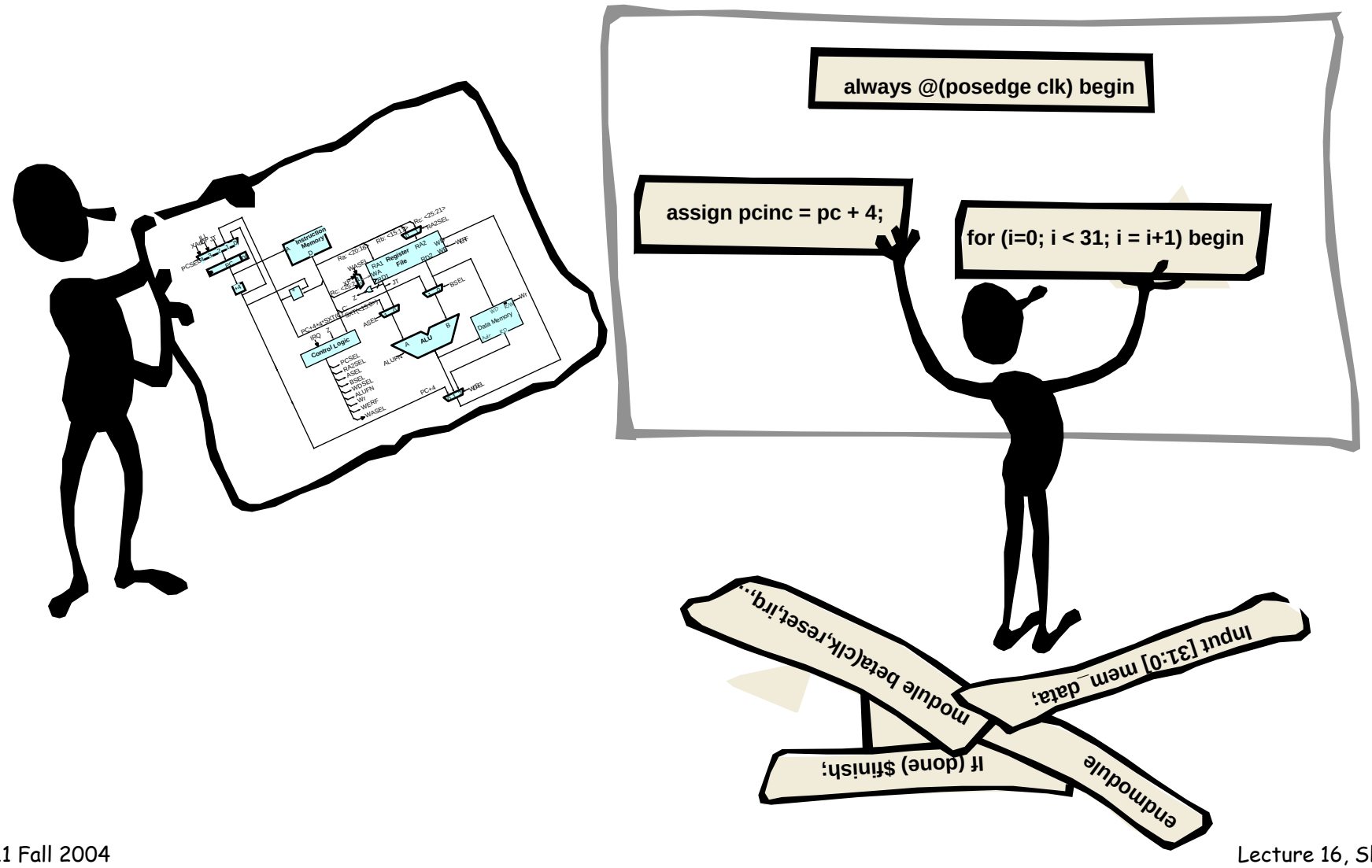
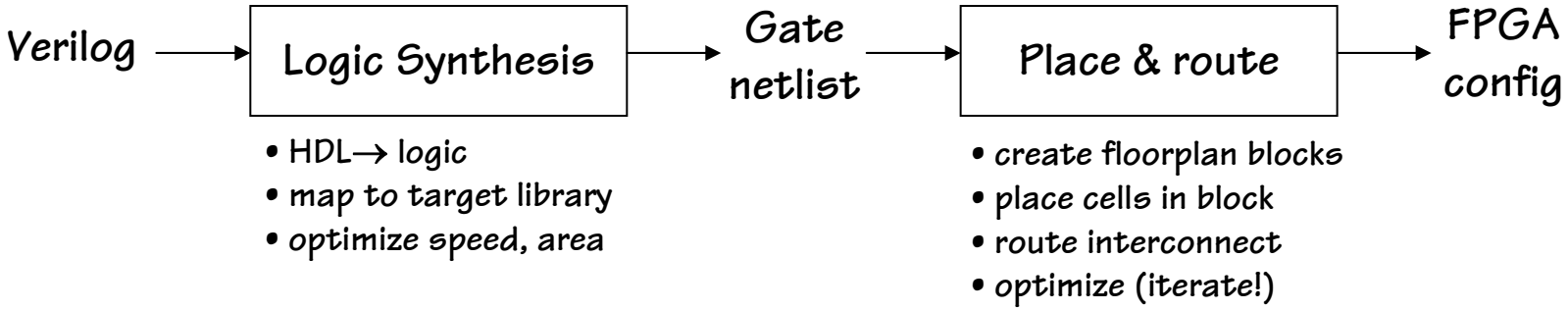
Architecture → Verilog
An Extended Example



Reprise: Why use an HDL?

- Want an executable functional specification
  - Document exact behavior of all the modules and their interfaces
  - Executable models can be tested & refined until they do what you want
- HDL description is first step in a mostly automated process to build an implementation directly from the behavioral model



Helping the Tools*

In an ideal world it shouldn't matter how you write the Verilog – optimization in the CAD tools will find the best solution. But the world is not ideal (yet…)

- Tools work best on smaller problems
  - Need to partition real problem into pieces for you and the tools (it's hard to think about 1M gates at one time). Decompose large problems into smaller problems and then connect the solutions
  - Hierarchy in Verilog ↔ partitions in physical layout

- Tools use your code as a starting point.
  - Your structure isn't completed eliminated (this is good…)
  - Little optimization will be done between top-level blocks

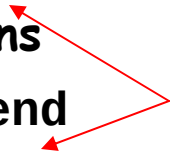
- Structure of the problem is often important
  - Finding a "good" way to think about the problem is key

Like optimizing compilers for C, tools are good for local optimizations but don't expect them to rewrite your code and change your algorithm. With practice you'll learn what works and what doesn't…

