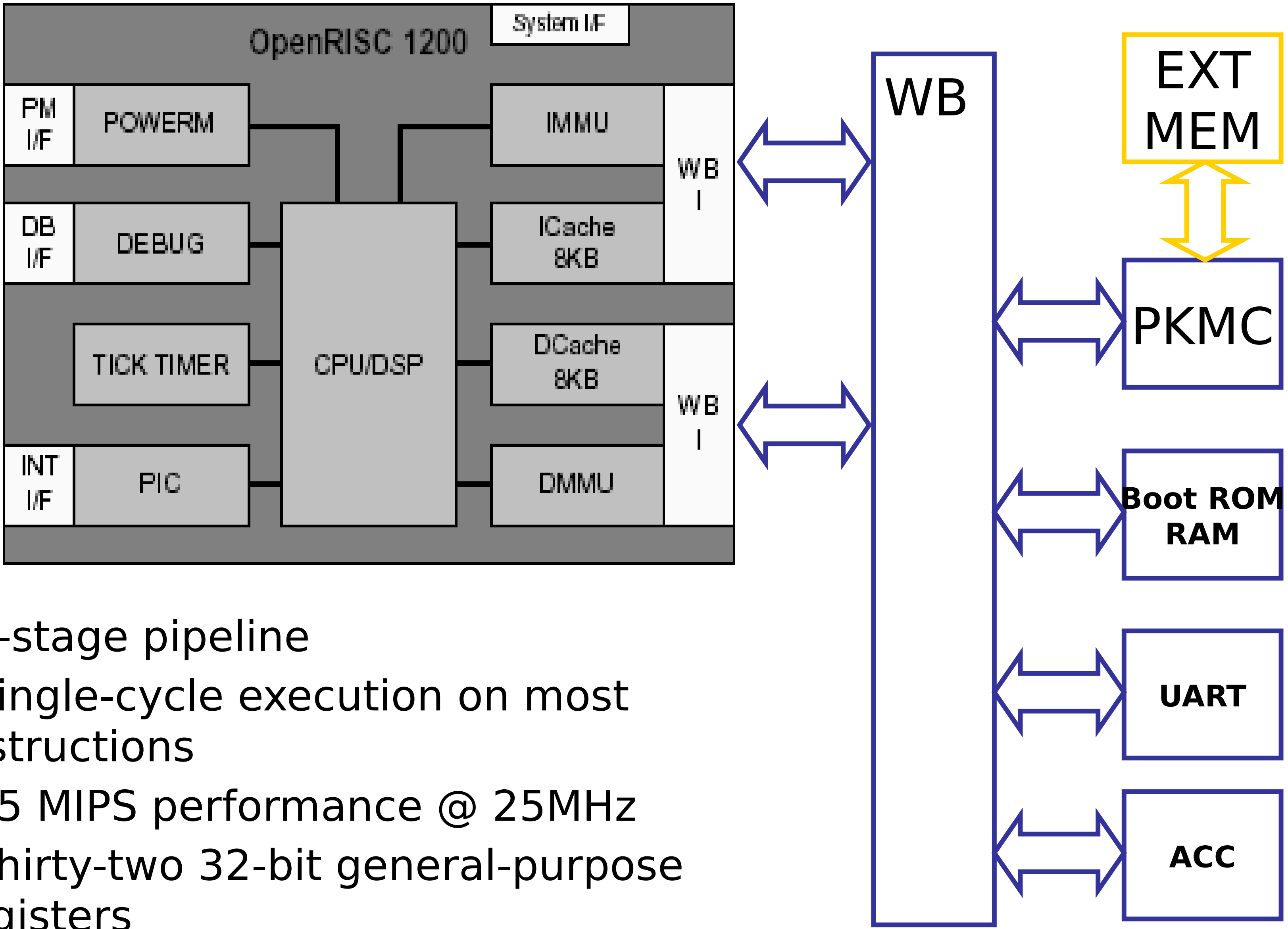
A soft CPU

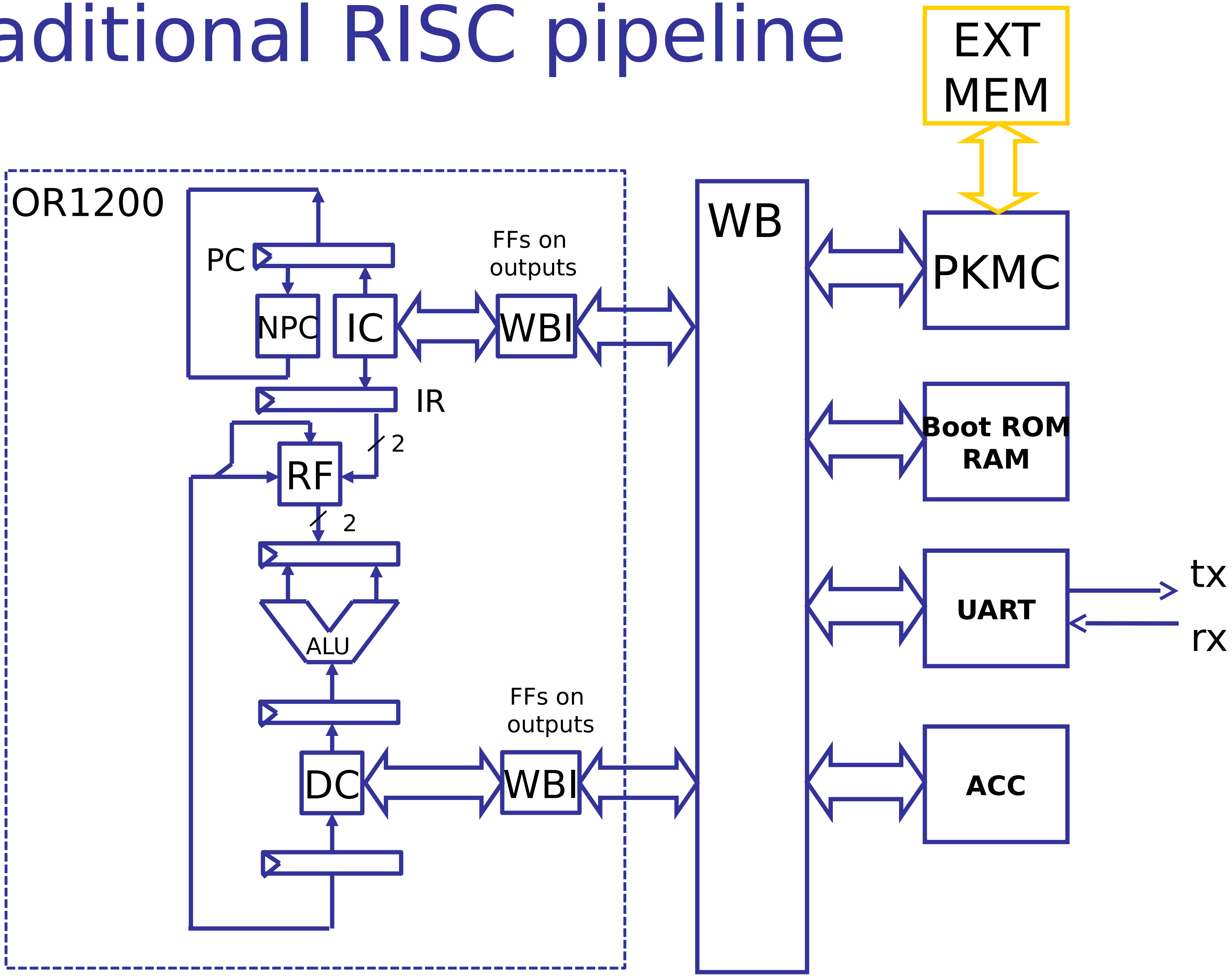
- OR1200
  - Architecture
  - Instruction set
  - C example
- Wishbone bus
  - cycles
  - arbitration
  - SV interface
  - Lab 1
- OR 1200
  - Pipelining ...

OpenRISC 1200 RISC Core



- 5-stage pipeline
- Single-cycle execution on most instructions
- 25 MIPS performance @ 25MHz
- Thirty-two 32-bit general-purpose registers

Traditional RISC pipeline



Instruction Set Architecture

IC and DC compete for the WB

- reduce usage of data memory
  - Many registers => 32 register à 32 bits
  - All arithmetic instructions access only registers
  - Only load/store access memory
- reduce usage of stack
  - save return address in link register **`r9`**
  - parameters to functions in registers

l.add Add

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10	9 8	7 . . 4	3 . . 0
opcode 0x38	D	A	B	reserved	opcode 0x0	reserved	opcode 0x0
6 bits	5 bits	5 bits	5 bits	1 bits	2 bits	4 bits	4bits

```
l.add rD,rA,rB        ; rD = rA + rB
                      ; SR[CY] = carry
                      ; SR[OV] = overflow
```

l.lw Load Word

31 26	25 . . . 21	20 . . . 16	15 0
opcode 0x21	D	A	I
6 bits	5 bits	5 bits	16bits

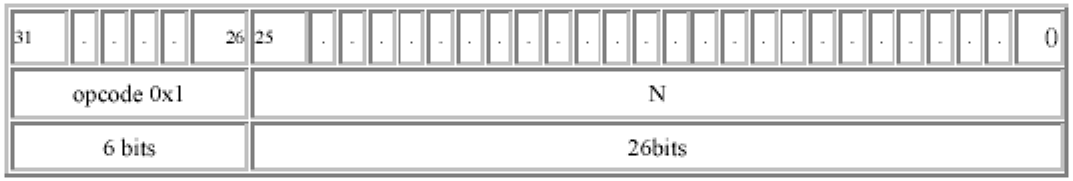
```
l.lw rD,I(rA)        ; rD = M(exts(I) + rA)
```

A piece of code

```
l.movhi r3,0x1234    // r3 = 0x1234_0000

l.ori   r3,r3,0x5678 // r3 |=  0x0000_5678

l.lw    r5,0x5(r3)   // r5 = M(0x1234_567d)

l.sfeq   r5,r0    // set conditional branch
   // flag SR[F] if r5==0

l.bf somewhere       // jump if SR[F]==1

l.nop             // 1 delay slot, always executed

...
```

Subroutine jump

`l.jal` **Jump and Link**



31 26	25 0
opcode 0x1	N
6 bits	26bits

Format:
`l.jal N`

Description:
The immediate value is shifted left two bits, sign-extended to program counter width, and then added to the address of the jump instruction. The result is the effective address of the jump. The program unconditionally jumps to EA with a delay of one instruction. **The address of the instruction after the delay slot is placed in the link register**.

