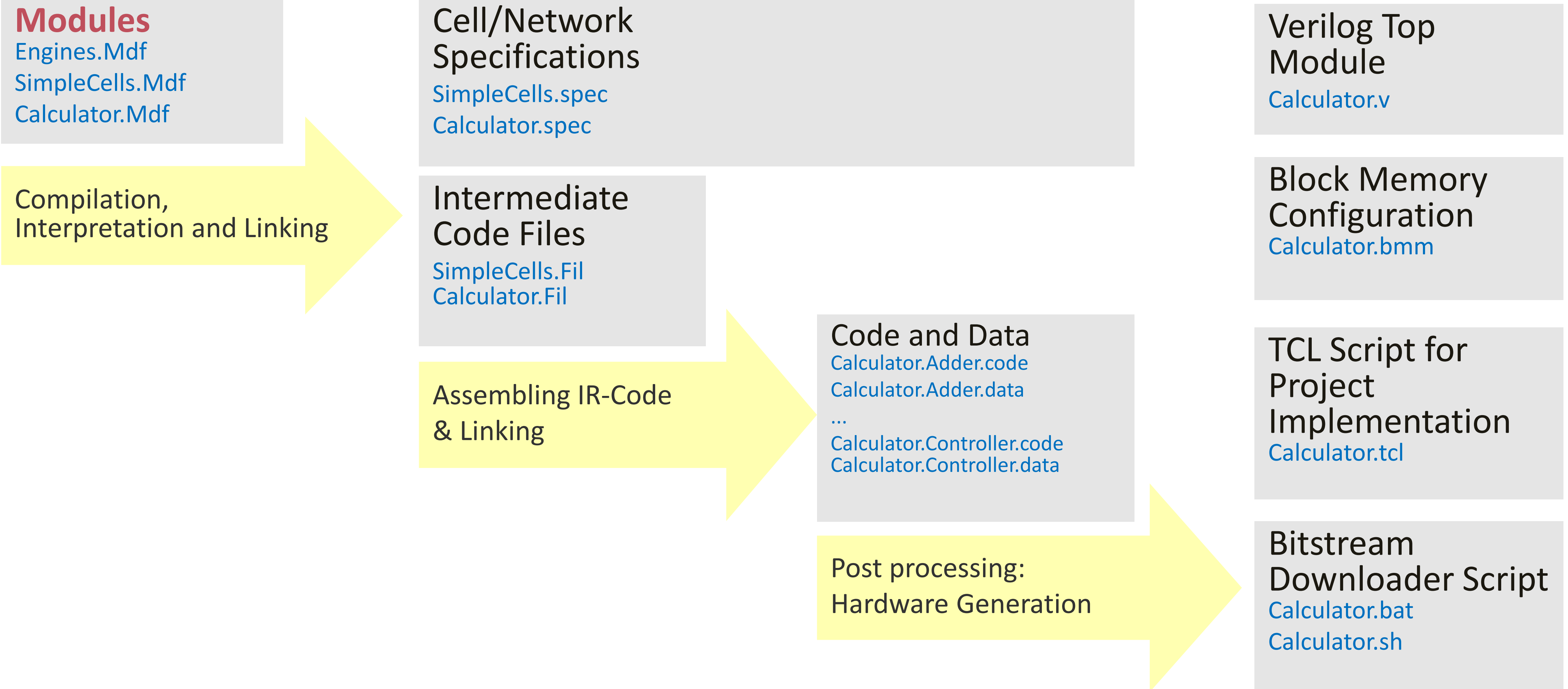
Active Cells Code Generation



Tool-Chain Steps in Detail

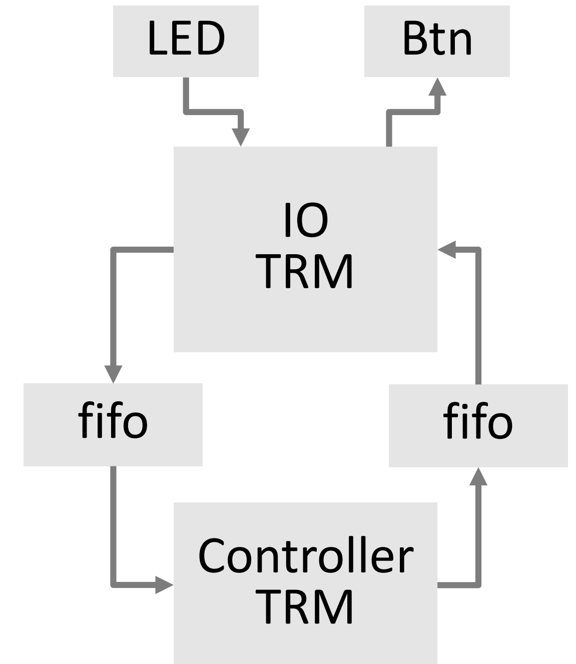


Perspective of the compiler