*adapted from a Stanford EE271 lecture by Mark Horowitz

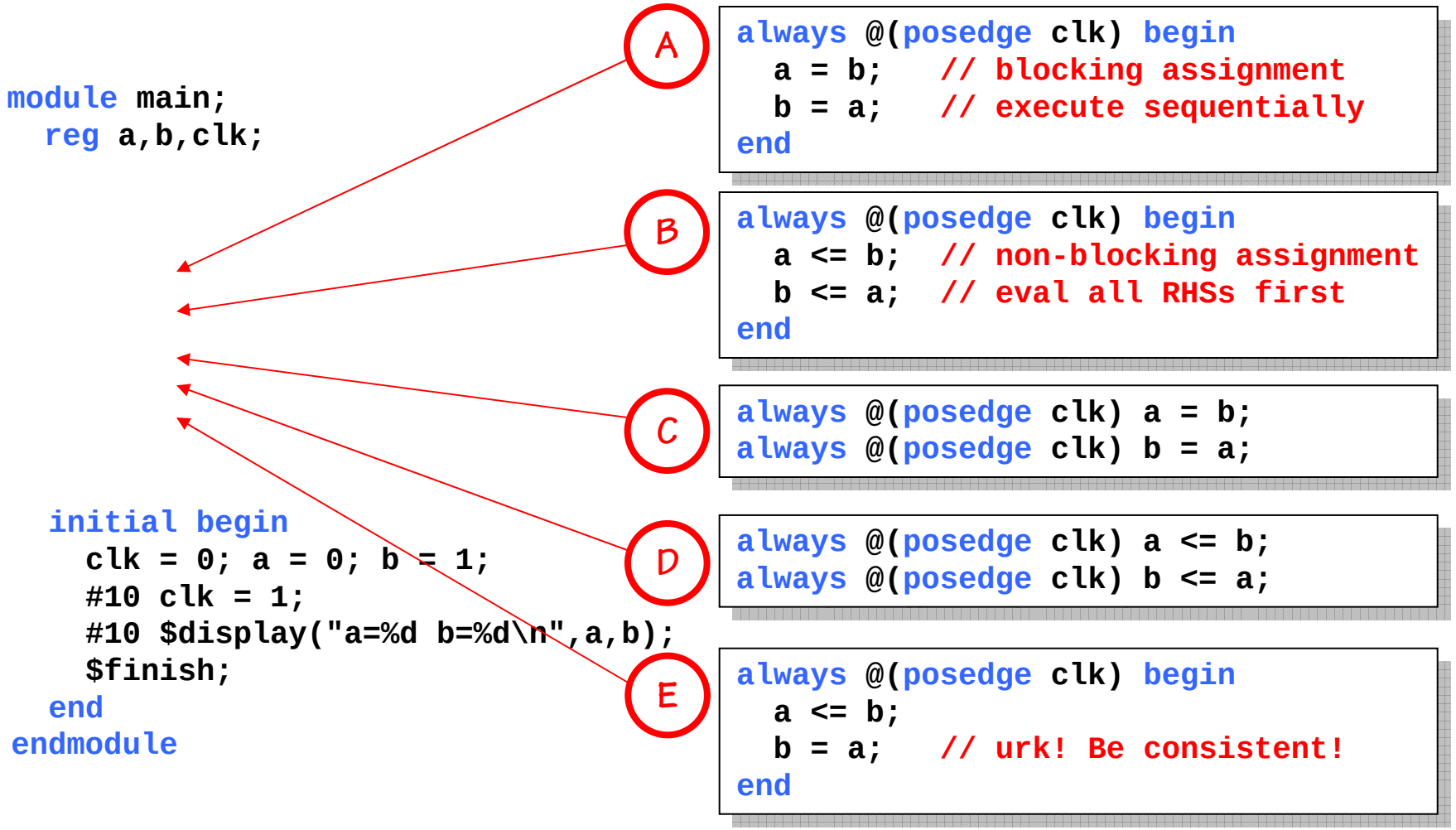
Choosing the right style…

- Structural Verilog
  - Use for hierarchy (instantiating other modules)
  - Floorplanning tools often require that modules which include structural verilog not include other styles. In other words the leafs of the hierarchy are dataflow/behavioral modules, all other modules are pure structural verilog.
- Dataflow Verilog: `assign` *`target = expression`*
  - Use for (most) combinational logic
  - Avoids problems with activation list omissions
- Behavioral Verilog: `always @(…) begin … end`
  - Use to model state elements (e.g., registers)
  - Sometimes useful for combinational logic expressed using `for` or `case` statements
  - Simulates much faster than dataflow statements since no waveforms are produced for signals internal to behavioral block. Here's where you can make the tradeoff between simulation speed and debugability.



These two styles are often mixed in a single module

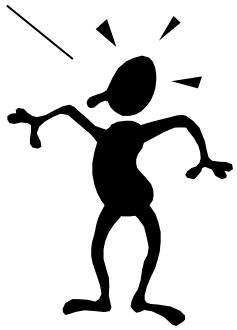
= vs. <= inside begin … end

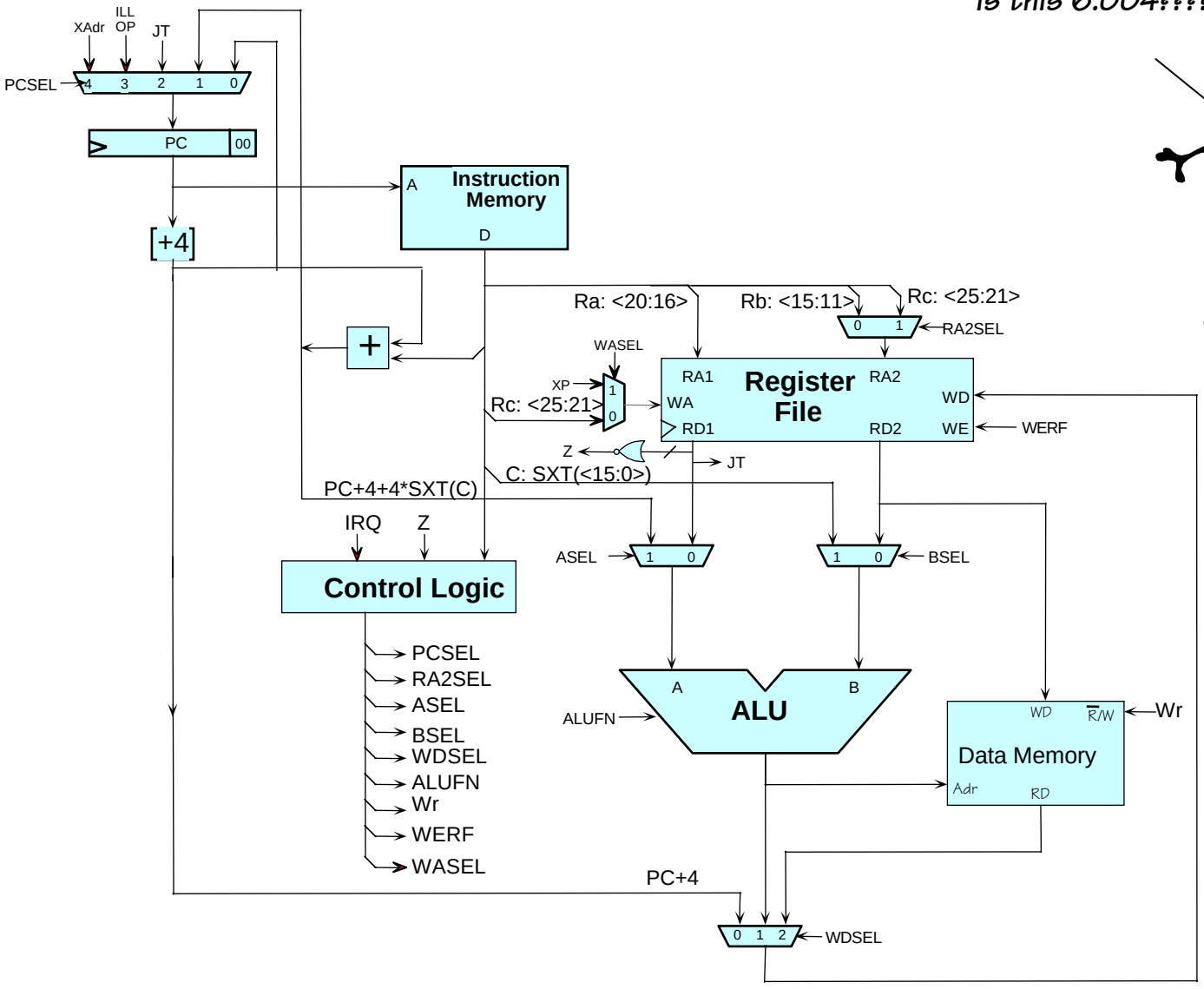


Rule: always change state using <= (e.g., inside `always @(posedge clk)…`)

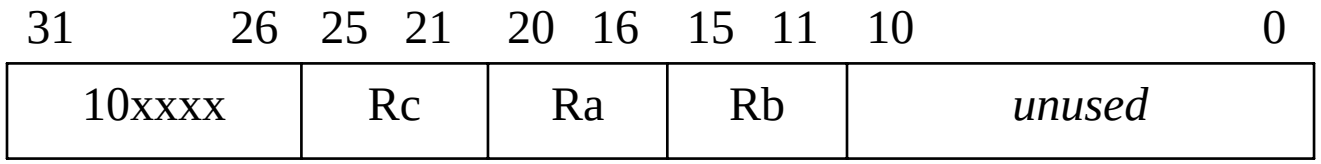
Example: the Beta!

