TSEA44: Computer hardware – a system on a chip

Lecture 3: The OR1200 Soft CPU

LIU LINKÖPING UNIVERSITY

Practical Issues

- Errata at lab webpage (when errors are found)
- Lab1 – Lab4 solved in groups of 3 student each
  - Not allowed to form group unless all students in the group have a pass on Lab 0

LIU LINKÖPING UNIVERSITY

Agenda

- OR1200
  - Architecture
  - Instruction set
  - C example
- Wishbone bus
  - Cycles
  - Arbitration
  - SV interface
 - Lab 1
- OR12
  - Pipelining etc.

LIU LINKÖPING UNIVERSITY

Practical Issues, cont.

- Once labgroups defined, a shared location will be available
  - /site/edu/da/labs/dafk/dafkXX
- To allow everyone in the group access, setup your umask when working in dafkXX folder
  - umask 7
  - NOTE: Only do this when working in the shared directory
    - Will make all newly created files in e.g. your home folder readable to everyone!

**LIU** LINKÖPING UNIVERSITY

Some soft CPUs

	Open RISC	Leon	Nios	Micro-Blaze
who	opencores	gaisler	altera	Xilinx
what	verilog	VHDL	netlist	netlist
CPU stages	RISC 5	RISC 5	RISC 6/5/1	RISC 3
cache	Direct IC/DC	IC/DC	IC/DC	IC/DC
MMU	Split IMMU DMMU			
bus	Wishbone simple/Xbar	AMBA (AHP/APB)	Avalon	LMB/OPB/FSL

LINKÖPING UNIVERSITY

Traditional RISC pipeline

LINKÖPING UNIVERSITY
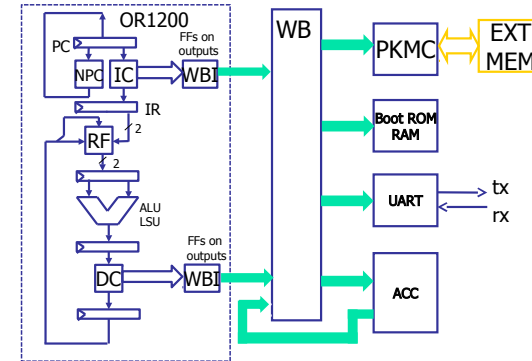
OpenRISC 1200 RISC Core

- 5 stage pipeline
- Single-cycle execution on most instructions

- 25 MIPS performance @ 25MHz
- Thirty-two 32-bit general-purpose registers
- Custom user instructions

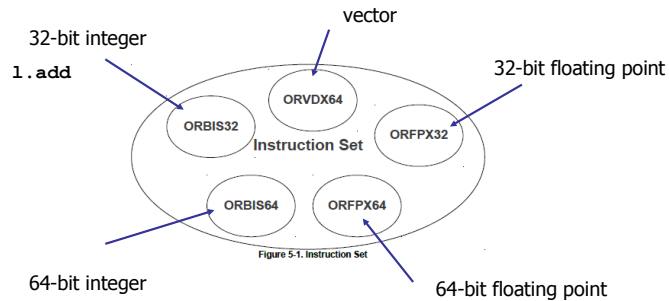LINKÖPING UNIVERSITY
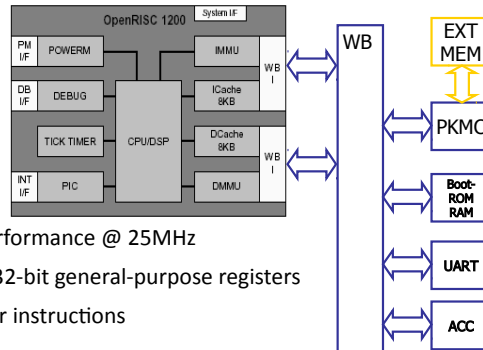
Instruction Set Architecture

- IC and DC compete for the WB
  - Reduce usage of data memory
    - Many register
    - All arithmetic instructions only access registers
    - Only load/store access to memory
  - Reduce usage of stack
    - Save return address in link register **r9**
    - Parameters to functions in registers

LINKÖPING UNIVERSITY

Instruction set

- Divided into classes:

