MIT OpenCourseWare
http://ocw.mit.edu

6.004 Computation Structures
Spring 2009

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.

Building the Beta

Figure by MIT OpenCourseWare.

Lab #5 due Thursday

CPU Design Tradeoffs

Maximum Performance: measured by the numbers of instructions executed per second

Minimum Cost : measured by the size of the circuit.

Best Performance/Price: measured by the ratio of MIPS to size. In power-sensitive applications MIPS/Watt is important too.

Performance Measure

Millions of Instructions per Second

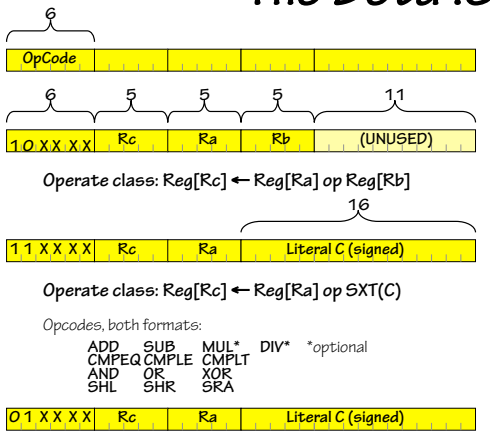
$$\text{MIPS} = \frac{\text{Clock Frequency (MHz)}}{\text{C.P.I.}}$$

Clocks per instruction

PUSHING PERFORMANCE ...

TODAY: 1 cycle/inst.
LATER: more MHz via pipelining

The Beta ISA



LD: Reg[Rc] ← Mem[Reg[Ra]+SXT(C)]
ST: Mem[Reg[Ra]+SXT(C)] ← Reg[Rc]
JMP: Reg[Rc] ← PC+4; PC ← Reg[Ra]
BEQ: Reg[Rc] ← PC+4; if Reg[Ra]=0 then PC ← PC+4+4*SXT(C)
BNE: Reg[Rc] ← PC+4; if Reg[Ra]≠0 then PC ← PC+4+4*SXT(C)
LDR: Reg[Rc] ← Mem[PC + 4 + 4*SXT(C)]

Instruction classes distinguished by OPCODE:
- OP
- OPC
- MEM
- Transfer of Control

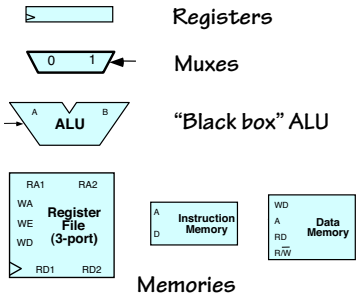
Approach: Incremental Featurism

Each instruction class can be implemented using a simple component repertoire. We'll try implementing data paths for each class individually, and merge them (using MUXes, etc).

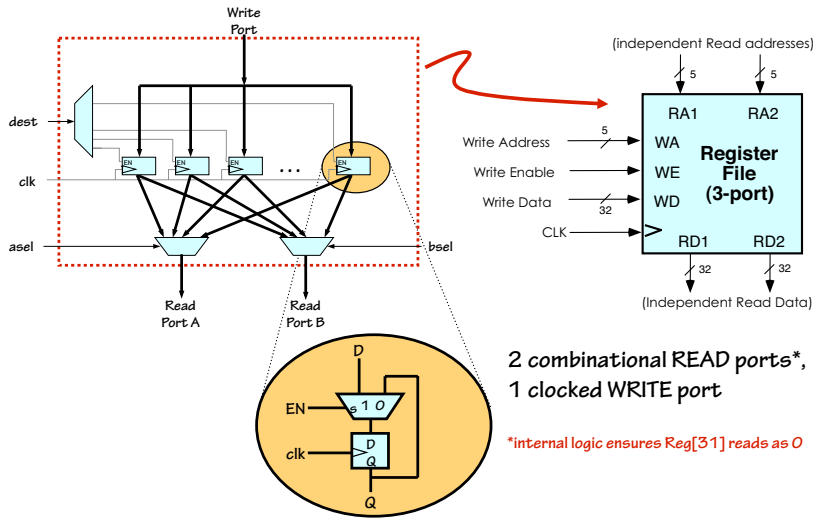
Steps:

1. Operate instructions
2. Load & Store Instructions
3. Jump & Branch instructions
4. Exceptions
5. Merge data paths

Our Bag of Components:



Multi-Port Register Files

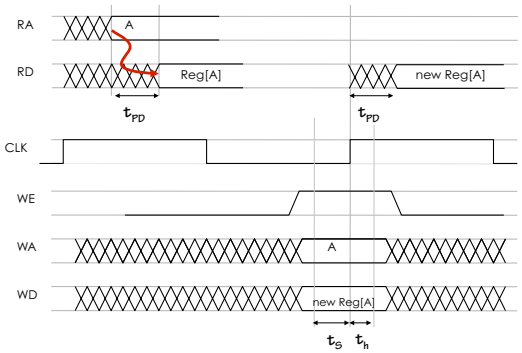


2 combinational READ ports*, 1 clocked WRITE port

*internal logic ensures Reg[31] reads as 0

Register File Timing

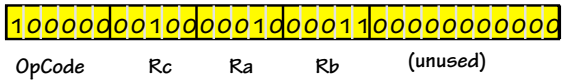
2 combinational READ ports, 1 clocked WRITE port



What if (say) WA=RA1???
RD1 reads "old" value of Reg[RA1] until next clock edge!

Starting point: ALU Ops

32-bit (4-byte) ADD instruction:



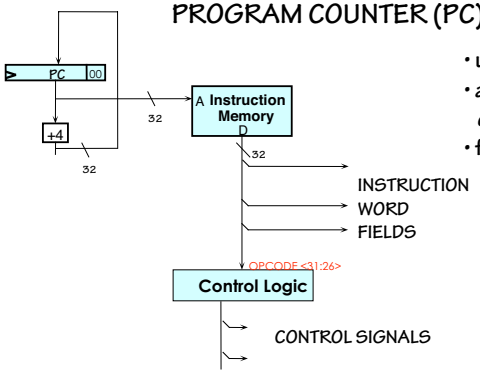
Means, to BETA, Reg[R4] ← Reg[R2] + Reg[R3]

First, we'll need hardware to:

- Read next 32-bit instruction
- DECODE instruction: ADD, SUB, XOR, etc
- READ operands (Ra, Rb) from Register File;
- PERFORM indicated operation;
- WRITE result back into Register File (Rc).

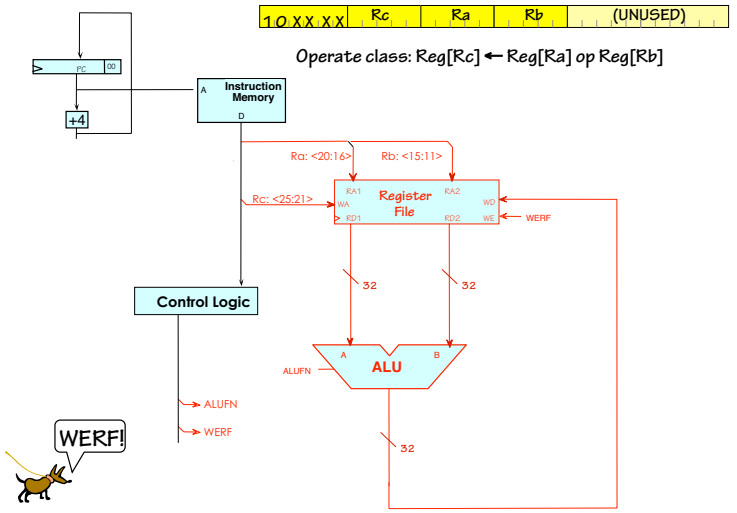
Instruction Fetch/Decode

- Use a counter to FETCH the next instruction: PROGRAM COUNTER (PC)

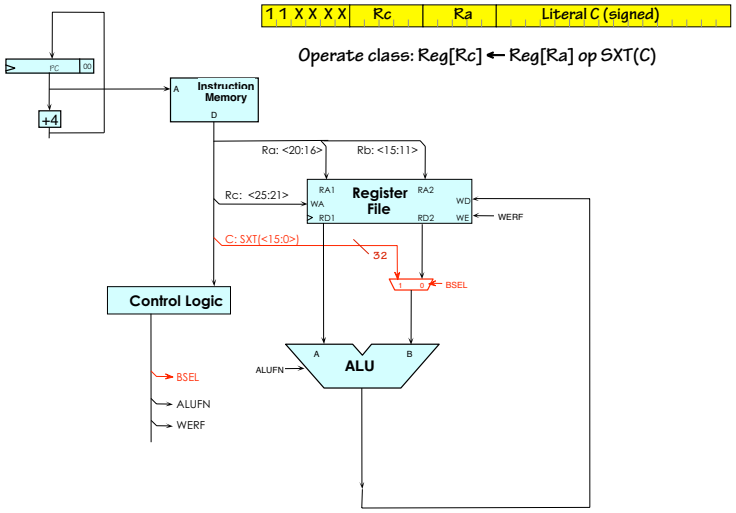


- use PC as memory address
- add 4 to PC, load new value at end of cycle
- fetch instruction from memory
  - use some instruction fields directly (register numbers, 16-bit constant)
  - use bits <31:26> to generate controls