Is this 6.004????



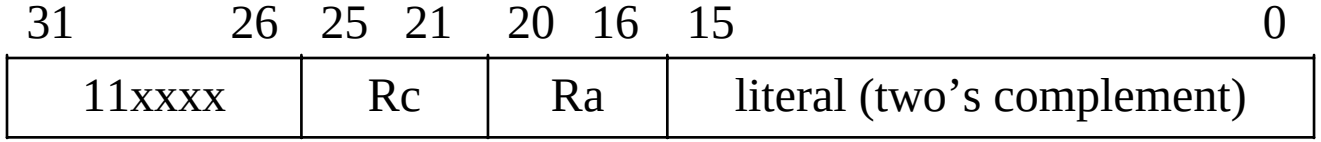


Beta Instructions - I



OP(Ra,Rb,Rc): Reg[Rc] ← Reg[Ra] op Reg[Rb]

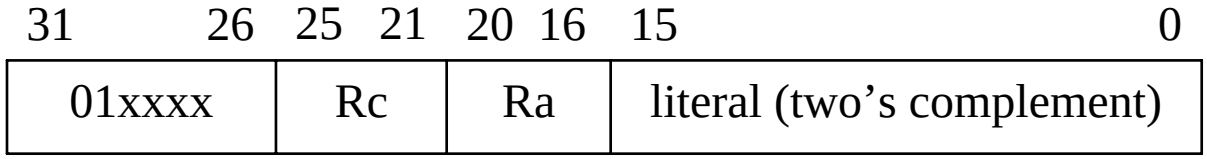
Opcodes:
ADD (plus), SUB (minus), **MUL (multiply)**, **DIV (**divided by),
AND (bitwise and), **OR (**bitwise or), **XOR (**bitwise exclusive or)
CMPEQ (equal), CMPLT (less than), **CMPLE (**less than or equal) [result = 1 if true, 0 if false]
SHL (left shift), **SHR (**right shift w/o sign extension), **SRA (**right shift w/ sign extension)



OPC(Ra,literal,Rc): Reg[Rc] ← Reg[Ra] op SEXT(literal)

Opcodes:
ADDC (plus), SUBC (minus), **MULC (multiply)**, **DIVC (**divided by)
ANDC (bitwise and), **ORC (**bitwise or), **XORC (**bitwise exclusive or)
CMPEQC (equal), CMPLTC (less than), **CMPLEC (**less than or equal) [result = 1 if true, 0 if false]
SHLC (left shift), **SHRC (**right shift w/o sign extension), **SRAC (**right shift w/ sign extension)

Beta Instructions - II



LD(Ra,literal,Rc): Reg[Rc] ← Mem[Reg[Ra] + SEXT(literal)]

ST(Rc,literal,Ra): Mem[Reg[Ra] + SEXT(literal)] ← Reg[Rc]

JMP(Ra,Rc): Reg[Rc] ← PC + 4; PC ← Reg[Ra]

BEQ/BF(Ra,label,Rc): Reg[Rc] ← PC + 4;
if Reg[Ra] = 0 then PC ← PC + 4 + 4*SEXT(literal)

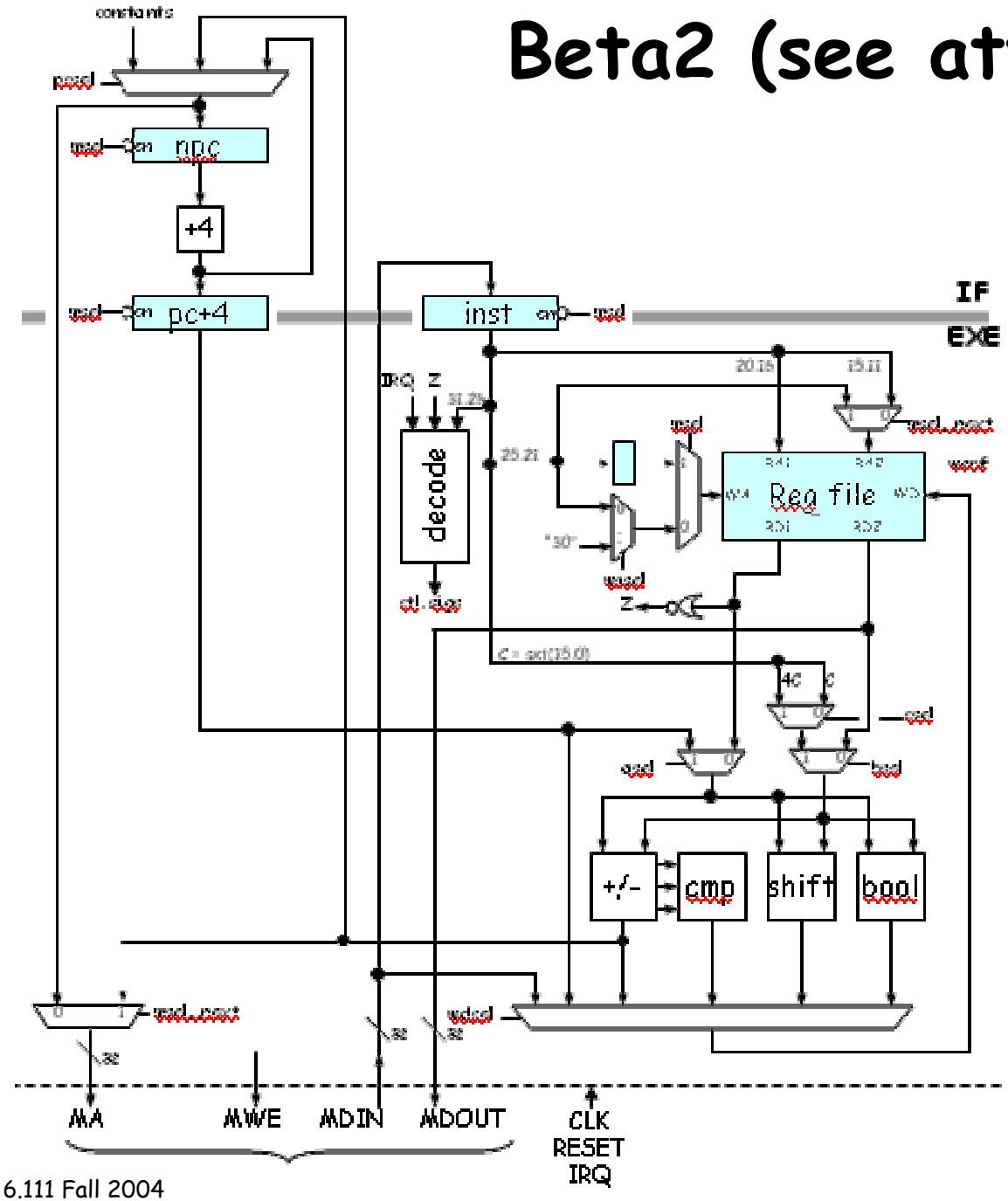
BNE/BT(Ra,label,Rc): Reg[Rc] ← PC + 4;
if Reg[Ra] ≠ 0 then PC ← PC + 4 + 4*SEXT(literal)

LDR(label,Rc): Reg[Rc] ← Mem[PC + 4 + 4*SEXT(literal)]