32-bit integer

vector

32-bit floating point

`l.add`

ORBIS32

ORVDX64

ORFPX32

Instruction Set

ORBIS64

ORFPX64

Figure 5-1. Instruction Set

64-bit integer

64-bit floating point

Example 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

(1 additional HW NOP inserted if jump taken)
```

Instruction descriptions

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 bit	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)
```

Subroutine jump instruction

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**.
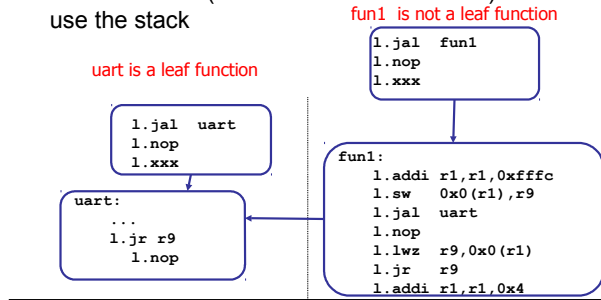
32-bit Implementation:
```
PC = exts(Immediate < < 2) + JumpInsnAddr = 4N + JIA
LR = DelayInsnAddr + 4                    = DIA+ 4
```

Example instruction sequence:

```
JIA:    l.jal N
DIA:    l.xxx
DIA+4:  l.yyy
```

Subroutine jump use

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

fun1 is not a leaf function

```
l.jal  fun1
l.nop
l.xxx
```

uart is a leaf function

```
l.jal  uart
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
```

```
uart:
    ...
    l.jr r9
    l.nop
```

LINKÖPING UNIVERSITY

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

LINKÖPING UNIVERSITY

A very simple C example

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

  return(a+b);
}


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

  nr = sum(a,b);