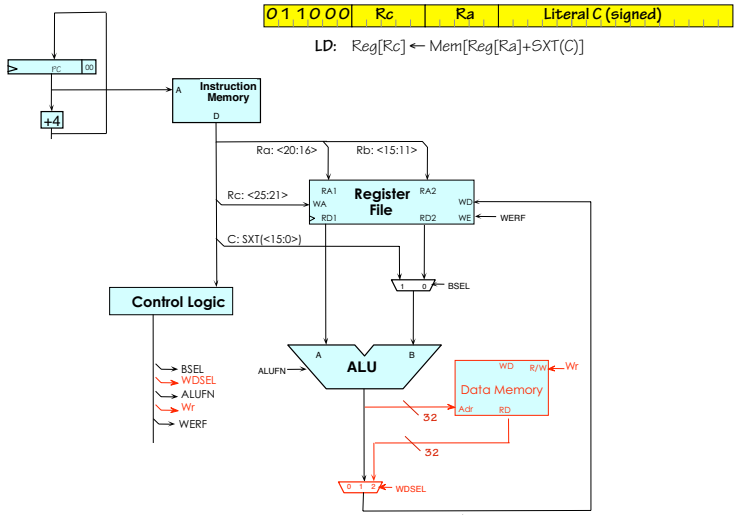
ALU Op Data Path



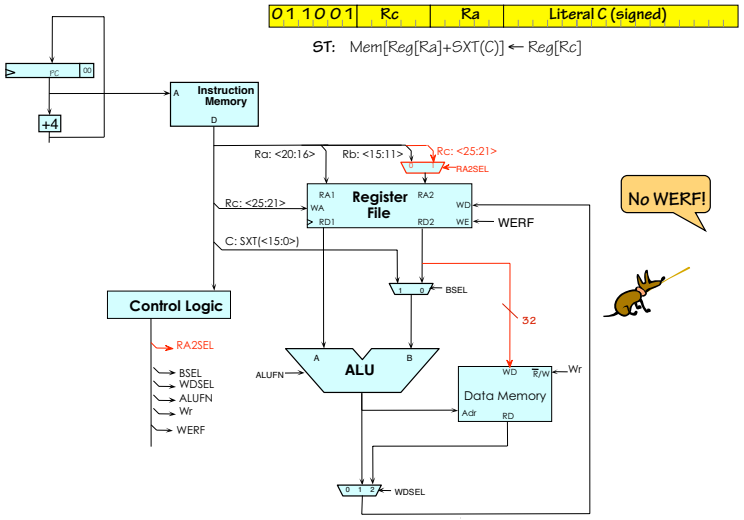
ALU Operations (w/constant)