32-bit Implementation:

```
PC = exts(Immediate < < 2) + JumpInsnAddr
LR = DelayInsnAddr + 4
```

Register usage
ABI=Application binary interface

R11	RV function return value
R9	LR (link register)
R3-R8	Function parameters 0-5
R2	FP (frame pointer)
R1	SP (stack pointer)
R0	=0

Subroutine jump

- In this implementation LR (link register) is r9
- A leaf function (no further subroutine calls) does not use the stack

uart is a leaf function

```
    l.jal  _uart
    l.nop
 l.xxx

_uart:
 ...
 l.jr r9
    l.nop
```

fun1 is not a leaf function

```
   l.jal  _fun1
   l.nop
   l.xxx

_fun1:
   l.addi r1,r1,0xfffc
   l.sw  0x0(r1),r9
   l.jal _uart
   l.nop
   l.lwz  r9,0x0(r1)
   l.jr r9
   l.addi r1,r1,0x4
```

A very simple C example

```
int sum(int a, int b)
{

  return(a+b);
}
```

```
l.add r3,r3,r4               ; a = a+b
l.ori r11,r3,0x0             ; rv = a
l.jr r9                      ; return
l.nop
```

```
int main(void)
{
  int a=1,b=2, nr;

  nr = sum(a,b);

  return(nr);
}
```

```
l.addi r1,r1,0xfffffffc         ; sp -= 4
l.sw 0x0(r1),r9                 ; M(sp)= lr

l.addi r3,r0,0x1                ; a = 1
l.jal  _sum
l.addi r4,r0,0x2                ; b = 2

l.lwz r9,0x0(r1)                ; lr = M(sp)

l.jr r9          ; return
l.addi r1,r1,0x4                ; sp += 4
```

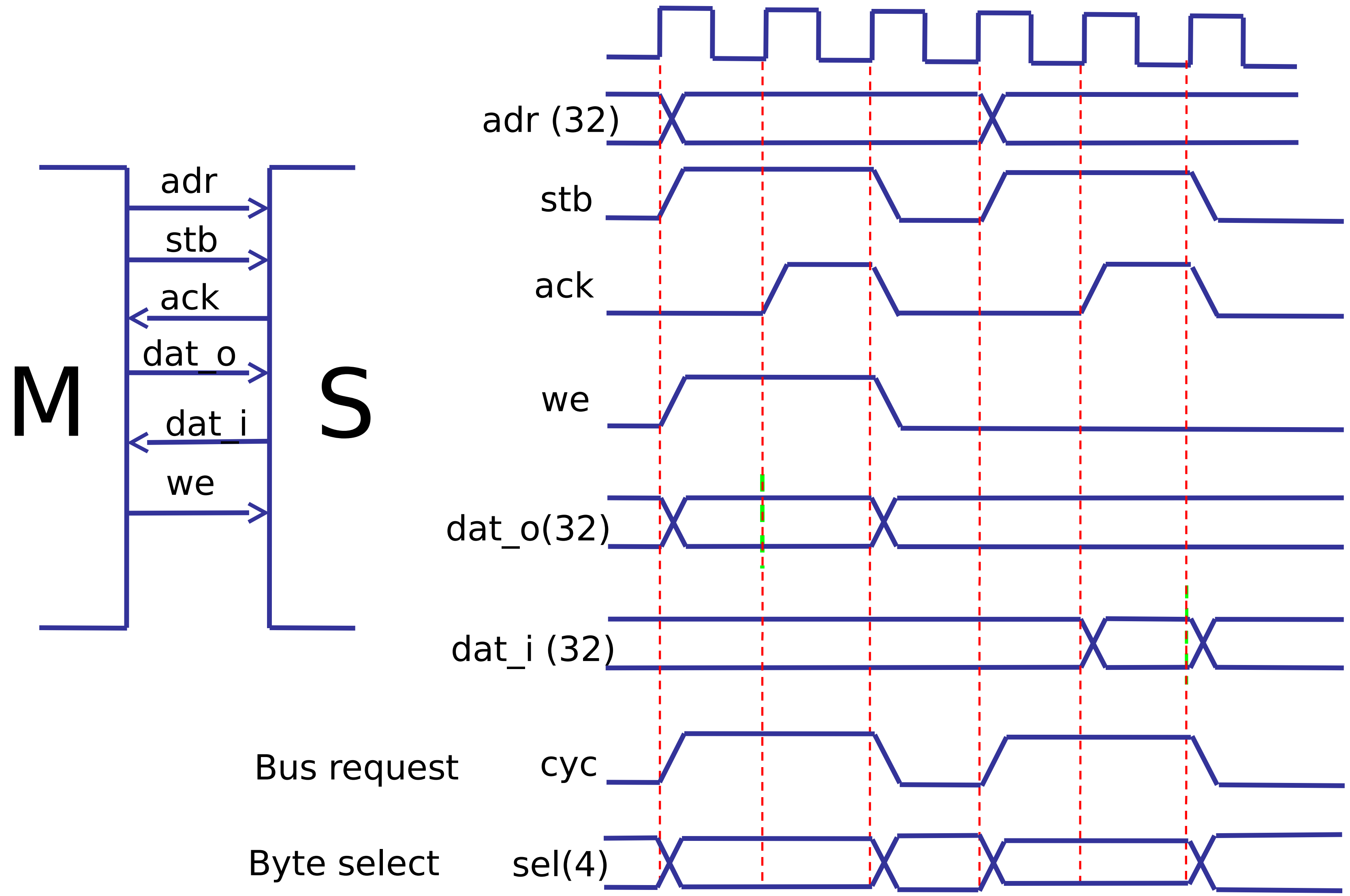
The Wishbone Interconnect

Some features:

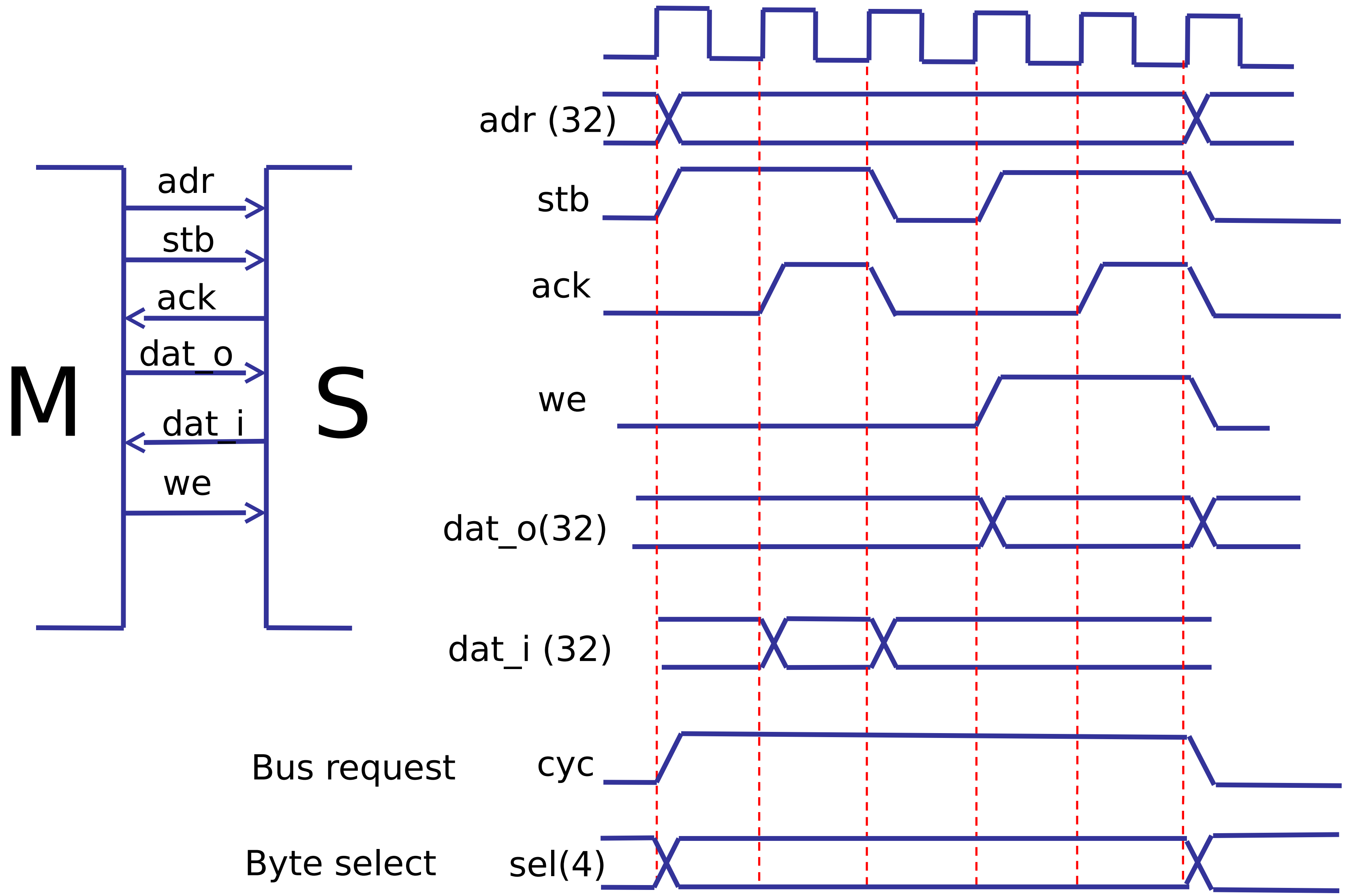
- Intended as a standard for connection of IP cores
- Full set of popular data transfer bus protocols including:
  - READ/WRITE cycle
  - RMW cycle
  - Burst cycles
- Variable core interconnection methods support point-to-point, shared bus, and crossbar switch.
- Arbitration methodology is defined by the end user (priority arbiter, round-robin arbiter, etc.)