Mapping to Hardware – Simple Example

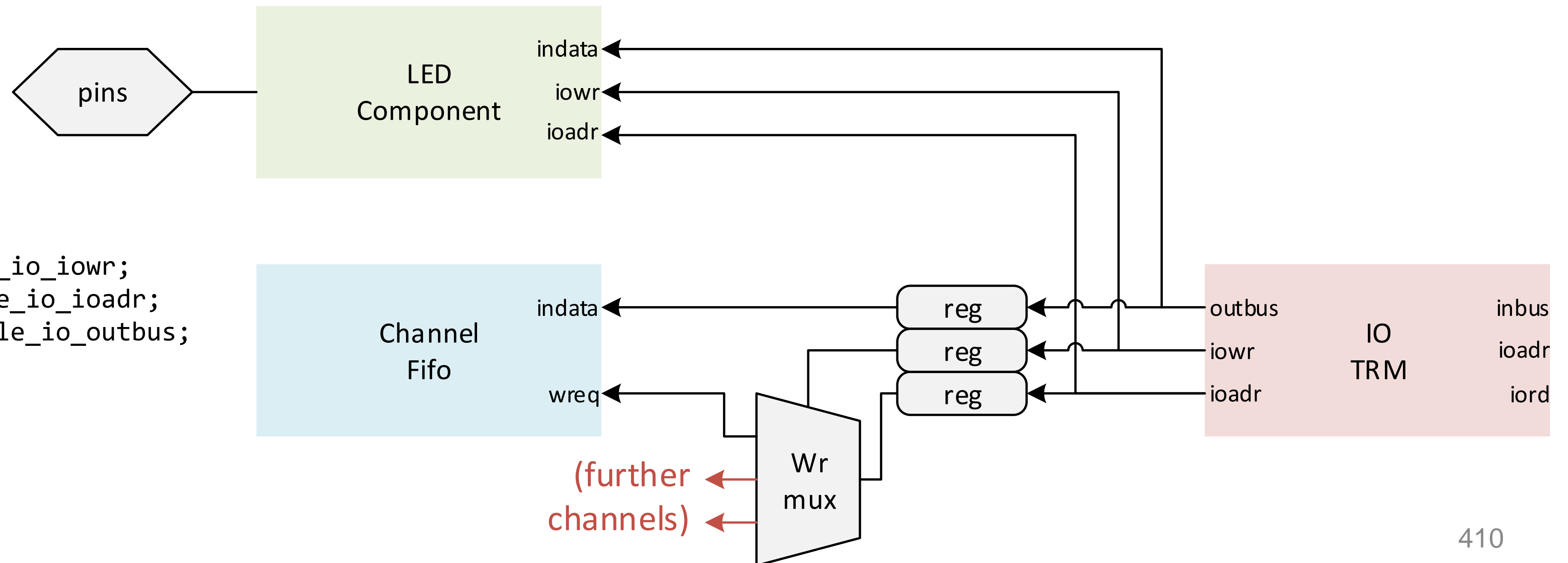
```
IO*=CELL {LED, Button} (in: PORT IN; out: PORT OUT);
BEGIN
END IO;