Beta Control Logic

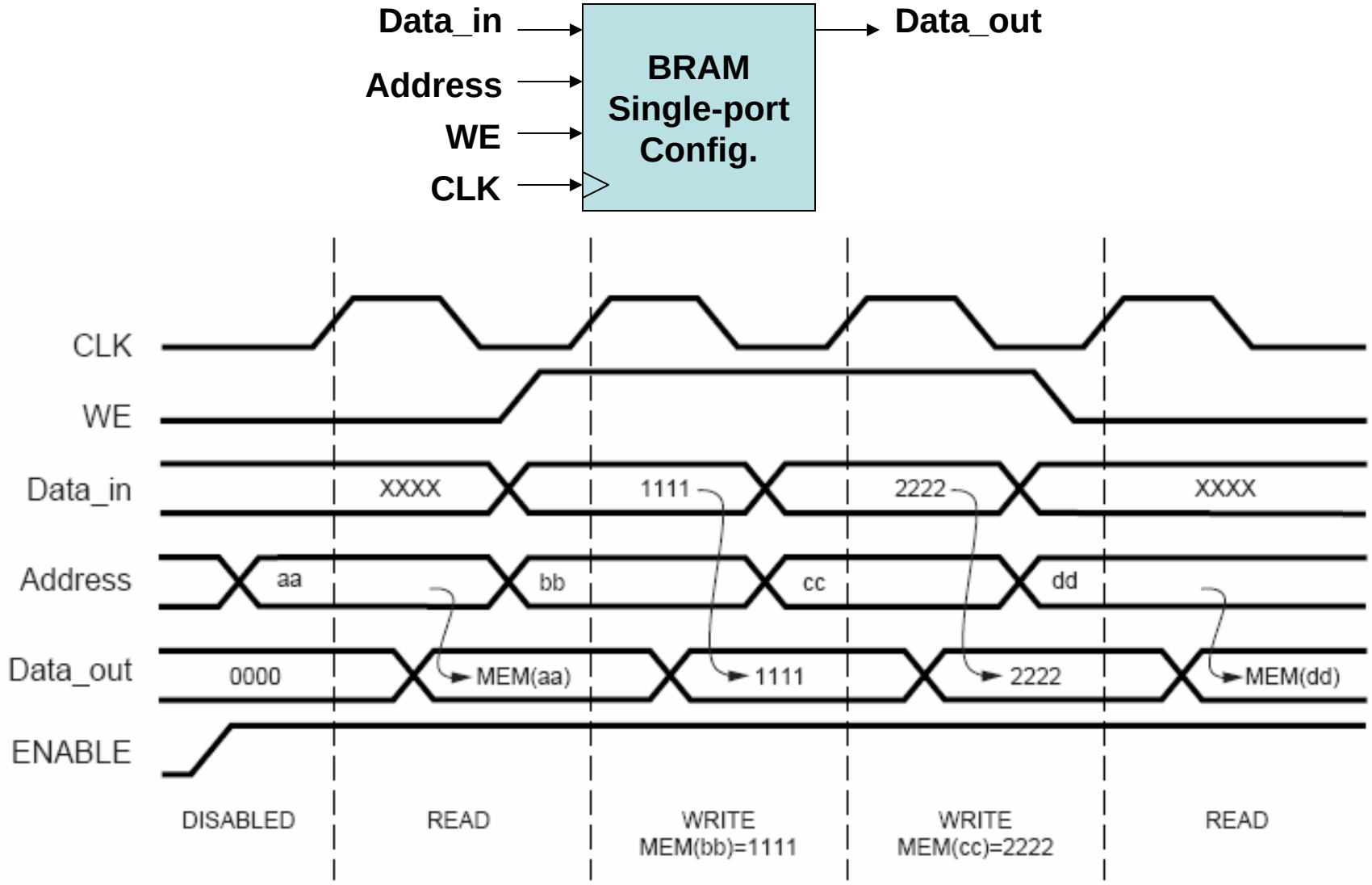
	OP	OPC	LD	ST	JMP	BEQ	BNE	LDR	Illop	trap
ALUFN	F(op)	F(op)	"+"	"+"	—	—	—	"A"	—	—
WERF	1	1	1	0	1	1	1	1	1	1
BSEL	0	1	1	1	—	—	—	—	—	—
WDSEL	1	1	2	—	0	0	0	2	0	0
WR	0	0	0	1	0	0	0	0	0	0
RA2SEL	0	—	—	1	—	—	—	—	—	—
PCSEL	0	0	0	0	2	Z ? 1 : 0	Z ? 0 : 1	0	3	4
ASEL	0	0	0	0	—	—	—	1	—	—
WASEL	0	0	0	—	0	0	0	0	1	1

Beta2 (see attached sheet)



- 2-stage pipeline, 1 annuled branch delay slot
- Memory ops (LD, LDR, ST) take two cycle in EXE stage: addr computed in 1st cycle, memory access made in 2nd
- Branch and LDR address arithmetic performed in ALU
- JMP routed thru ALU
- Single memory port shared by inst. fetch and memory access

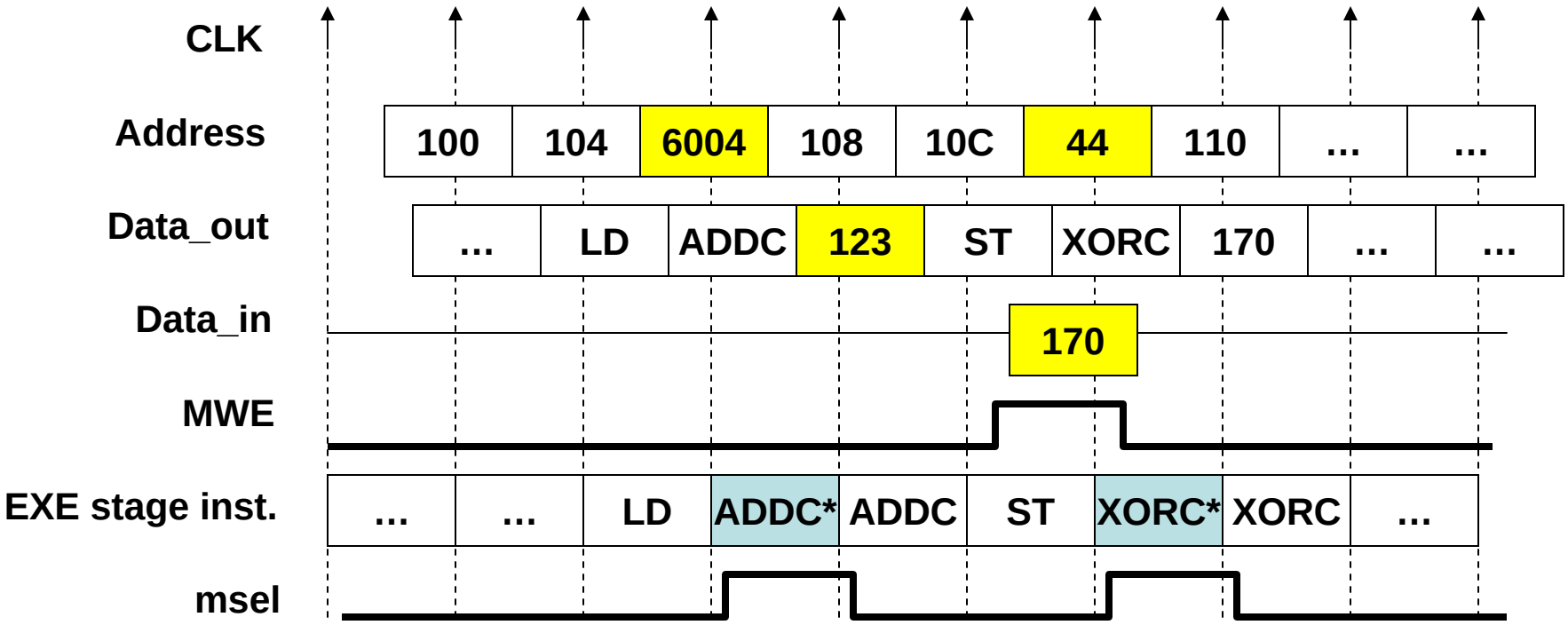
Xilinx Synchronous Block Memory



Source: Xilinx App Note 463

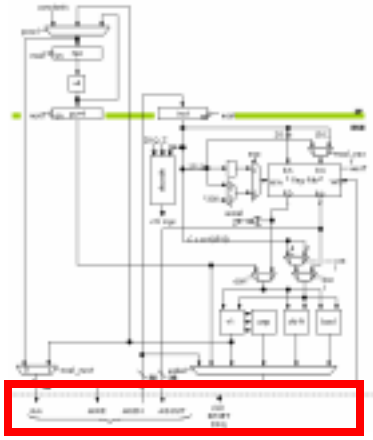
```
100: LD(R31,6004,R2)
104: ADDC(R2,47,R2)
108: ST(R2,44,R31)
10C: XORC(R2,-1,R2)
110: …
…
6004: 123
```