  return(nr);
}
```

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

```
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
```
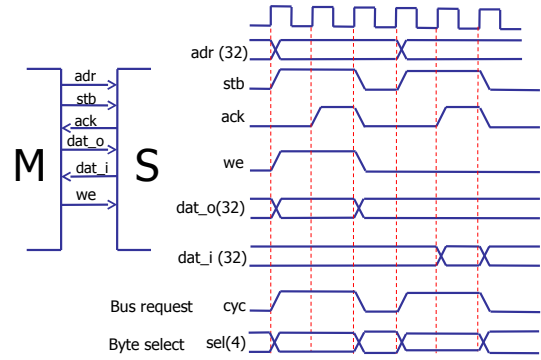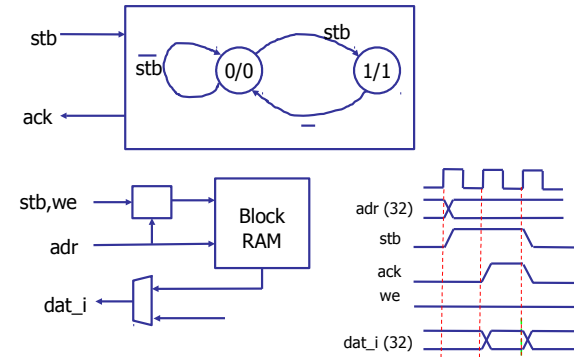
LINKÖPING UNIVERSITY

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 cyles
  - Variable core interconnection methods support point-to-point, shared bus, and crossbar switch
  - Arbitration method is defined by the end user (priority arbiter, **round-robin arbiter**, etc.)
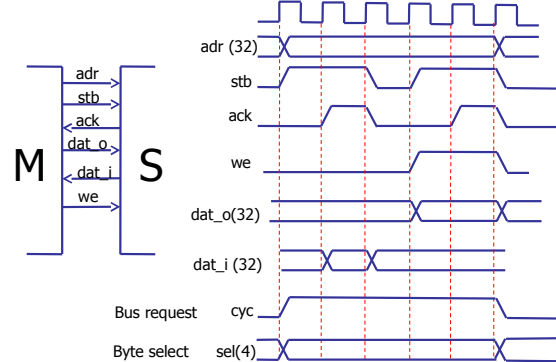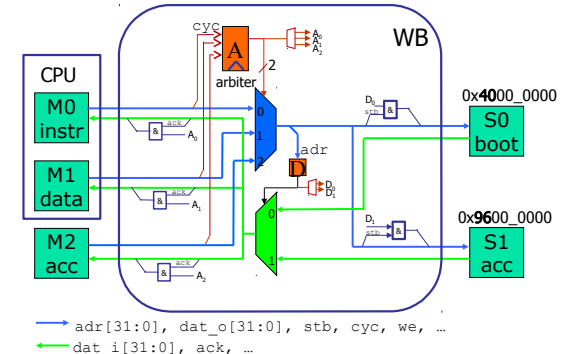
LINKÖPING UNIVERSITY

Simple Wishbone cycles

adr (32)
stb
ack
dat_o
dat_i
we

M   S

adr (32)
stb
ack
we
dat_o(32)
dat_i (32)
Bus request   cyc
Byte select   sel(4)

Read-modify-write cycle

adr
stb
ack
dat_o
dat_i
we

M   S

adr (32)
stb
ack
we
dat_o(32)
dat_i (32)
Bus request   cyc
Byte select   sel(4)

Ack FSM in each slave

stb

$\overline{stb}$ stb

$\overline{stb}$ 0/0   1/1

ack

stb,we
adr
dat_i

Block
RAM

adr (32)
stb
ack
we
dat_i (32)

Wishbone bus (3M, 2S)

cyc

WB

A
arbiter

CPU

M0
instr

M1
data

M2
acc

S0
boot   0x4000_0000

S1
acc   0x9600_0000

adr

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

Round robin arbiter

state	priority
0	0,1,2
1	1,2,0
2	2,0,1

A master cannot be forced
off the bus. Release the bus
by deasserting cyc!

cyc0,cyc1,cyc2

000,1--   01-   000,-1-

0   100   1

001   010   -01

1-0

000,--1

2

Wishbone cycles with arbitration

| state | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 |

cyc0
ack0

cyc1
ack1

cyc2
ack2

Comparison to Xilinix Zynq (in Zedboard)

MEM

Single chip
ARM processor plus
FPGA design area

Fixed computer

Crossbar

AXI bus

Build here

Processing System

Flash Controller
NOR, NAND, SRAM, Quad SPI

Multiport DRAM Controller
DDR3, DDR3L, DDR2

2x SPI

2x I2C

2x CAN

2x UART

GPIO

2x SDIO
with DMA

2x USB
with DMA

2x GigE
with DMA

AMBA Interconnect   AMBA Interconnect

ARM CoreSight Multi-Core Debug and Trace

NEON DSP/FPU Engine   NEON DSP/FPU Engine

Cortex-A9 MPCore
32/32 KB I/D Caches   Cortex-A9 MPCore
32/32 KB I/D Caches

512 Kbyte L2 Cache

General Interrupt
Controller   Watchdog
Timer   Snoop
Control
Unit   256-Kbyte
On-Chip
Memory

Configuration Timers   DMA

AMBA Interconnect   Security