WISHBONE COMPATIBLE

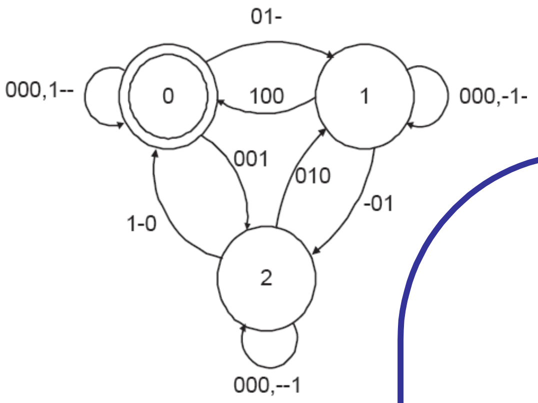
Simple Wishbone cycles

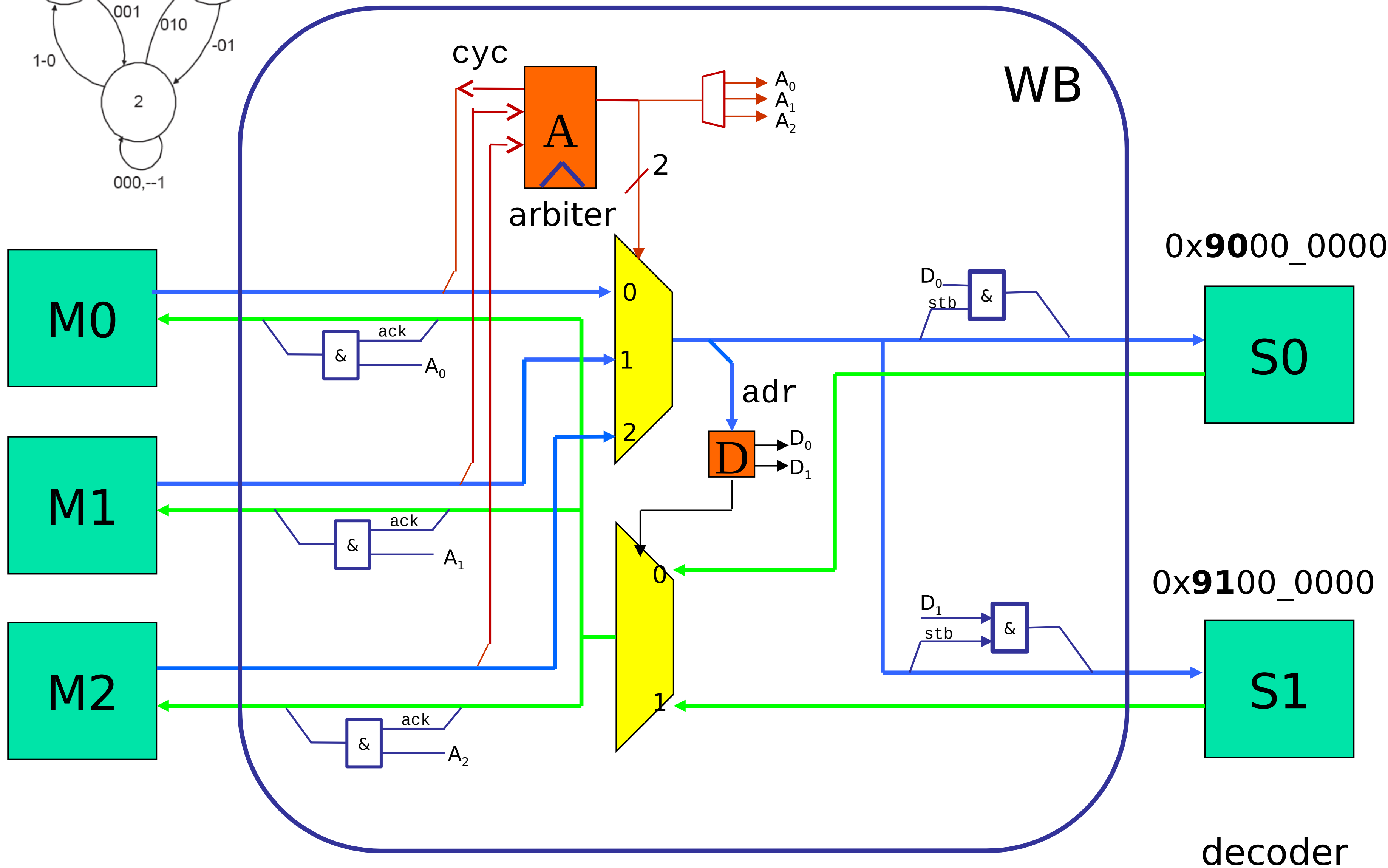


Read-modify-write cycle



Wishbone bus (3M,2S)

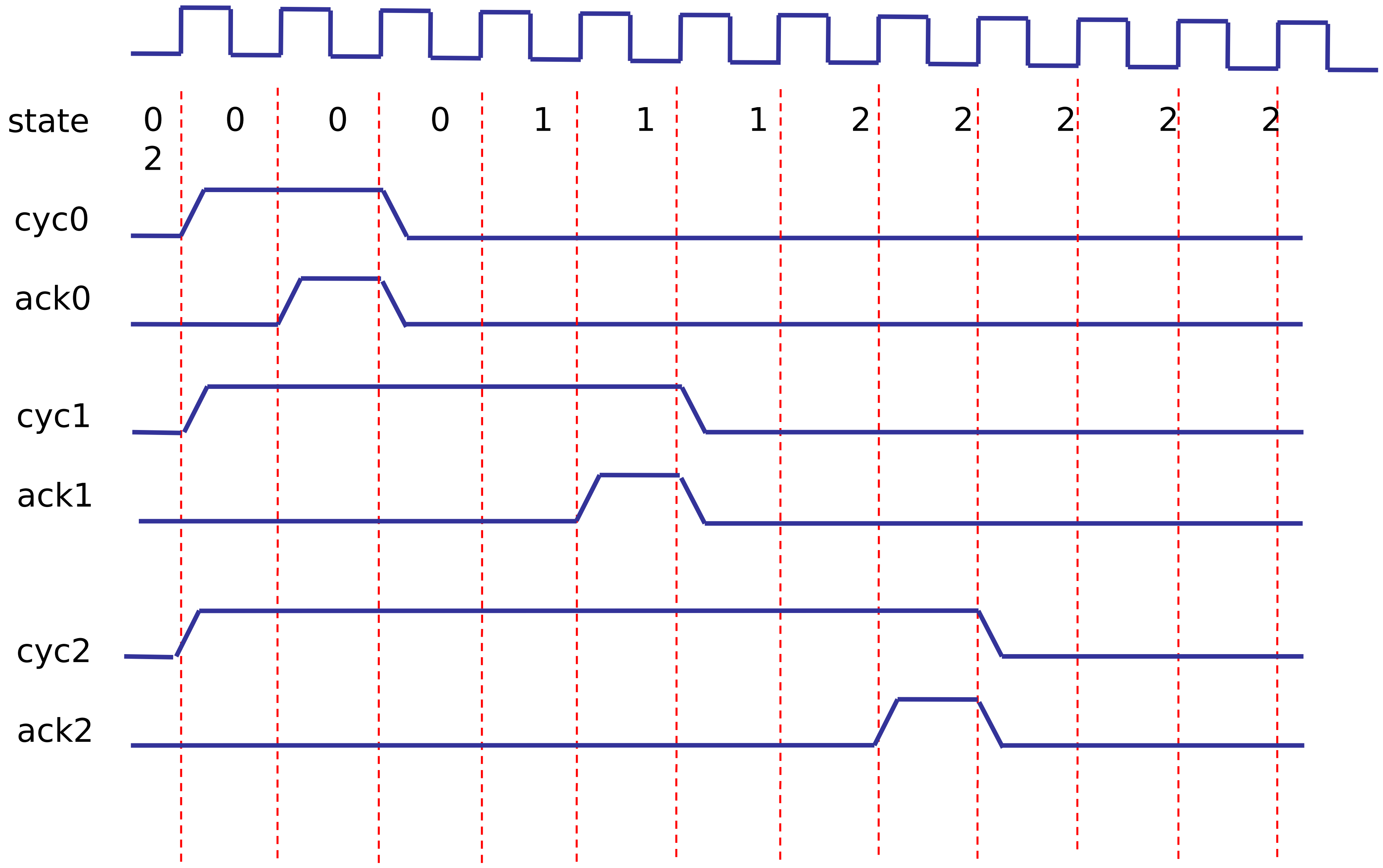




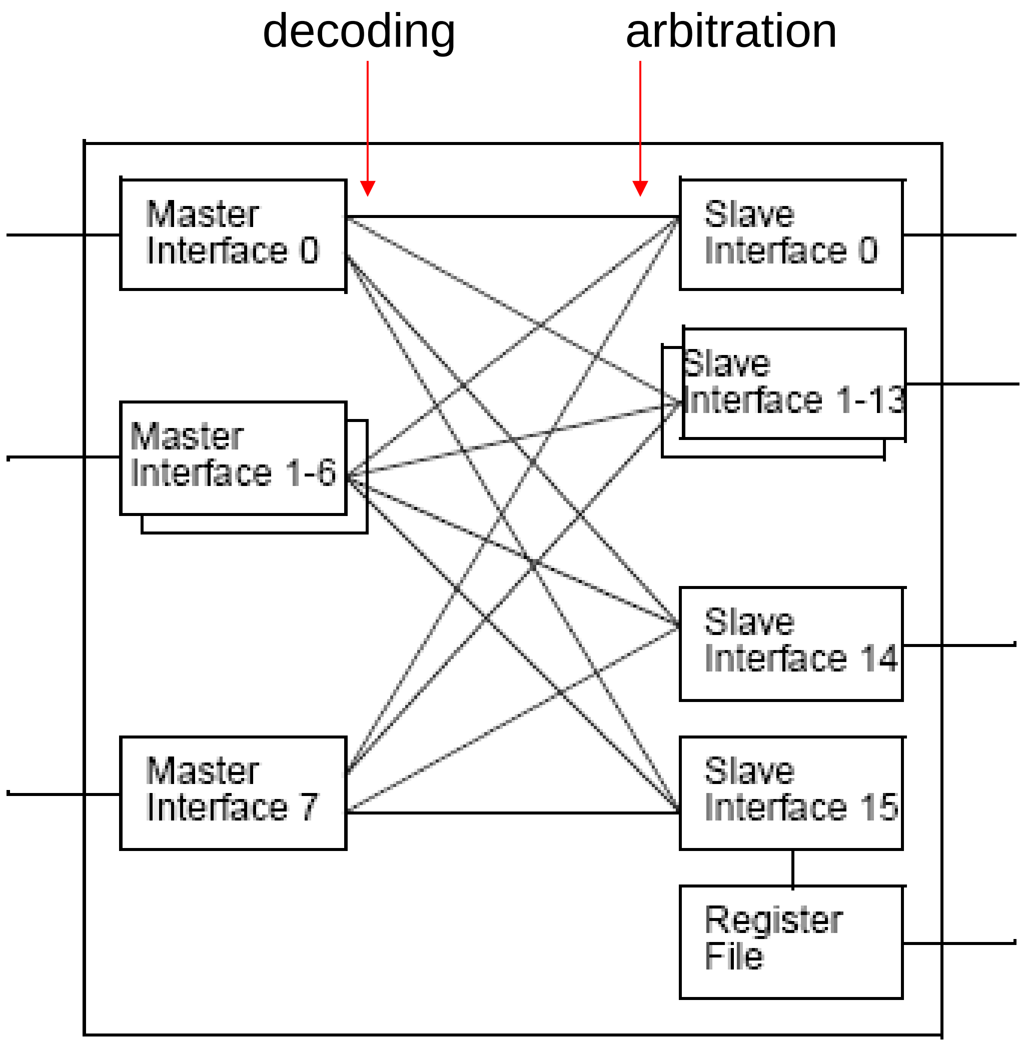
→ adr[31:0], dat_o[31:0], stb, cyc, we, …