Instruction Pipeline Diagram



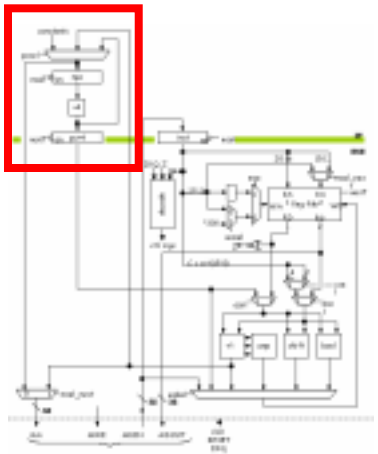
*** Stalled in pipeline**

beta2.v



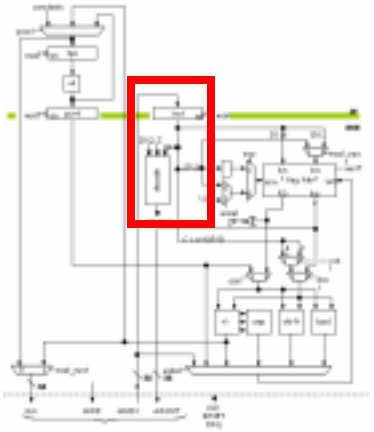
```
module beta2(clk,reset,irq,ma,mdin,mdout,mwe);
  input clk,reset,irq;
  output [31:0] ma,mdout;
  input [31:0] mdin;
  output mwe;

  // beta2 registers
  reg [31:0] regfile[31:0];
  reg [31:0] npc,pc_inc,inst;
  reg [4:0] rc_save;  // needed for second cycle on LD,LDR

  // internal buses
  wire [31:0] rd1,rd2,wd,a,b,xb,c,addsub,cmp,shift,boole;

  // control signals
  wire wasel,werf,z,asel,bsel,csel;
  wire addsub_op,cmp_lt,cmp_eq,shift_op
  wire shift_sxt,boole_and,boole_or;
  …
endmodule
```

PC Logic



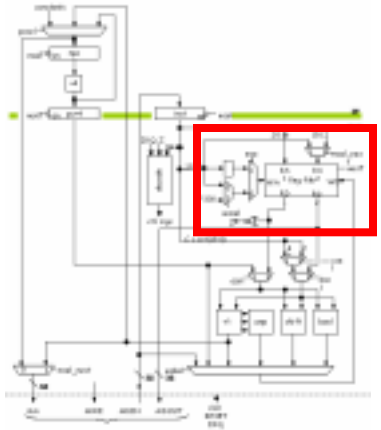
```
// pc
wire [31:0] npc_inc,npc_next;

assign npc_inc = npc + 4;
assign npc_next = reset ? 32'h80000000 :
                  msel ? npc :
                  branch ? {npc[31]&addsub[31],
                            addsub[30:2],2'b00} :
                  trap ? 32'h80000004 :
                  interrupt ? 32'h80000008 :
                  npc_inc;

always @ (posedge clk) begin
  npc <= npc_next;   // stall on msel handled above
  if (!msel) pc_inc <= npc_inc;
end
```

Instruction Register & Decode



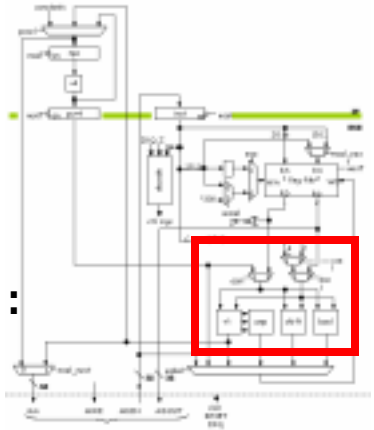
```
// instruction reg
always @ (posedge clk) if (!msel) inst <= mdin;

// control logic
decode ctl(.clk(clk),.reset(reset),.irq(irq & !npc[31]),
           .z(z),.opcode(inst[31:26]),
           .asel(asel),.bsel(bsel),.csel(csel),
           .wasel(wasel),.werf(werf),.msel(msel),
           .msel_next(msel_next),.mwe(mwe),
           .addsub_op(addsub_op),.cmp_lt(cmp_lt),
           .cmp_eq(cmp_eq),
           .shift_op(shift_op),.shift_sxt(shift_sxt),
           .boole_and(boole_and),.boole_or(boole_or),
           .wd_addsub(wd_addsub),.wd_cmp(wd_cmp),
           .wd_shift(wd_shift),.wd_boole(wd_boole),
           .branch(branch),.trap(trap),
           .interrupt(interrupt));
```

Register File



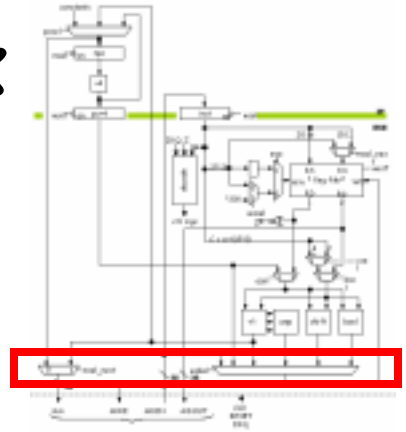
```
// register file
wire [4:0] ra1,ra2,wa;
always @ (posedge clk)
  if (!msel) rc_save <= inst[25:21];

assign ra1 = inst[20:16];
assign ra2 = msel_next ? inst[25:21] : inst[15:11];
assign wa = msel ? rc_save :
            wasel ? 5'd30 : inst[25:21];
assign rd1 = (ra1 == 31) ? 0 : regfile[ra1];
assign rd2 = (ra2 == 31) ? 0 : regfile[ra2];
always @ (posedge clk)
  if (werf) regfile[wa] <= wd;

assign z = ~| rd1;   // used in BEQ/BNE instructions
```

ALU



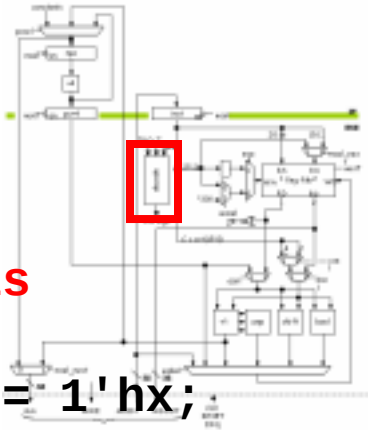
```
// alu
assign a = asel ? pc_inc : rd1;
assign b = bsel ? c : rd2;
assign c = csel ? {{14{inst[15]}},inst[15:0],2'b00} :
                  {{16{inst[15]}},inst[15:0]};

wire addsub_n,addsub_v,addsub_z;
assign xb = {32{addsub_op}} ^ b;
assign addsub = a + xb + addsub_op;
assign addsub_n = addsub[31];
assign addsub_v = (addsub[31] & ~a[31] & ~xb[31]) |
                  (~addsub[31] & a[31] & xb[31]);
assign addsub_z = ~| addsub;

assign cmp[31:1] = 0;
assign cmp[0] = (cmp_lt & (addsub_n ^ addsub_v)) |
                (cmp_eq & addsub_z);

wire [31:0] shift_right;
shift_right sr(shift_sxt,a,b[4:0],shift_right);
assign shift = shift_op ? shift_right : a << b[4:0];

assign boole = boole_and ? (a & b) :
               boole_or ? (a | b) : a ^ b;
```

Result Mux, Address Mux



```
// result mux, listed in order of speed (slowest first)
assign wd = msel ? mdin :
            wd_cmp ? cmp :
            wd_addsub ? addsub :
            wd_shift ? shift :
            wd_boole ? boole :
            pc_inc;

// assume synchronous external memory
assign ma = msel_next ? addsub : npc_next;
assign mdout = rd2;
```