AES, SHA, RSA   AMBA Interconnect

EMIO   General Purpose
AXI Ports   ACP   High Performance
AXI Ports

XADC
2x ADC, Mux,
Thermal Sensor   Programmable Logic
(System Gates, DSP, RAM)   PCIe Gen2
1-8 Lanes

Multi-Standard I/Os (3.3V & High-Speed 1.8V)   Multi-Gigabit Transceivers

Crossbar

Decoding
for each master

Arbitration
for each slave

D   M0   76   A   S0

D   M1   A

D   M2   S1

35

AXI (ARM standard)

- Address/control phases are separate from data phases
- Burst possible with only start adress issued
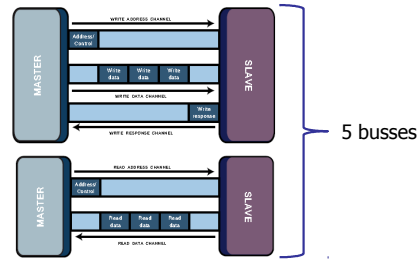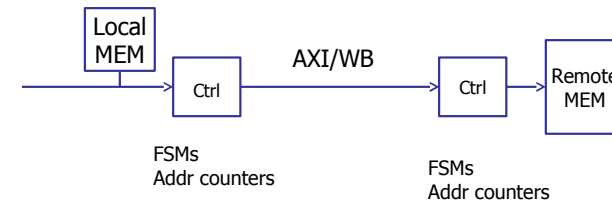- Read and write data channels are separate

5 busses
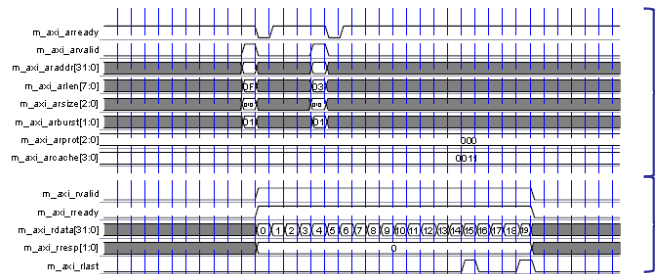
Figure 19.2: AXI4 read channel architecture

LINKÖPING UNIVERSITY

Burst mode comment:

- Require controller at both ends

FSMs
Addr counters

FSMs
Addr counters

LINKÖPING UNIVERSITY

Example: AXI read burst

- 2 busses: Read adress bus, read data bus

LINKÖPING UNIVERSITY

Lab 1

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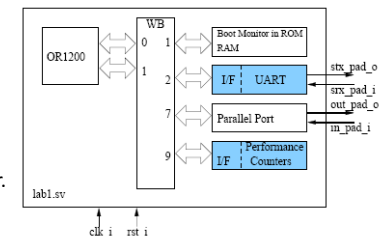
3. download and execute a benchmark program, that performs (the DCT part of) JPEG compression on a small image in your RAM module

4. simulate the computer running the benchmark program

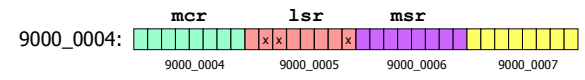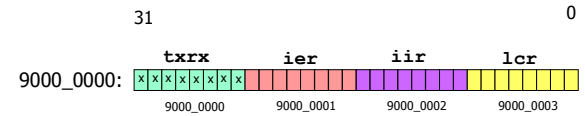5. design a module containing hardware performance counters

LINKÖPING UNIVERSITY

UART IP Core (16550)

- Compatible with industri standard 16550
- 16 character FIFOs for rx and tx
- Seven 8-bit registers on a 32-bit bus
- Full description in course material (UART specification)
- We do not use all registers in our board
  - Modem control lines
  - Baud rate dividor control

rx

tx

int

LINKÖPING UNIVERSITY
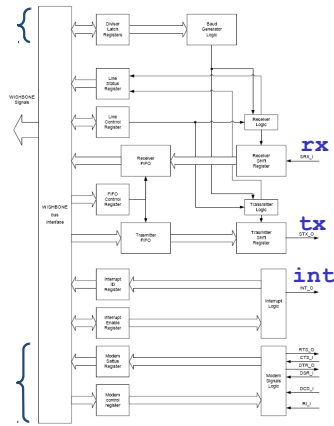
Or1200 is a 32-bit machien

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

31   0

txrx ier iir lcr

9000_0000:

9000_0000 　　9000_0001 　　9000_0002 　　9000_0003

mcr lsr msr

9000_0004:

9000_0004 　　9000_0005 　　9000_0006 　　9000_0007

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

LINKÖPING UNIVERSITY reg

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
```
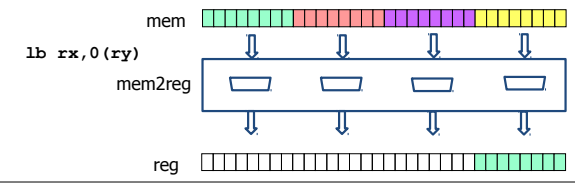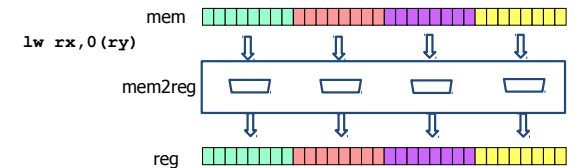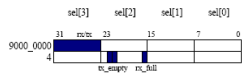
90000000:

90000005:

LINKÖPING UNIVERSITY
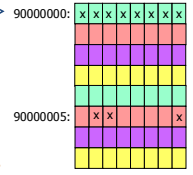
Bytes, halfwords can be read/written

mem

```
lw rx,0(ry)
```

mem2reg

reg

mem

```
lb rx,0(ry)
```

mem2reg

reg

LINKÖPING UNIVERSITY

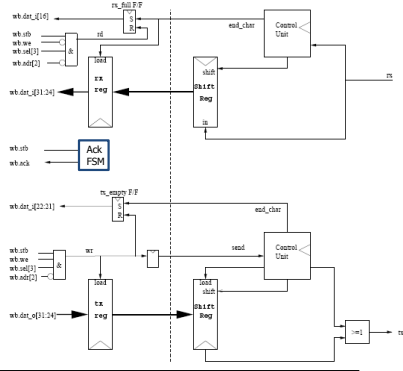# Programmers model   Block diagram

Existing driver wants this

Handshake F/Fs

rx_full   Set: end_char
           Reset: read txrx

tx_empty   Set: end_char
   Reset: write txrx

---

Interface definition


```
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
```
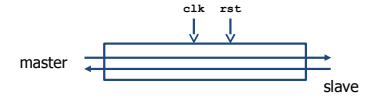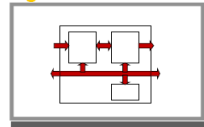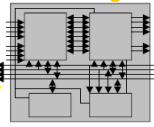

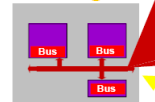# SystemVerilog – Interfaces
"a bundle of wires"

Design On A White Board

HDL Design

SystemVerilog Design

Interface Bus
Signal 1
Signal 2
Read()
Write()
Assert

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

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
```
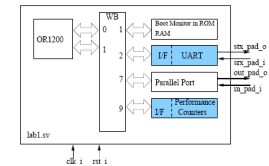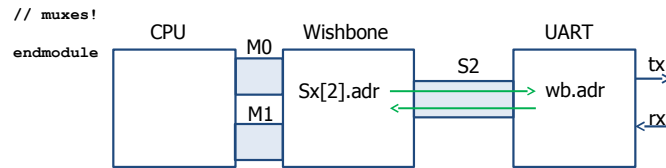
In the wishbone end (wb/wb_top.sv)
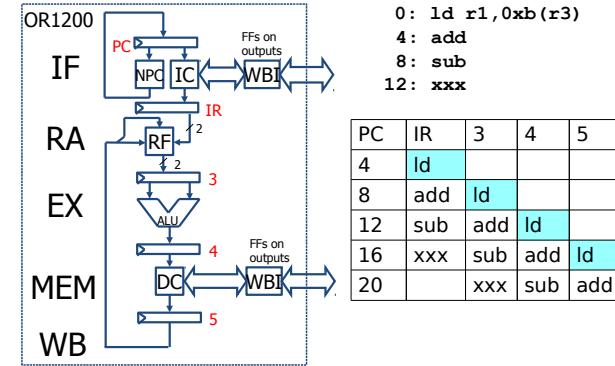
```
module wb_top(
  input clk_i, rst_i,

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

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

// muxes!

endmodule
```

CPU   M0   Wishbone   UART   tx

M1   Sx[2].adr   S2   wb.adr

rx

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 = wb.stb;
    assign wb.err = 1'b0;
    assign wb.rty = 1'b0;


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

Pipelining and diagram

OR1200

IF   PC   NPC   IC   FFs on outputs   WBI

RA   IR   RF

EX   ALU

MEM   DC   FFs on outputs   WBI

WB

```
 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
20		xxx	sub	add

Lab 1 cont.: Performance counters

- Two master ports from CPU
  - M0: instruction fetch
  - M1: data in/out
- Measure time spent on waiting for instructions/data to/from memory
  - Cyc and Stb active
- Measure number of instruction/data words fetch/stored in memory
  - Ack active
- Remember printouts will introduce additional instructions and data transfers
  - Store counter values in local variables before calculating difference and printing