← dat_i[31:0], ack, …

Wishbone cycles with arbitration

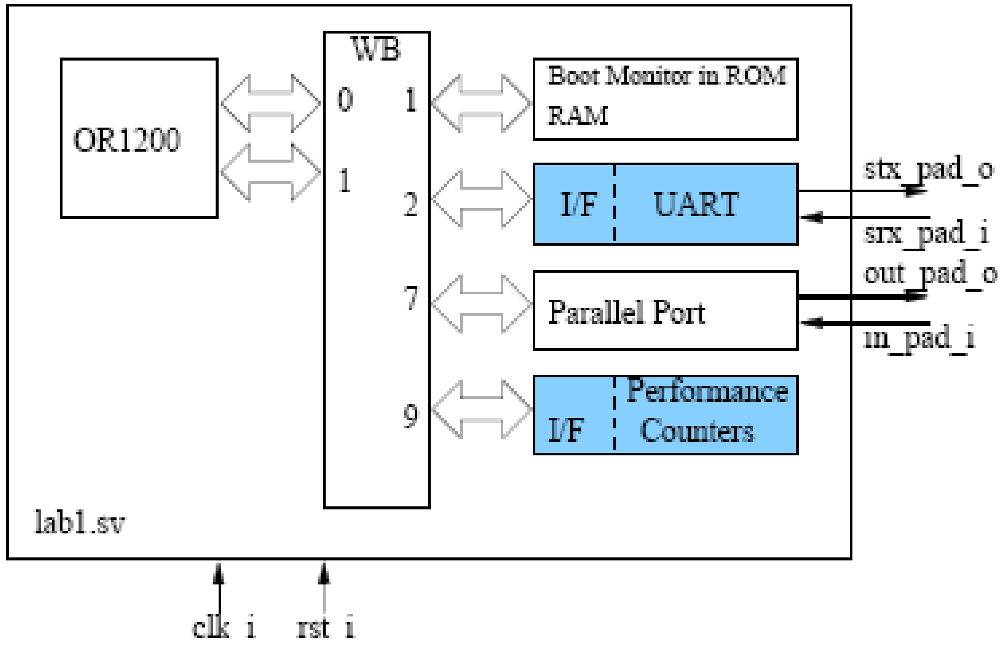


Wishbone - Crossbar



The Slave Interface consists of a WISHBONE Master that can connect to an external WISHBONE Slave. When an internal Master Interface starts a WISHBONE cycle, the Slave Interface will select the appropriate Master Interface, based on internal arbitration.

Lab 1



1. modify your UART from the previous lab and interface it to the Wishbone bus

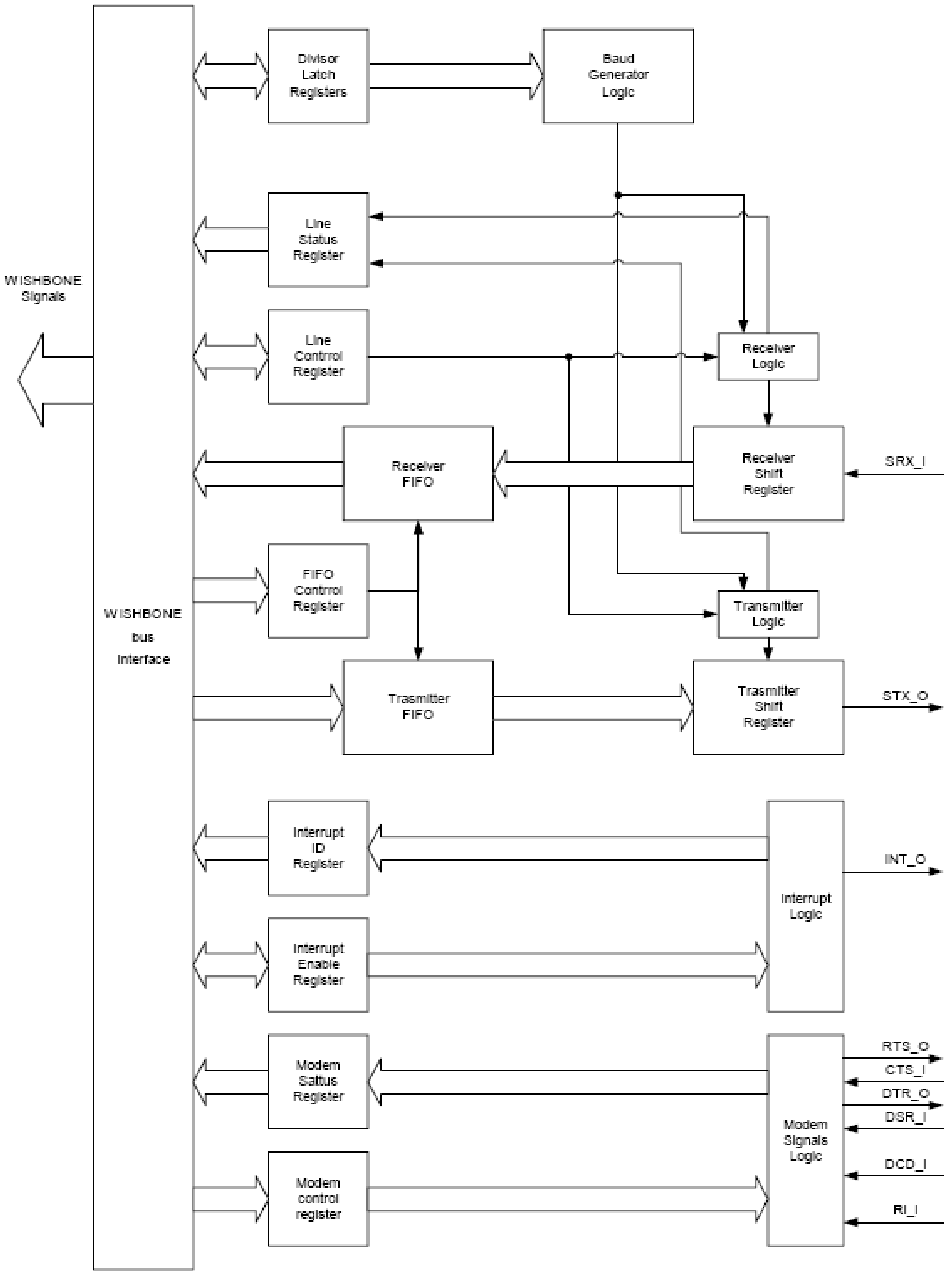
2. check the UART device drivers in the boot monitor. Your UART will replace an existing UART 16550.

3. download and execute a benchmark program, that performs (the DCT part of) JPEG compression on a small image in your RAM module.

4. simulate the computer running the benchmark program.

5. design a module containing hardware performance counters.

uart 16550



rx

tx

int

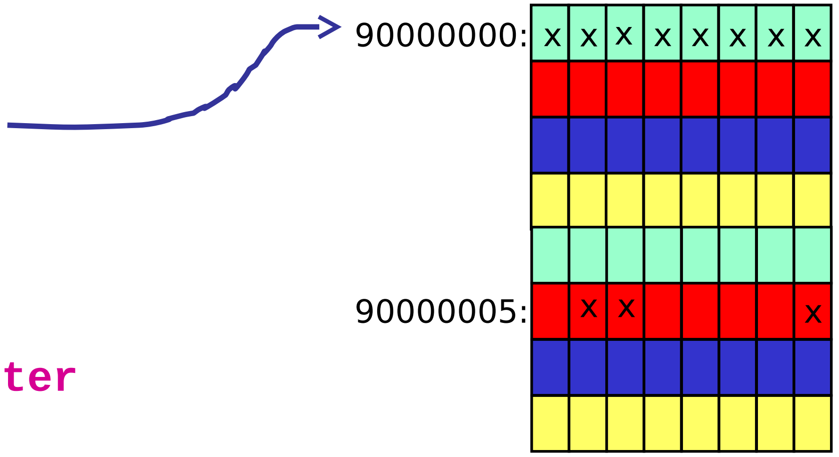
- Compatible with industry standard 16550
- 16 character FIFOS for rx and tx
- seven 8-bit registers on a 32-bit bus

}

Not used on our board

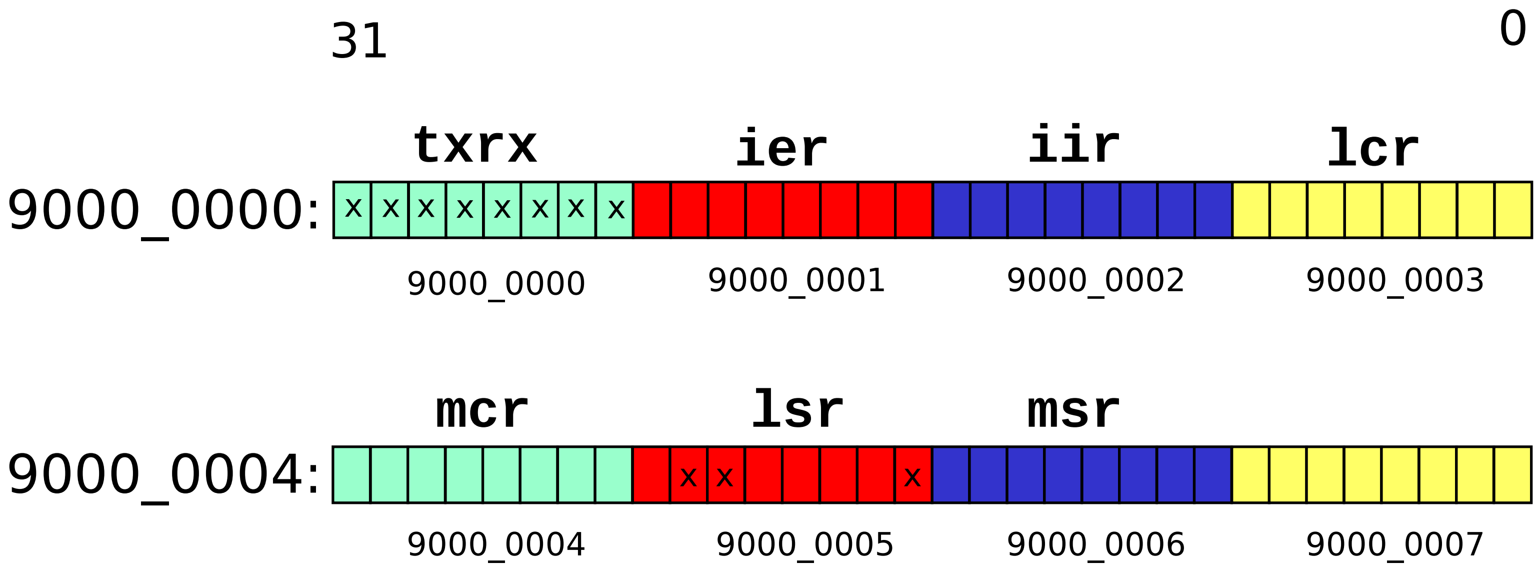
uart 16550-driver in the monitor

```
typedef struct
{
    unsigned char txrx;    // 0. transmit(W), receive(R)
    unsigned char ier;     // 1. interrupt enable (RW)
    unsigned char iir;     // 2. interrupt flags(R),FIFO ctrl(W)
    unsigned char lcr;     // 3. line control (RW)
    unsigned char mcr;     // 4. modem control (W)
    unsigned char lsr;     // 5. line status (R)
    unsigned char msr;     // 6. modem status (R)
} UART;

UART volatile *pu = (UART *) 0x90000000;
int c;

…

// It takes 2170 clocks to tx/rx a character
// with 25 MHz clock. Handshake needed!

while (!(pu->lsr & 0x01)) // has the character been received?
  c = (int) pu->txrx;     // read char

while (!(pu->lsr & 0x60))           // has the character been transmitted
  pu->txrx = (unsigned char) c;     // write new char
```