Control Logic (decode.v)



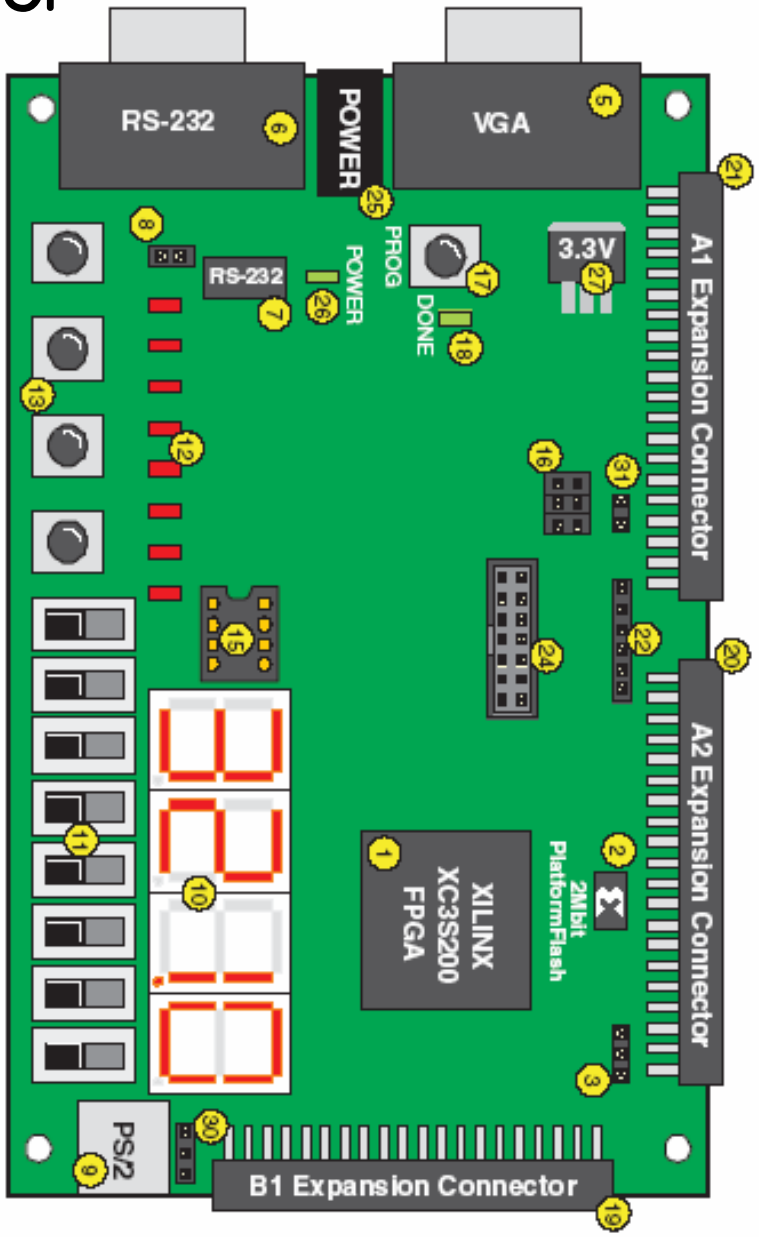
```
always @ (opcode or z or annul or irq or reset)
 begin
   // initial assignments for all control signals
   asel = 1'hx; bsel = 1'hx; csel = 1'hx;
   addsub_op = 1'hx; shift_op = 1'hx; shift_sxt = 1'hx;
   cmp_lt = 1'hx; cmp_eq = 1'hx;
   boole_and = 1'hx; boole_or = 1'hx;
   wasel = 0; mem_next = 0;
   wd_addsub = 0; wd_cmp = 0; wd_shift = 0; wd_boole = 0;
   branch = 0; trap = 0; interrupt = 0;

   if (irq && !reset && !annul) begin
     interrupt = 1;
     wasel = 1;
   end else casez (opcode)
     6'b011000: begin   // LD
                  asel = 0; bsel = 1; csel = 0;
                  addsub_op = 0;
                  mem_next = 1;
                end

     …
```

Control Logic (cont'd.)

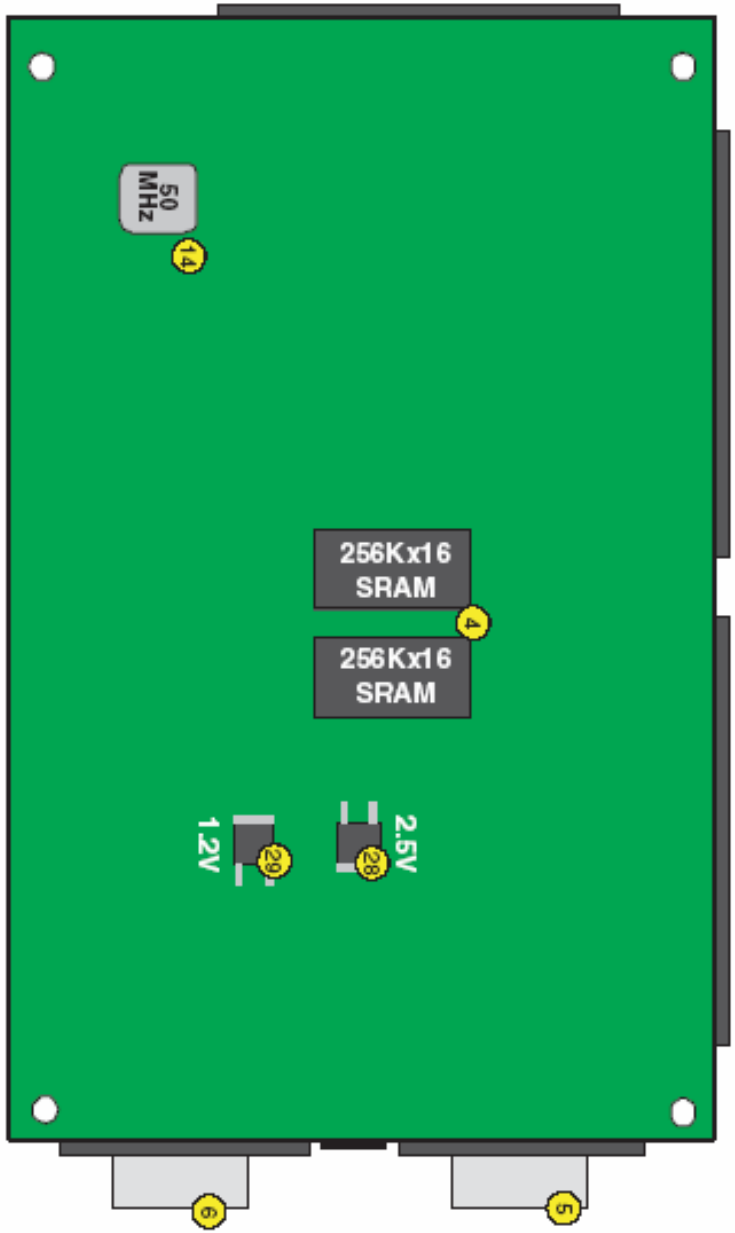
```
…
    6'b1?1100: begin   // SHL, SHLC
                 asel = 0; bsel = opcode[4]; csel = 0;
                 shift_op = 0;
                 wd_shift = 1;
               end
    6'b1?1101: begin   // SHR, SHRC
                 asel = 0; bsel = opcode[4]; csel = 0;
                 shift_op = 1; shift_sxt = 0;
                 wd_shift = 1;
               end
    6'b1?1110: begin   // SRA, SRAC
                 asel = 0; bsel = opcode[4]; csel = 0;
                 shift_op = 1; shift_sxt = 1;
                 wd_shift = 1;
               end
    default:   begin   // illegal opcode
                 trap = !annul; wasel = 1;
               end
  endcase
end  // always @ (opcode or …)
```