Pipelining

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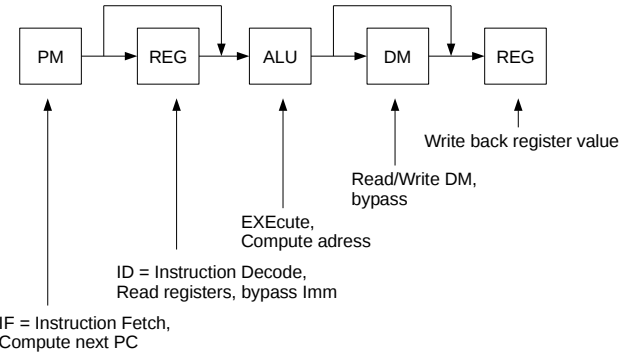- add
- write back r3 to RF

```
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
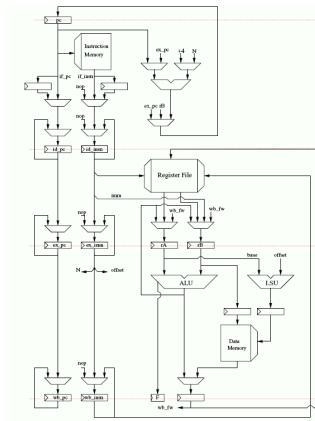
4-5 stages?

The standard pipeline

PM → REG → ALU → DM → REG

Write back register value

Read/Write DM, bypass

EXEcute, Compute adress

ID = Instruction Decode, Read registers, bypass Imm

IF = Instruction Fetch, Compute next PC

Classic RISC pipeline

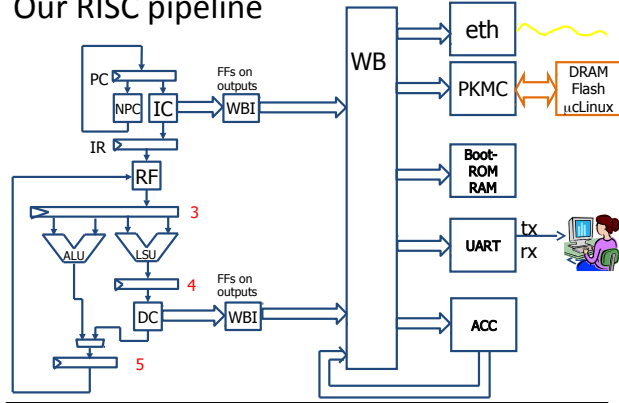|	PM	RF	ALU	DM
PC	IR	3	4	5
4	ld			
8	add	ld		
12	sub	add	ld	
16	xxx	sub	add	ld
20		xxx	sub	add

- Add,sub do nothing in the DM stage
- Instruction decode and read register simultaneously

Our RISC pipeline

- IF = Instruction fetch, compute next PC

- ID - Instruction Decode, read registers

- EX – instruction execute, access DM
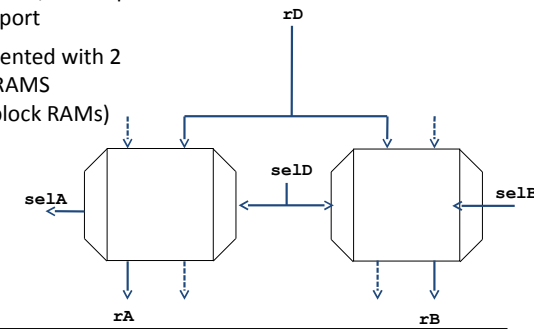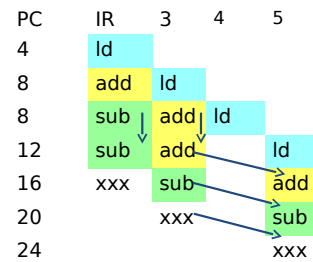
- WB – Write back register

Our RISC pipeline

PC
NPC IC
FFs on outputs
WBI
IR
RF
3
ALU LSU
4
DC WBI
FFs on outputs
5
WB
eth
PKMC → DRAM Flash µcLinux
Boot-ROM RAM
UART tx rx
ACC

LINKÖPING UNIVERSITY

Register file

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

rD

selA   selD   selB

rA   rB

LINKÖPING UNIVERSITY

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

LINKÖPING UNIVERSITY

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;
```
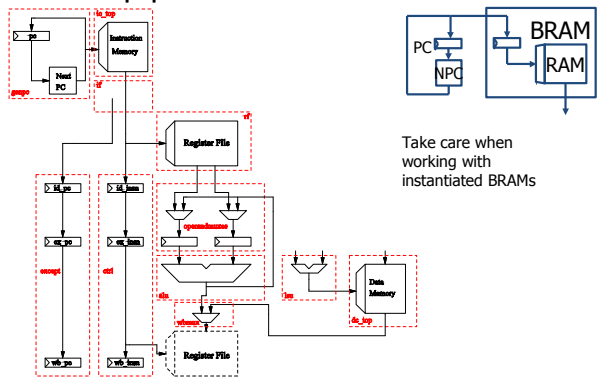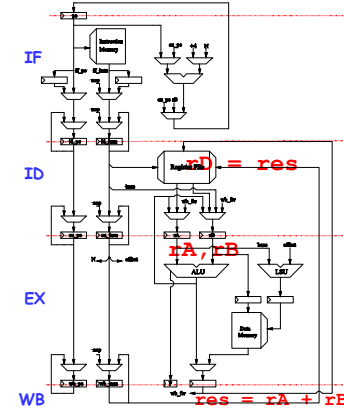
LINKÖPING UNIVERSITY