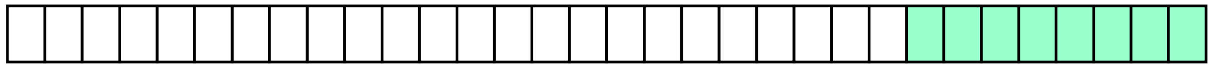


or1200 is a 32-bit machine

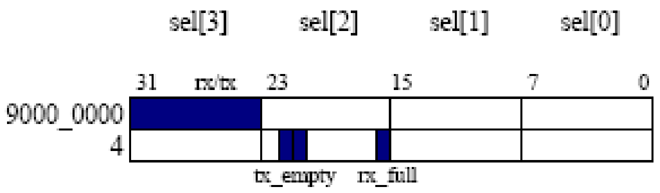
The bytes are ordered like this (big-endian):



```
c = (int) pu->txrx;  // read char
```



Programmers model



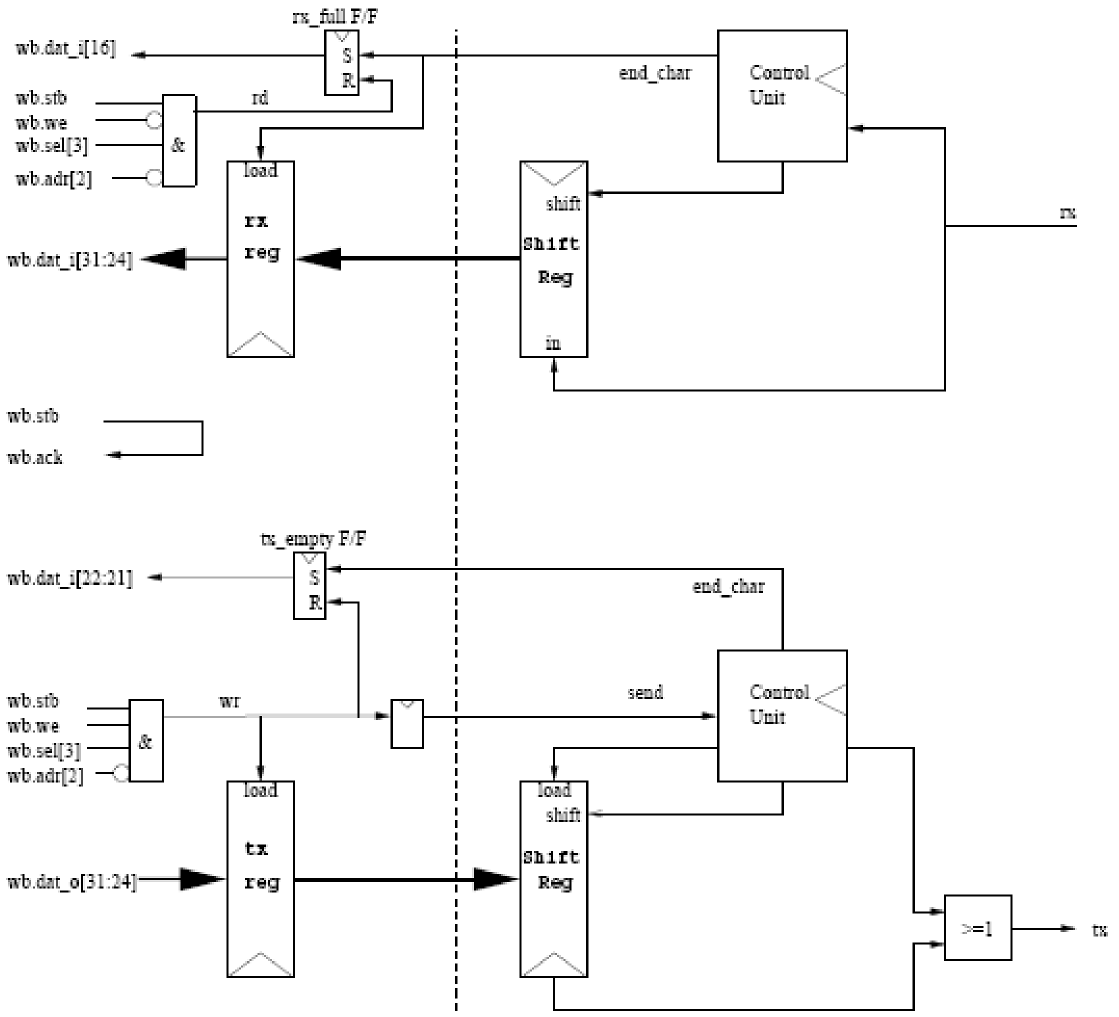
Existing driver wants this

Handshake F/Fs

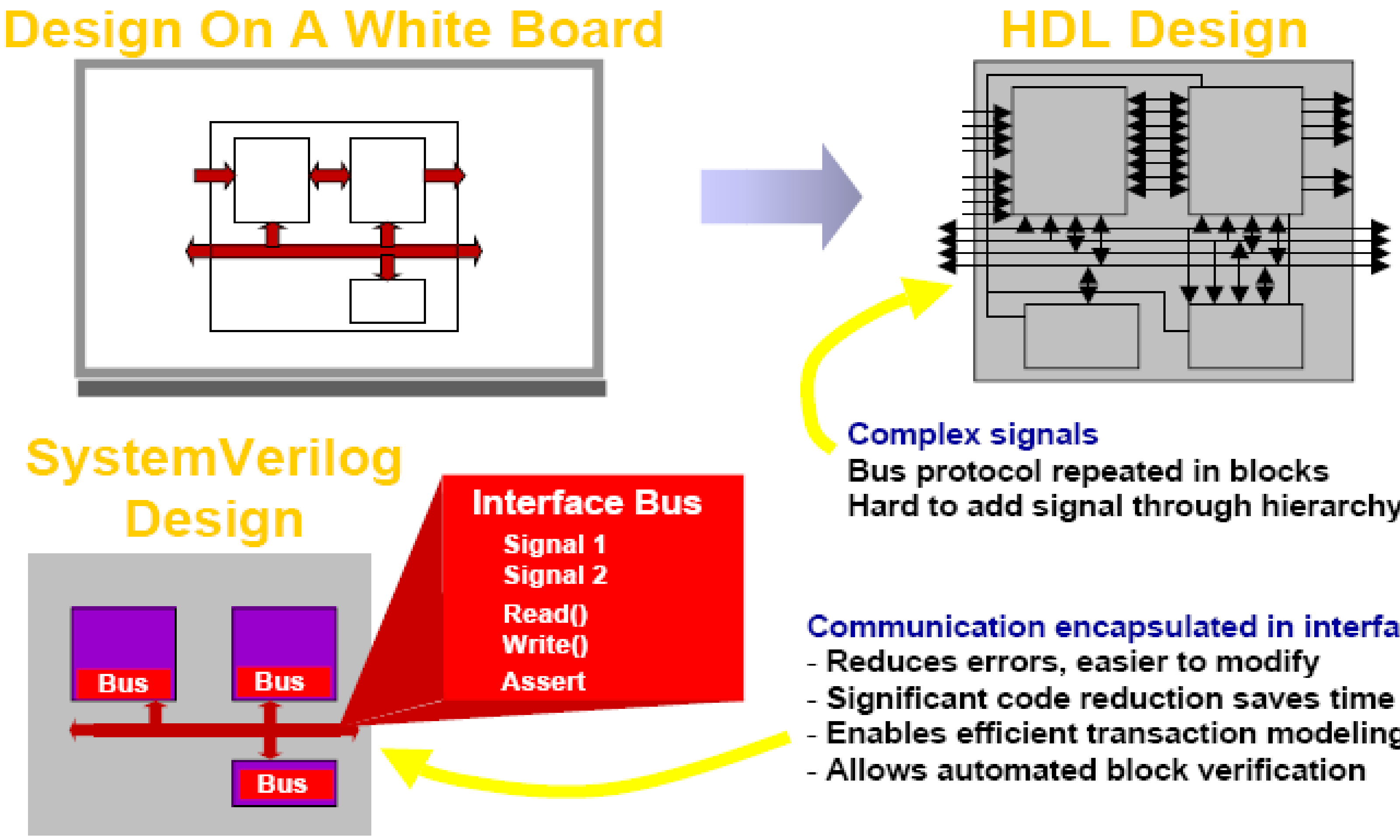
rx_full Set: end_char
Reset: read txrx

tx_empty Set: end_char
Reset: write txrx

Block diagram



SystemVerilog – Interfaces ”a bundle of wires”

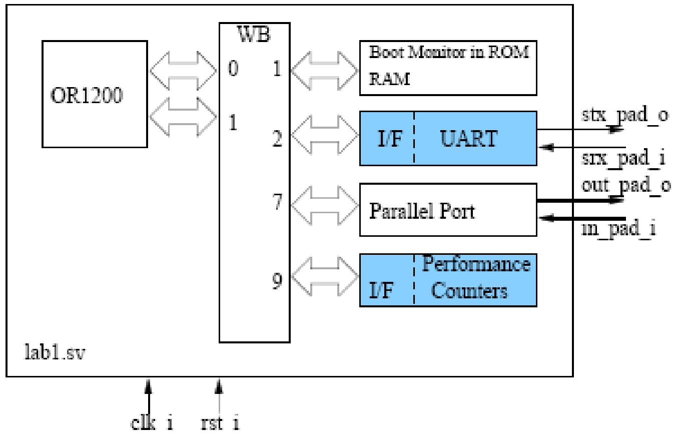


Complex signals
Bus protocol repeated in blocks
Hard to add signal through hierarchy