Xilinx Spartan-3 Starter Board

TOP

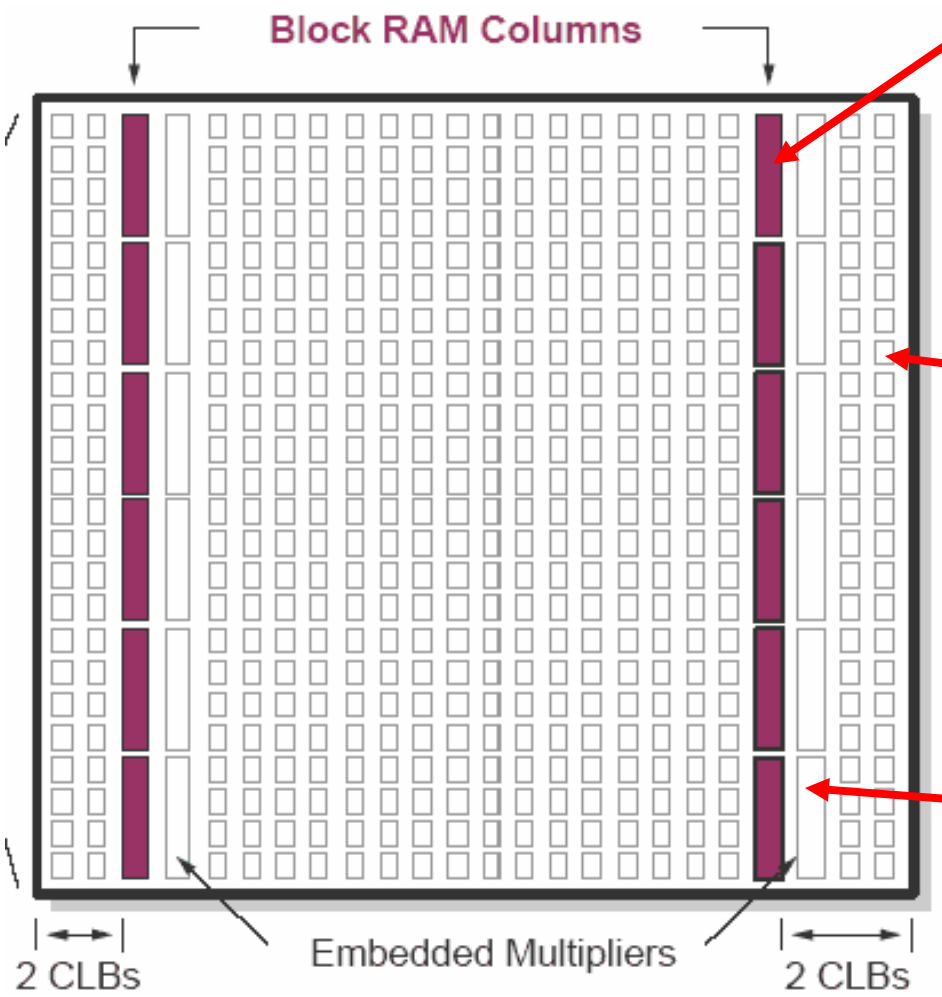


BOTTOM



Source: Xilinx Spartan-3 Starter Kit Board User Guide

Xilinx XC3S200 FPGA



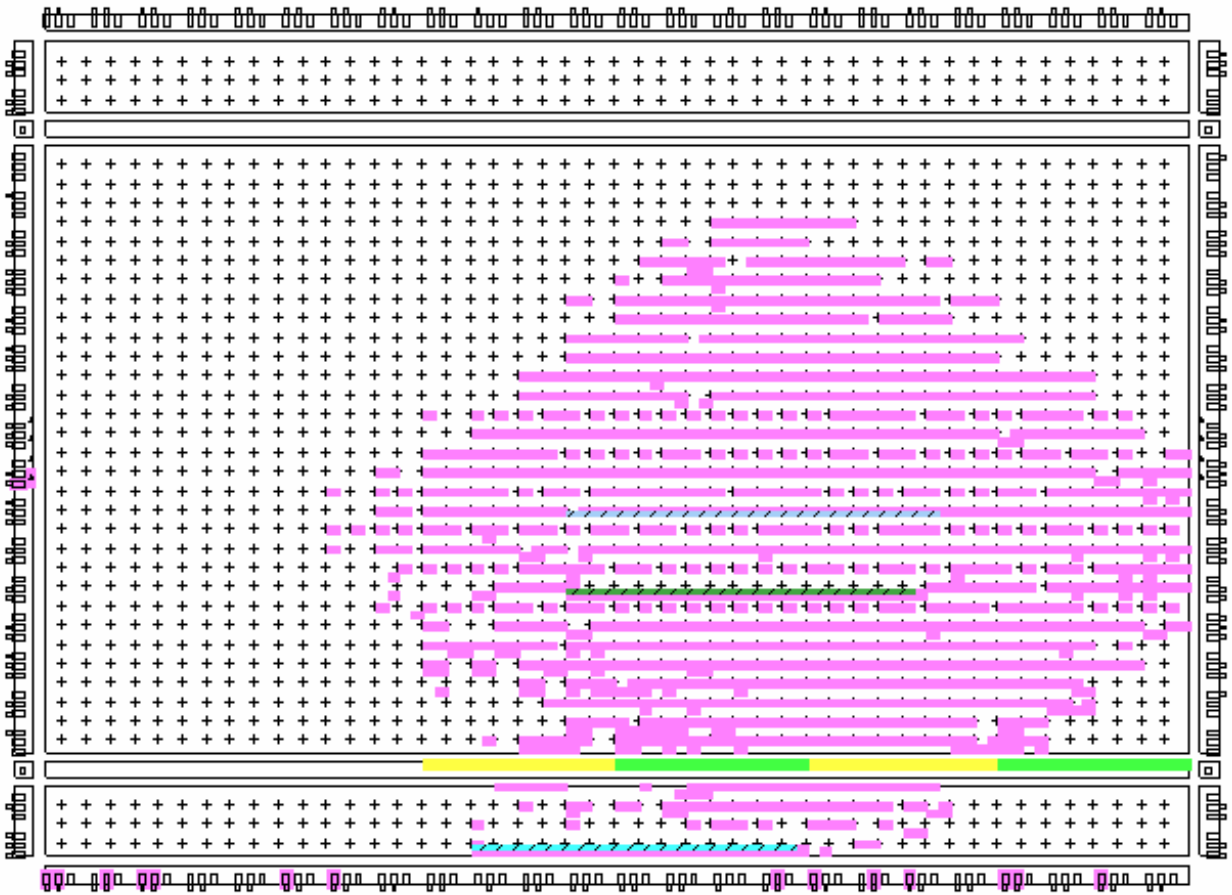
Memories:
ROM/RAM, Single/dual port
16Kx1 … 512x32
Quantity: 12

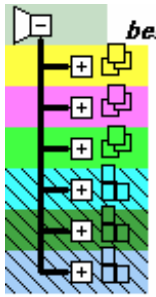
CLBs:
24 rows x 20 cols = 480
= 4320 logic cells
= 200K gates
Beta2 takes ~30%

18x18 Multipliers:
Sync/async
Quantity: 12

Source: Xilinx App Note 463

Beta2 Floorplan





beta2demo "beta2demo" [15 IOBs, 4 BLKRAMs, 128 DPRAMs, 1051 FGs, 60 FG5s, 90 CYs, 174 DFFs, 1 BUFG]

imem "rom_1k_32" [2 BLKRAMs]

beta2demo "Primitives" [15 IOBs, 128 DPRAMs, 1051 FGs, 60 FG5s, 3 CYs, 174 DFFs, 1 BUFG]

dmem "ram_1k_32" [2 BLKRAMs]