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

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



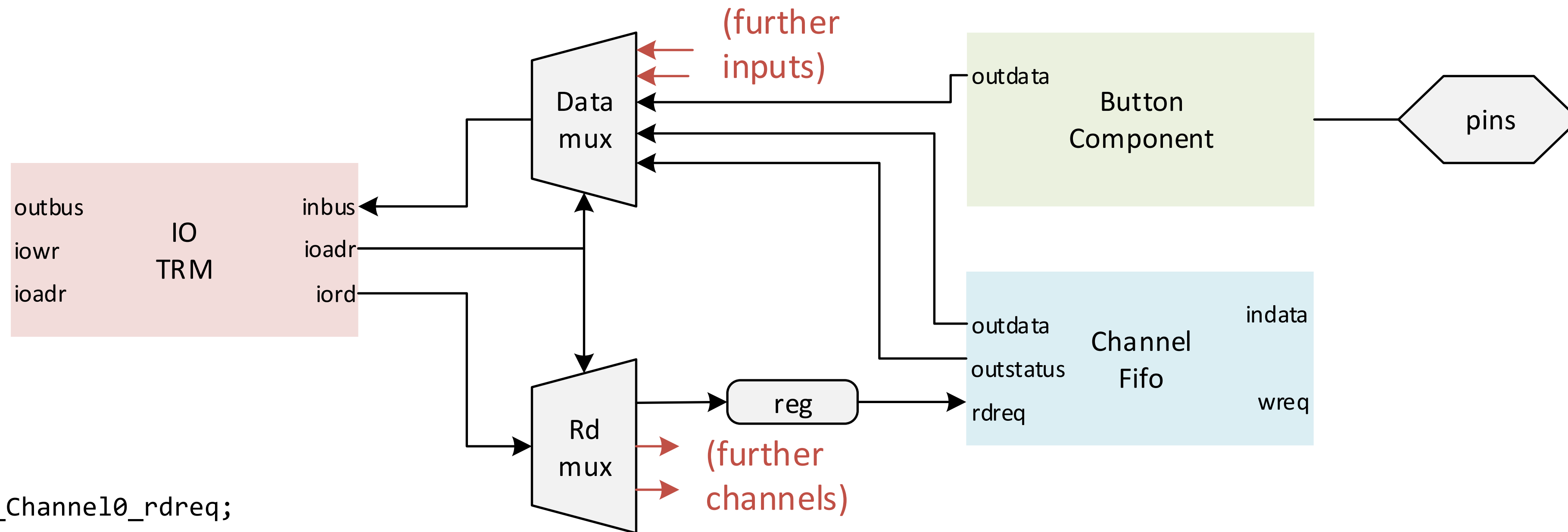
TRM outbound communication