Communication encapsulated in interface
- Reduces errors, easier to modify
- Significant code reduction saves time
- Enables efficient transaction modeling
- Allows automated block verification

```
interface wishbone(input logic clk, rst);
   typedef logic [31:0] adr_t;
   typedef logic [31:0] dat_t;

   adr_t       adr; // address bus
   dat_t       dat_o;    // write data bus
   dat_t       dat_i;    // read data bus
   logic       stb; // strobe
   logic       cyc; // cycle valid
   logic       we;  // indicates write transfer
   logic [3:0]      sel; // byte select
   logic       ack; // normal termination
   logic       err; // termination w/ error
   logic       rty; // termination w/ retry
   logic       cab; //
   logic [2:0]      cti; // cycle type identifier
   logic [1:0]      bte; // burst type extension


   modport master(
      output      adr, dat_o, stb, cyc, we, sel, cab, cti, bte,
      input       clk, rst, dat_i, ack, err, rty);

   modport slave(
      input       clk, rst, adr, dat_o, stb, cyc, we, sel, cab, cti, bte,
      output      dat_i, ack, err, rty);

   modport monitor(
      input       clk, rst, adr, dat_o, stb, cyc, we, sel, cab, cti, bte,dat_i,
                       ack, err, rty);

endinterface: wishbone
```

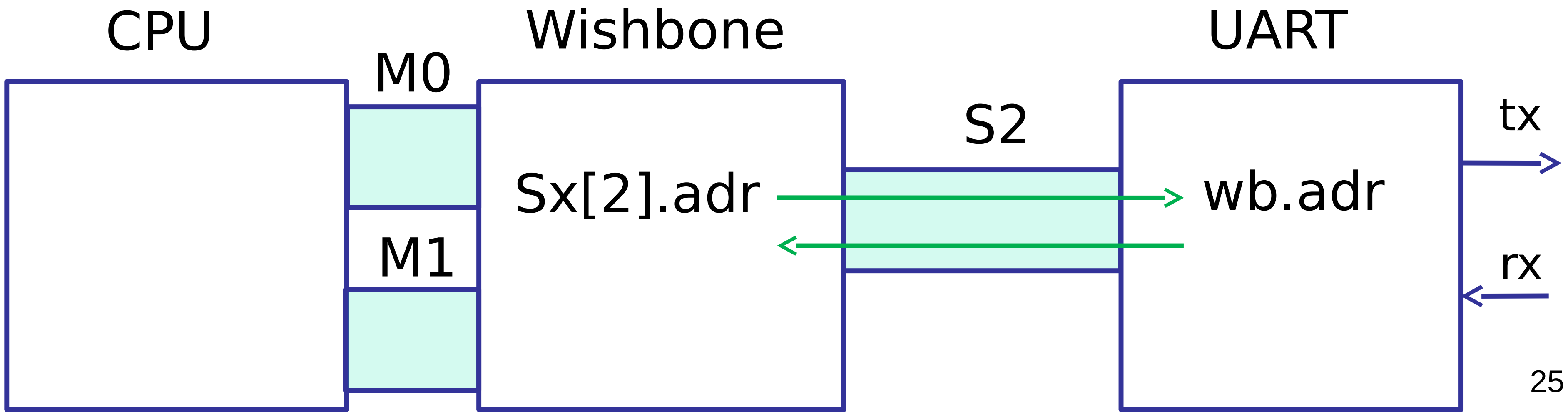
Top file: lab1.sv



```
module lab1
  (input clk, rst,
   output tx,
   input  rx);

   wishbone m0(clk,rst), m1(clk,rst),
            s1(clk,rst), s2(clk,rst), s7(clk,rst), s9(clk,rst);

   or1200_top cpu(.m0(m0), .m1(m1), …);

   wb_top w0(.*);

   romram rom0(s1);

   lab1_uart my_uart(.wb(s2), .int_o(uart_int),
                     .stx_pad_o(tx),.srx_pad_i(rx));


   …

endmodule
```

In the wishbone end wb/wb_top.sv

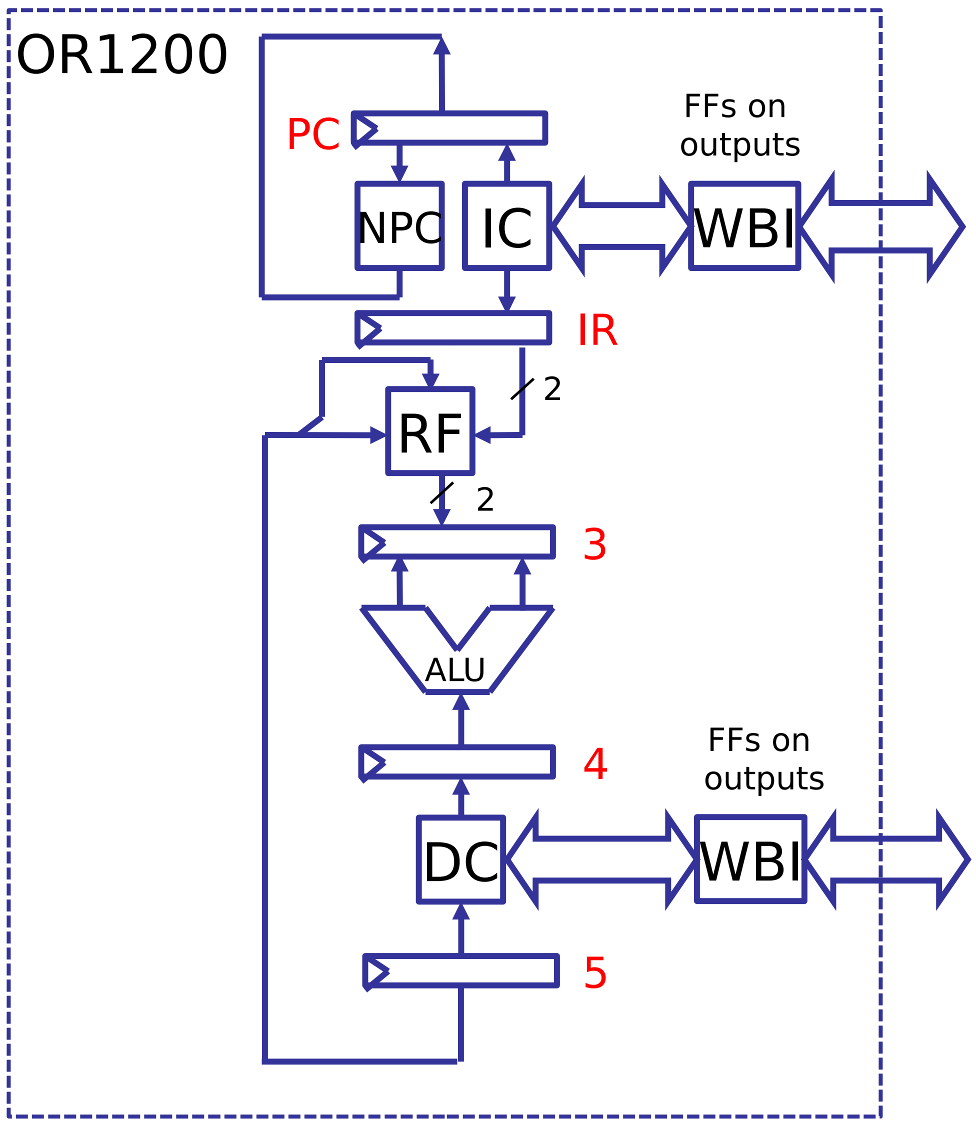
```
module wb_top(
  input clk_i, rst_i,

  // Connect to Masters
  wishbone.slave Mx[0:`Nm-1],

  // Connect to Slaves
  wishbone.master Sx[0:`Ns-1]
  );
```



In the UART end: lab1/lab1_uart_top.sv

```
module lab1_uart_top
  (wishbone.slave wb,
    output int_o,
    input srx_pad_i,
    output stx_pad_o);

   assign int_o = 1'b0;  // Interrupt, not used in this lab

   // Here you must instantiate lab0_uart
   // You will also have to change the interface of
   // lab0_uart to make this work.
   assign wb.dat_i = 32’h0;
   assign wb.ack = 1’b0 ;
   assign wb.err = wb.stb;


   assign stx_pad_o = srx_pad_i; // Change this line.. :)
endmodule
```

Pipeline & diagram



```
 0: ld r1,0xb(r3)
 4: add
 8: sub
12: xxx
```

PC	IR	3	4	5
4	ld			
8	add	ld		
12	sub	add	ld	
16	xxx	sub	add	ld

Pipelining

```
l.add r3,r2,r1
```

- fetch from IC (M)
- read r2,r1 from RF
- add
- write back r3 to RF

```
l.lwz r3,0xb(r1)
```

- fetch from IC
- read r1 from RF
- add r1 + 0xb
- read operand from DC (M)
- write back r3

```
l.sw 0xb(r1),r3
```