CarryChain_5 "CarryChain" [27 CYs] ← **Clock divider for 7-seg display**

CarryChain_3 "CarryChain" [29 CYs] ← **npc + 4**

CarryChain_1 "CarryChain" [31 CYs] ← **alu adder**

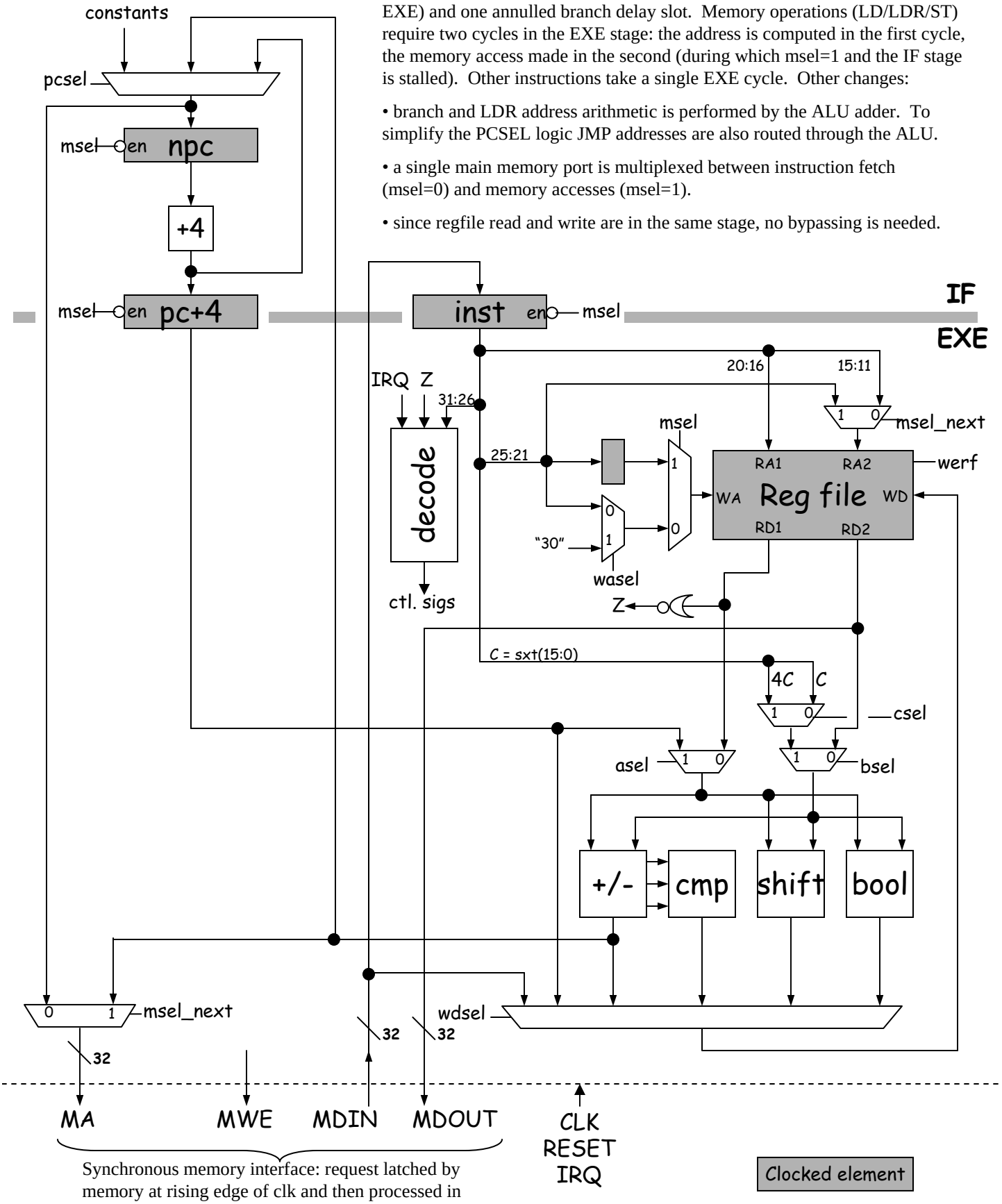
BETA2

An implementation of the 6.004 Beta processor with a 2-stage pipeline (IF, EXE) and one annulled branch delay slot. Memory operations (LD/LDR/ST) require two cycles in the EXE stage: the address is computed in the first cycle, the memory access made in the second (during which msel=1 and the IF stage is stalled). Other instructions take a single EXE cycle. Other changes:

• branch and LDR address arithmetic is performed by the ALU adder. To simplify the PCSEL logic JMP addresses are also routed through the ALU.

• a single main memory port is multiplexed between instruction fetch (msel=0) and memory accesses (msel=1).

• since regfile read and write are in the same stage, no bypassing is needed.

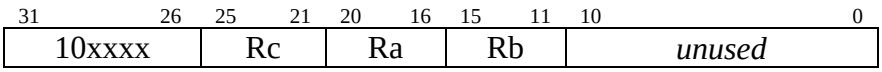


Synchronous memory interface: request latched by memory at rising edge of clk and then processed in following cycle. (cf. Xilinx Block RAMs)

Summary of β Instruction Formats

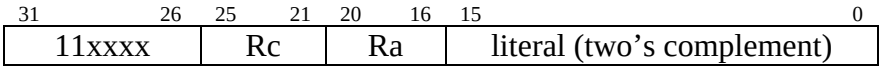
Operate Class:

Register	Symbol	Usage
R31	R31	Always zero
R30	XP	Exception pointer
R29	SP	Stack pointer
R28	LP	Linkage pointer
R27	BP	Base of frame pointer



OP(Ra,Rb,Rc): Reg[Rc] ← Reg[Ra] op Reg[Rb]

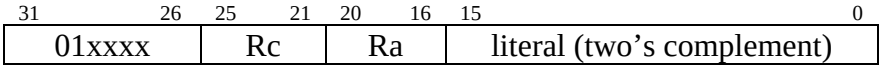
Opcodes: **ADD** (plus), **SUB** (minus), **MUL** (multiply), **DIV** (divided by)
AND (bitwise and), **OR** (bitwise or), **XOR** (bitwise exclusive or)
CMPEQ (equal), **CMPLT** (less than), **CMPLE** (less than or equal) [result = 1 if true, 0 if false]
SHL (left shift), **SHR** (right shift w/o sign extension), **SRA** (right shift w/ sign extension)



OPC(Ra,literal,Rc): Reg[Rc] ← Reg[Ra] op SEXT(literal)

Opcodes: **ADDC** (plus), **SUBC** (minus), **MULC** (multiply), **DIVC** (divided by)
ANDC (bitwise and), **ORC** (bitwise or), **XORC** (bitwise exclusive or)
CMPEQC (equal), **CMPLTC** (less than), **CMPLEC** (less than or equal) [result = 1 if true, 0 if false]
SHLC (left shift), **SHRC** (right shift w/o sign extension), **SRAC** (right shift w/ sign extension)

Other:



LD(Ra,literal,Rc): Reg[Rc] ← Mem[Reg[Ra] + SEXT(literal)]
ST(Rc,literal,Ra): Mem[Reg[Ra] + SEXT(literal)] ← Reg[Rc]
JMP(Ra,Rc): Reg[Rc] ← PC + 4; PC ← Reg[Ra]
BEQ/**BF**(Ra,label,Rc): Reg[Rc] ← PC + 4; if Reg[Ra] = 0 then PC ← PC + 4 + 4*SEXT(literal)
BNE/**BT**(Ra,label,Rc): Reg[Rc] ← PC + 4; if Reg[Ra] ≠ 0 then PC ← PC + 4 + 4*SEXT(literal)
LDR(label,Rc): Reg[Rc] ← Mem[PC + 4 + 4*SEXT(literal)]

Opcode Table: (*optional opcodes)

2:0 *5:3*	*000*	*001*	*010*	*011*	*100*	*101*	*110*	*111*
000								
001								
010								
011	**LD**	**ST**		**JMP**		**BEQ**	**BNE**	**LDR**
100	**ADD**	**SUB**	**MUL***	**DIV***	**CMPEQ**	**CMPLT**	**CMPLE**	
101	**AND**	**OR**	**XOR**		**SHL**	**SHR**	**SRA**	
110	**ADDC**	**SUBC**	**MULC***	**DIVC***	**CMPEQC**	**CMPLTC**	**CMPLEC**	
111	**ANDC**	**ORC**	**XORC**		**SHLC**	**SHRC**	**SRAC**	