Or1200 pipeline - schematic

Instruction Memory

Next PC

Register File

operandmux

Data Memory

Register File

PC
NPC

BRAM
RAM

Take care when working with instantiated BRAMs

LINKÖPING UNIVERSITY

Lets study a few instructions

```
l.add rD,rA,rB
```

	opcode 0x38	D	A	B	reserved	opcode 0x0	reserved	opcode 0x0
	6 bits	5 bits	5 bits	5 bits	1 bit	2 bits	4 bits	4bits

```
l.addi rD, rA,K
```

opcode 0x27	D	A	I
6 bits	5 bits	5 bits	16bits

```
l.sfeq rA,rB
```

opcode 0x35	I	A	B	I
6 bits	5 bits	5 bits	5 bits	11bits

```
l.bf N
```

opcode 0x4	N
6 bits	26bits

```
l.lwz rD, I(rA)
```

opcode 0x21	D	A	I
6 bits	5 bits	5 bits	16bits

```
l.sw I(rA),rB
```

opcode 0x720	A	B	reserved
11 bits	5 bits	5 bits	11bits

LINKÖPING UNIVERSITY

l.add rD, rA, rB

- rD stored in middle pipeline
- LSU not used
- 4 pipeline steps

IF

ID

EX

WB

Instruction Memory

Register File

rD = res

rA, rB

ALU

LSU

Data Memory

res = rA + rB

LINKÖPING UNIVERSITY

lw rB, I(rA)

- Include extra pipeline stage in EX phase

IF

ID

EX

WB

Instruction Memory

Register File   rB = dat

rB   rA   I

ALU

LSU

adr=I + rA

Data Memory

rB   dat = DM[adr]

LINKÖPING UNIVERSITY
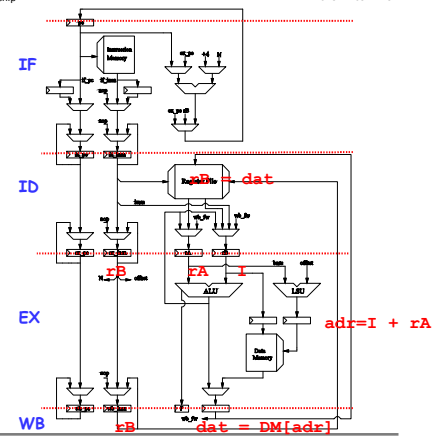
Conditional branch

```
0:sfeq
4:bf N
8:nop
C:xxx
...
20:yyy
```

- 1 delay slot
- 1 extra HW nop on taken branch

IF

ID

EX

WB

www.liu.se