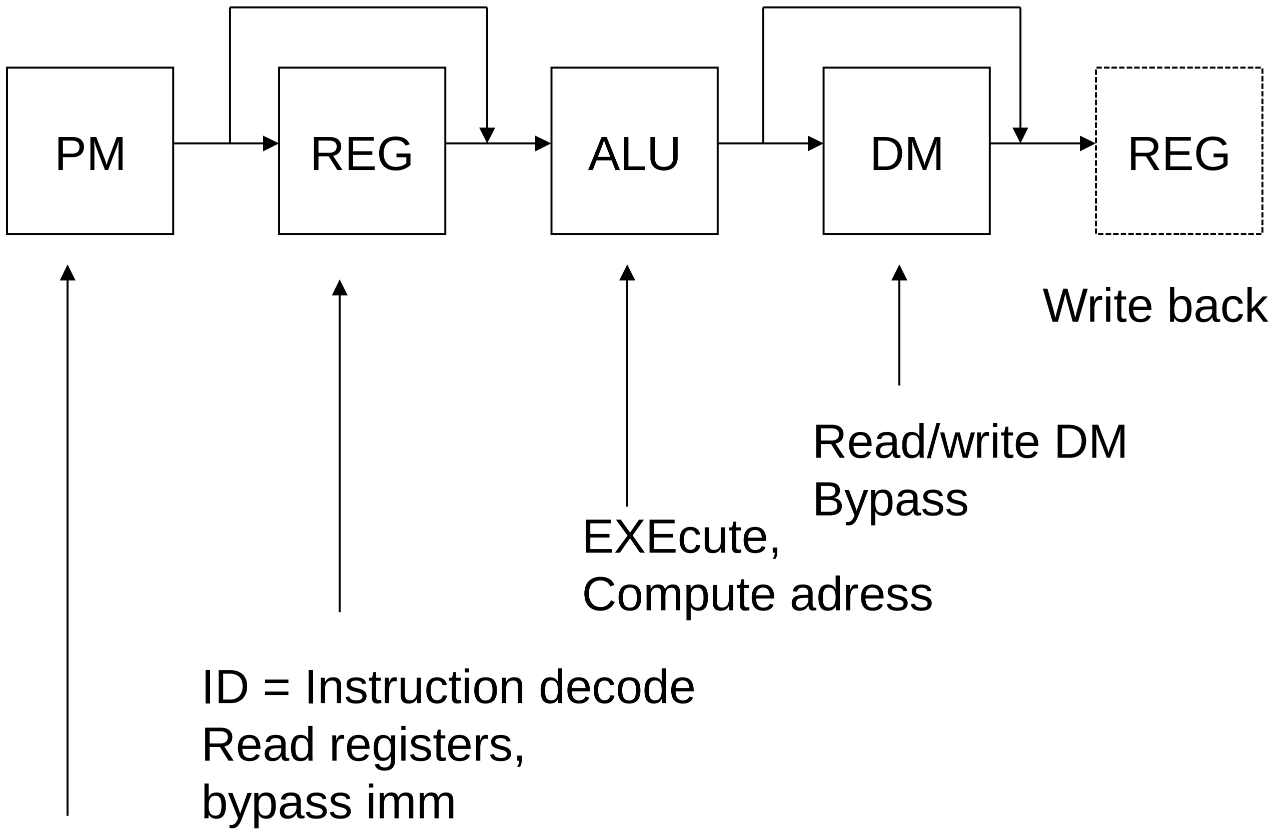
- fetch from IC
- read r1,r3 from RF
- add r1 + 0xb
- write operand to DC

Classic RISC pipeline

	PM	RF	ALU	DM
PC	IR	3	4	5
1	ld			
2	add	ld		
3	sub	add	ld	
4		sub	add	ld
5			sub	add
6				sub
7				

- add,sub do nothing in the DM stage
- Instruction decode, read registers simultaneously