```
reg[5:0] Simple_io_ioadr_reg;
reg Simple_io_iowr_reg;
reg[31:0] Simple_io_outbus_reg;
TRM #(.IMB(1), .DMB(4)) Simple_io(.clk(clk), .rst(rst), .inbus(Simple_io_inbus), .ioadr(Simple_io_ioadr), .iowr(Simple_io_iowr),
.iord(Simple_io_iord), .outbus(Simple_io_outbus));
LED instSimple_IO_LED(.clk(clk), .rst(rst), .inData(Simple_IO_LED_inData), .ioadr(Simple_io_ioadr), .iowr(Simple_io_iowr),
.outData(Simple_IO_LED_outData));
ParChannel #(.Size(32)) Simple__Channel1(.clk(clk), .rst(rst), .wreq(Simple__Channel1_wreq), .rdreq(Simple__Channel1_rdreq),
.inData(Simple__Channel1_inData), .outData(Simple__Channel1_outData), .status(Simple__Channel1_status));
assign Simple__Channel1_inData = Simple_io_outbus_reg;
assign Simple__Channel1_wreq = (Simple_io_ioadr_reg == 34) & Simple_io_iowr_reg;
assign Simple_IO_LED_inData = Simple_io_outbus[7:0];
```

```
always @(posedge clk)
begin
Simple_io_iowr_reg <= Simple_io_iowr;
Simple_io_ioadr_reg <= Simple_io_ioadr;
Simple_io_outbus_reg <= Simple_io_outbus;
end
```

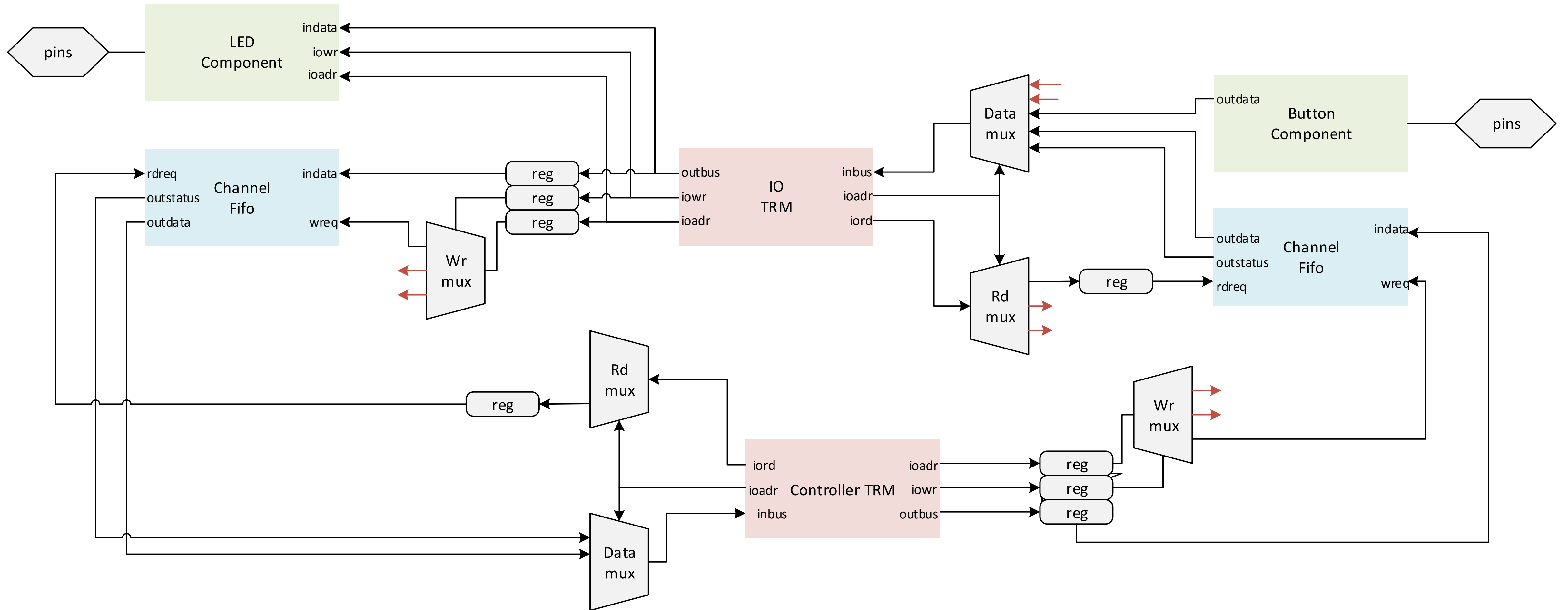


TRM inbound communication




```
reg Simple__Channel0_rdreq;
TRM ...
Button instSimple_IO_Button(.clk(clk), .rst(rst), .inData(Simple_IO_Button_inData), .outData(Simple_IO_Button_outData));
ParChannel #(.Size(32)) Simple__Channel0(.clk(clk), .rst(rst), .wreq(Simple__Channel0_wreq), .rdreq(Simple__Channel0_rdreq),
.inData(Simple__Channel0_inData), .outData(Simple__Channel0_outData), .status(Simple__Channel0_status));

assign Simple_io_inbus = (Simple_io_ioadr == 32)? Simple__Channel0_outData: (Simple_io_ioadr == 33)? Simple__Channel0_status:
((Simple_io_ioadr ==  7))? Simple_IO_Button_outData;

always @(posedge clk)
begin
    Simple__Channel0_rdreq <= ((Simple_io_ioadr == 32) & ~Simple__Channel0_rdreq & Simple_io_iord);
end
```

Putting it together



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
```

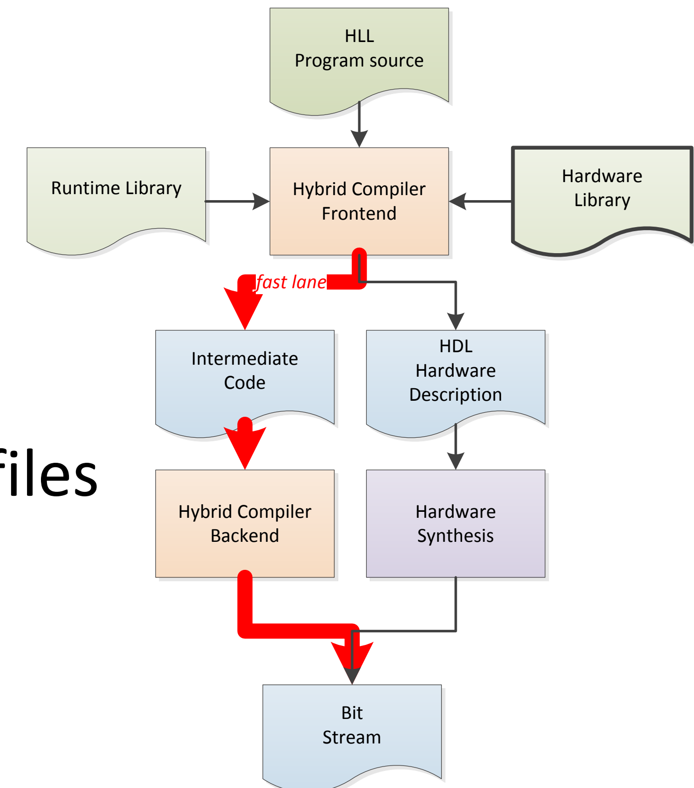
3. Call impact to upload to hardware

```
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



Spartan3 Board used in the labs

Resources

4-input LUTS	3840
Flip Flops	3840
BRAMs	12
DSPs	-

Resource Usage Scenarios

1 TRMS

```
CELLNET Game;

IMPORT LED;

TYPE

IO*=CELL {TRMS, DataMemorySize(512),
CodeMemorySize(1024), LED}
(in: PORT IN);
BEGIN
END IO;

VAR  io: IO;
BEGIN
   NEW(io);
END Game.
```

Resources

4-input LUTS	21%
Flip Flops	3%
BRAMs	16%

1 TRM

```
CELLNET Game;

IMPORT LED;

TYPE

IO*=CELL {DataMemorySize(512),
CodeMemorySize(1024), LED} (in: PORT
IN);
BEGIN
END IO;

VAR  io: IO;
BEGIN
   NEW(io);
END Game.
```

Resources

4-input LUTS	27%
Flip Flops	5%
BRAMs	16%

TRM →TRMS

```
VAR  io: IO; trm: TRM;
BEGIN
NEW(io); NEW(trm);
   CONNECT(trm.out, io.in);
END Game.
```

Resources

4-input LUTS	58%
Flip Flops	9%
BRAMs	33%

TRM ← →TRMS

```
VAR  io: IO; trm: TRM;
BEGIN
NEW(io); NEW(trm);
   CONNECT(trm.out, io.in);
END Game.
```

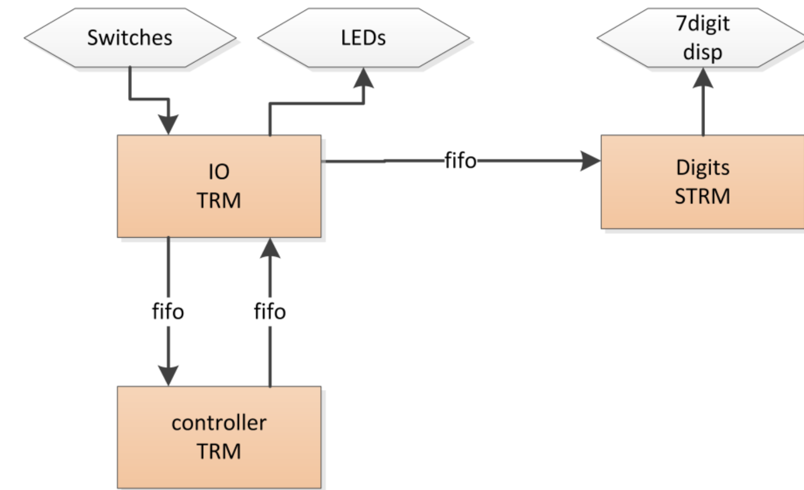
Resources

4-input LUTS	64%
Flip Flops	11%
BRAMs	33%

➔ TRM ≈ 21-27% (LUTs), 16% (BRAMs); Fifo ≈6% (LUTs)

Resource Usage (Lab)

```
VAR controller: Controller; io: IO; digits: Digits;
BEGIN
        NEW(controller); NEW(io); NEW(digits);
        CONNECT(controller.out, io.in);
        CONNECT(io.out, controller.in);
        CONNECT(io.digits, digits.in);
```



- 2 TRMs
- 1 TRMSs
  (simplified TRM, no Multiplier, no IRQs)
- 3 FIFOs

Resources

4-input LUTS	96%
Flip Flops	17%
BRAMs	50%