The standard pipeline



Write back register value

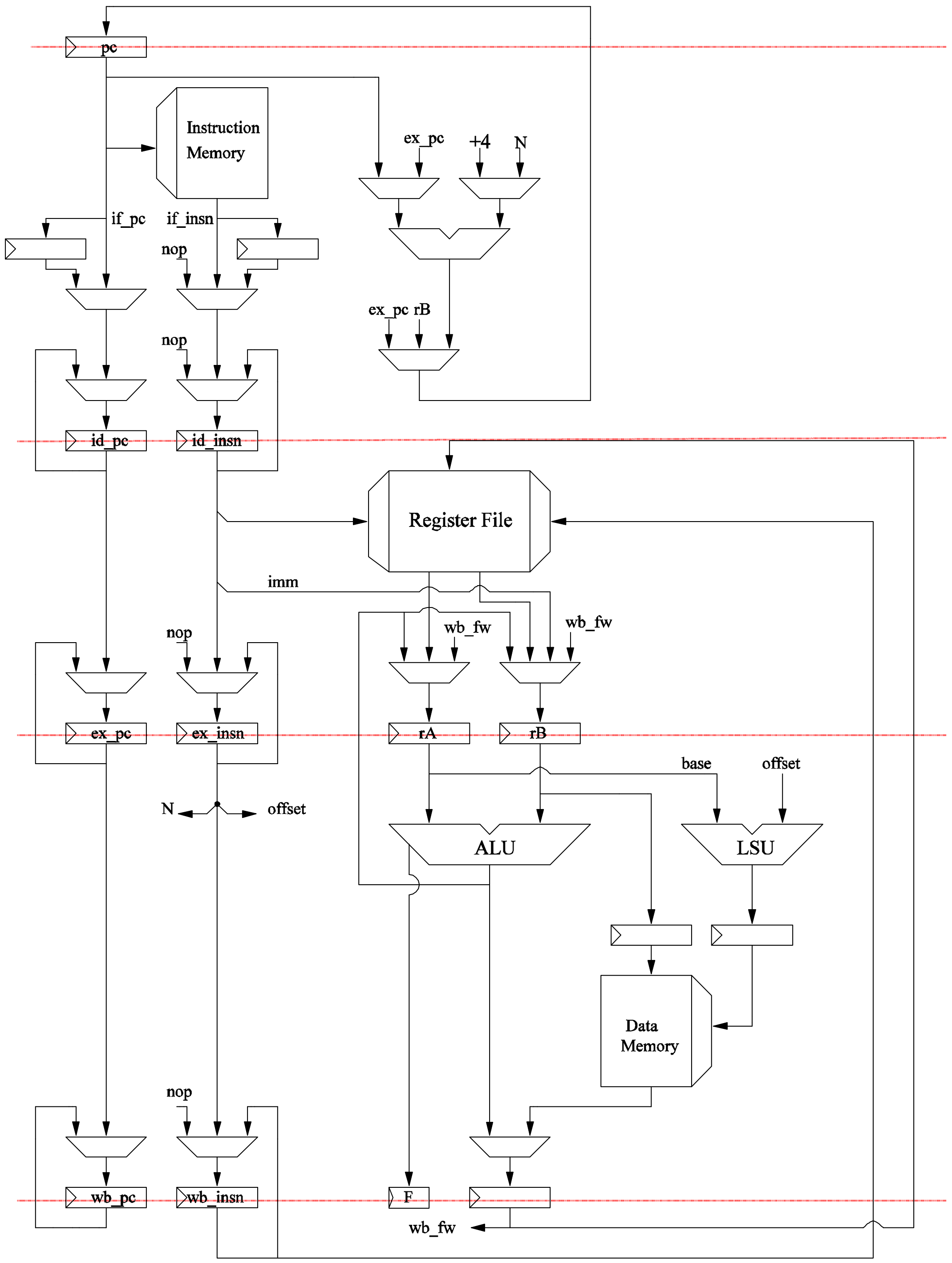
Read/write DM
Bypass

EXEcute,
Compute adress

ID = Instruction decode
Read registers,
bypass imm

IF = Instruction fetch,
compute next PC

or1200 pipeline – behavioral



IF – Instruction Fetch,

generate next PC

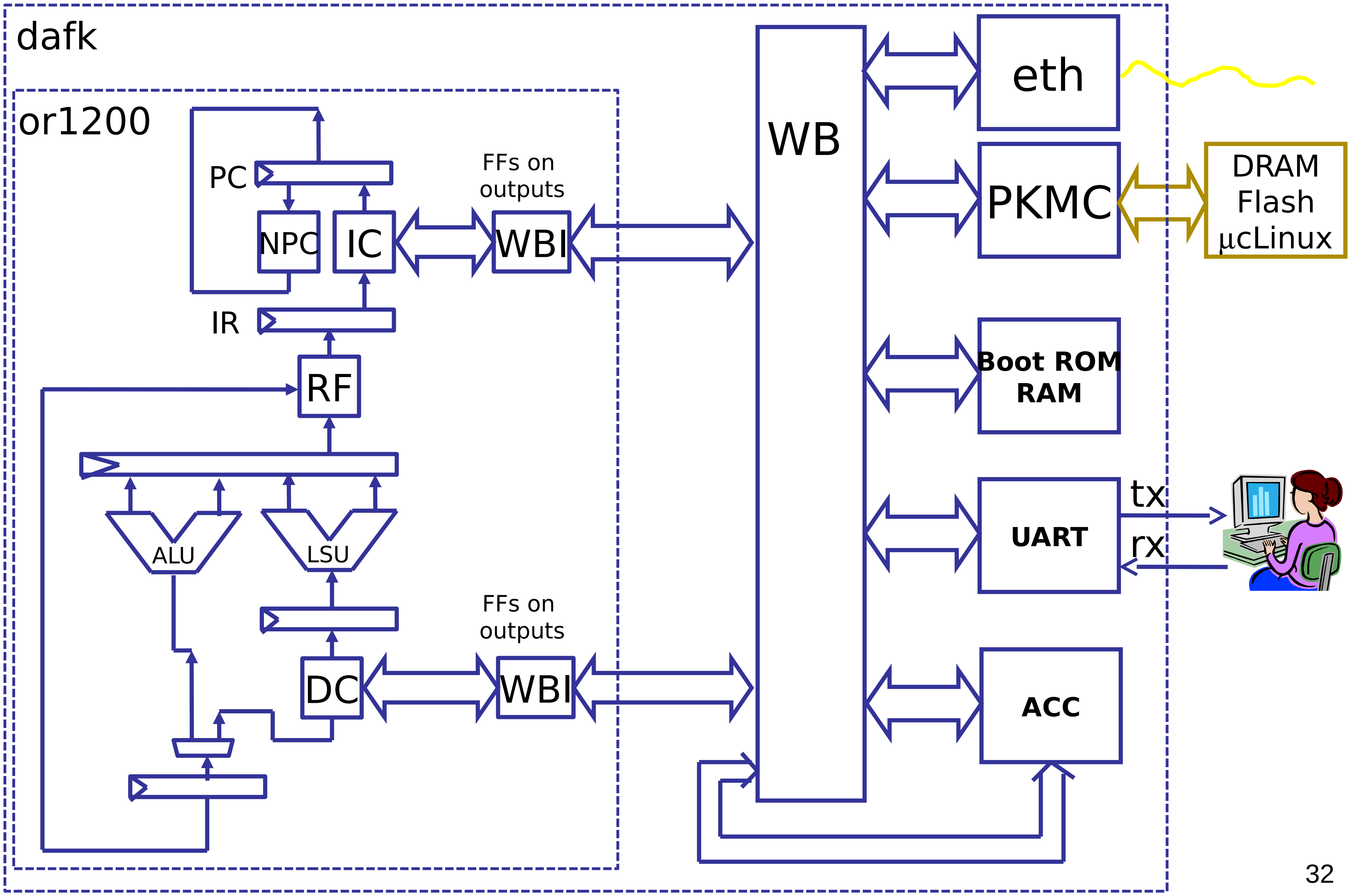
ID – Instruction Decode

Read registers

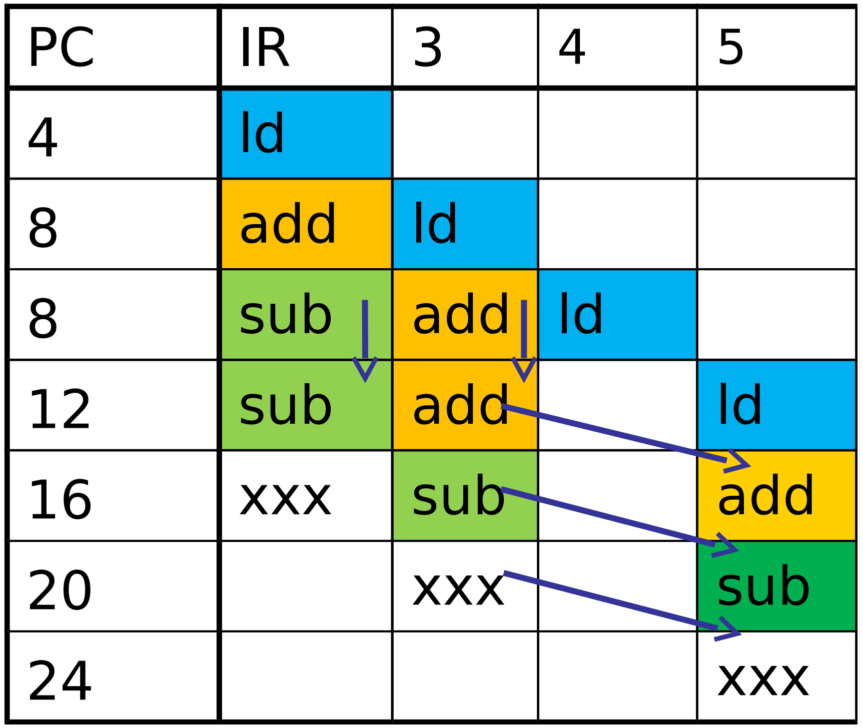
EX – Instruction Execute,
access DM

WB – Write back register

Our RISC pipeline



OR1200 pipeline



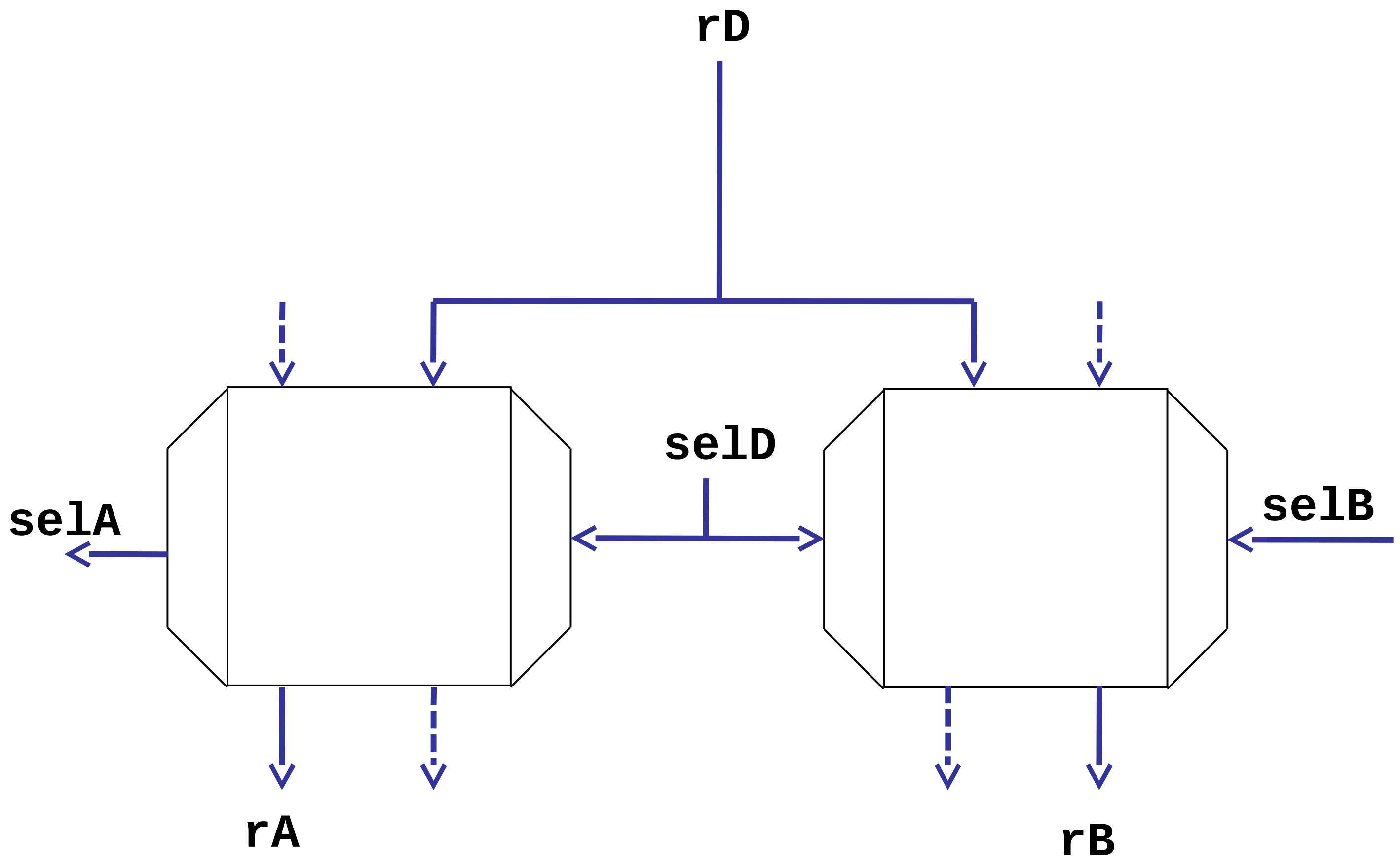
PC	IR	3	4	5
4	ld			
8	add	ld		
8	sub	add	ld	
12	sub	add		ld
16	xxx	sub		add
20		xxx		sub
24				xxx

Block RAM 512x32 simulation model

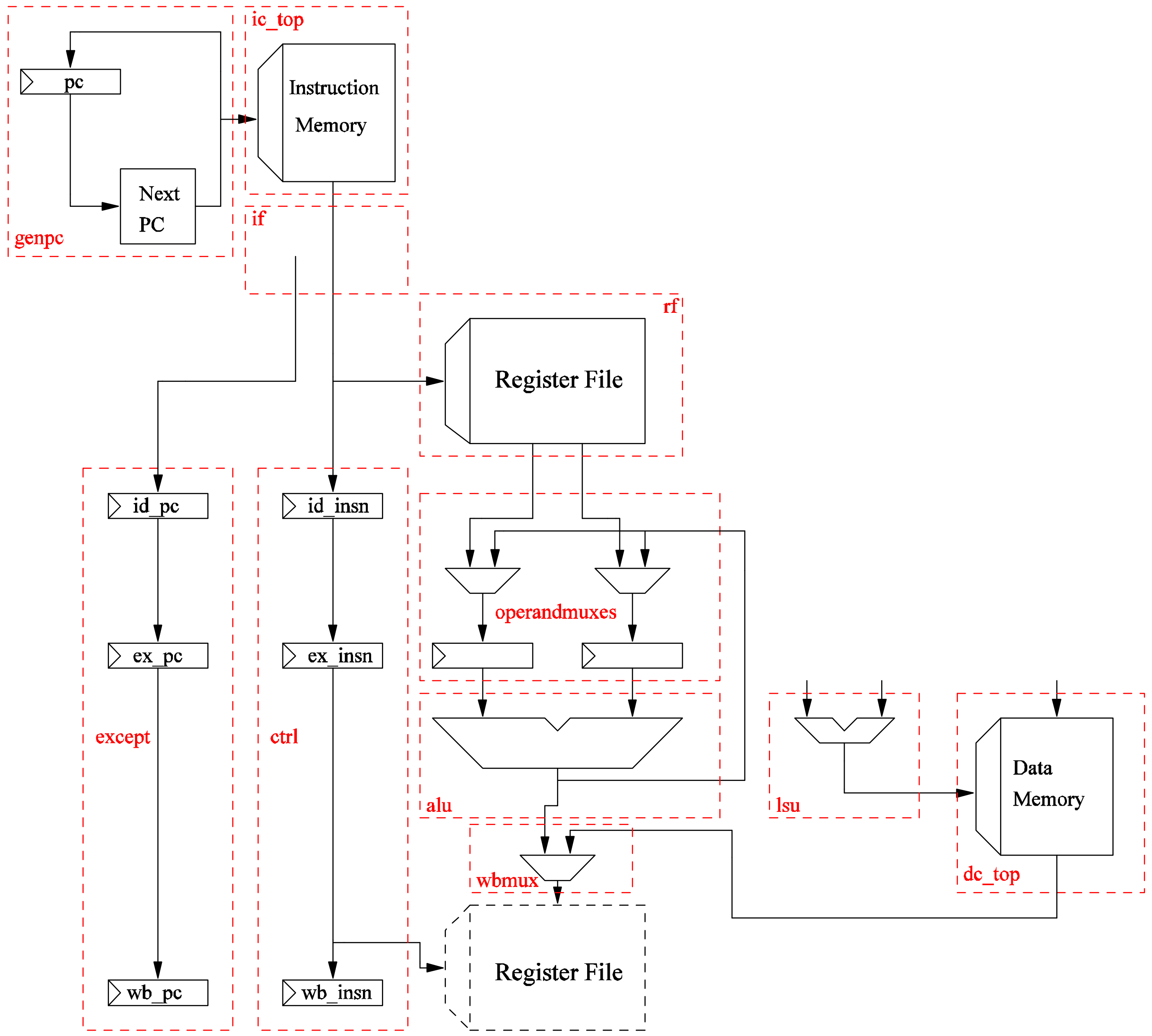
```
// Generic single-port synchronous RAM model
module (input clk,we,ce,oe,
        input [8:0] addr,
        input [31:0] di,
        output [31:0] doq);

// Generic RAM's registers and wires
reg [31:0] mem [0:511];     // RAM content
reg [31:0] addr_reg;    // RAM address register

// RAM address register
always @(posedge clk)
  if (ce)
    addr_reg <= addr;

// Data output drivers
assign doq = (oe) ? mem[addr_reg] : 32’h0;

// RAM write
always @(posedge clk)
  if (ce && we)
    mem[addr] <= di;
```

Register file

- 3-port RAM, 2 read ports and 1 write port
- Implemented with 2 2-port RAMs (Xilinx block RAMs)



or1200 pipeline - schematic


ic_top
pc
Instruction Memory
Next PC
genpc
if
rf
Register File
id_pc
id_insn
operandmuxes
ex_pc
ex_insn
except
ctrl
alu
lsu
Data Memory
dc_top
wbmux
Register File
wb_pc
wb_insn