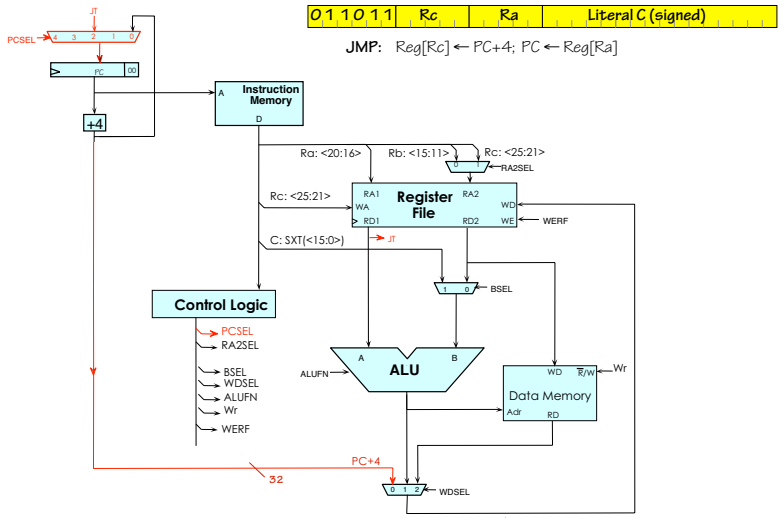
Load Instruction



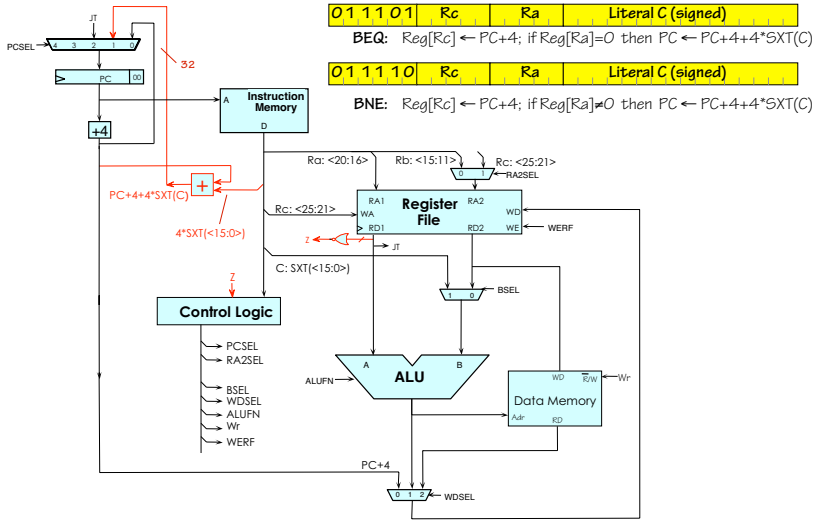
Store Instruction



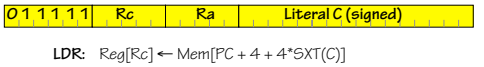
JMP Instruction



BEQ/BNE Instructions



Load Relative Instruction



LDR: Reg[Rc] ← Mem[PC + 4 + 4*SXT(C)]

Hey, WAIT A MINUTE. What's Load Relative good for anyway??? I thought

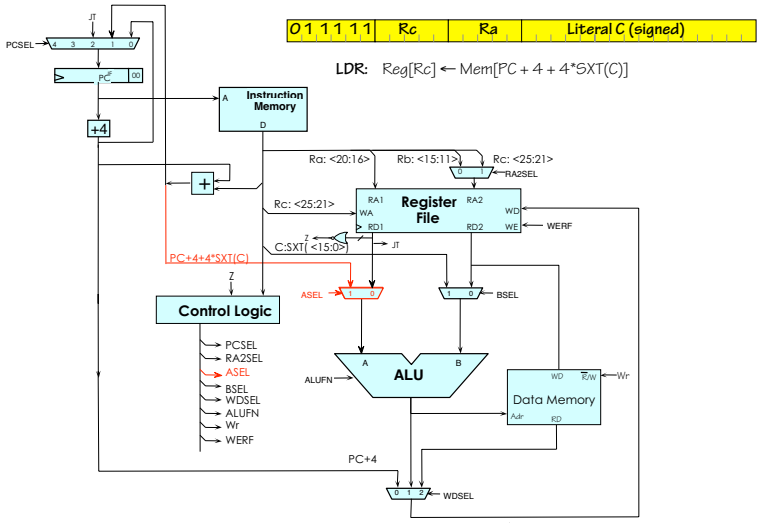
- Code is "PURE", i.e. READ-ONLY; and stored in a "PROGRAM" region of memory;
- Data is READ-WRITE, and stored either
  - On the STACK (local); or
  - In some GLOBAL VARIABLE region; or
  - In a global storage HEAP.

So why an instruction designed to load data that's "near" the instruction???
Addresses & other large constants

```
C:    X = X * 123456;

BETA:
      LD(X, r0)
      LDR(c1, r1)
      MUL(r0, r1, r0)
      ST(r0, X)
      ...
c1:   LONG(123456)
```

LDR Instruction



Exceptions

What if something BAD happens?

- Execution of an illegal op-code
- Reference to non-existent memory
- Divide by zero

Or, maybe, just something unanticipated…

- User hits a key
- A packet comes in via the network

GOAL: handle all these cases (and more) in *SOFTWARE:*

- Treat each such case as an (implicit) procedure call…
- Procedure handles problem, returns to interrupted program.
- *TRANSPARENT* to interrupted program!
- Important added capability: handlers for certain errors (illegal op-codes) can extend instruction set using software (Lab 7!).

Exception Processing

Plan:

- Interrupt running program
- Invoke exception handler (like a procedure call)
- Return to continue execution.

We'd like RECOVERABLE INTERRUPTS for

- Synchronous events, generated by CPU or system
  FAULTS (eg, Illegal Instruction, divide-by-0, illegal mem address)
  TRAPS & system calls (eg, read-a-character)

- Asynchronous events, generated by I/O
  (eg, key struck, packet received, disk transfer complete)

KEY: TRANSPARENCY to interrupted program.

- Most difficult for asynchronous interrupts

Implementation…

How exceptions work:

- Don't execute current instruction
- Instead fake a "forced" procedure call
  - save current PC (actually current PC + 4)
  - load PC with *exception vector*
    - 0x4 for synch. exception, 0x8 for asynch. exceptions

Question: where to save current PC + 4?

- Our approach: reserve a register (R30, aka XP)
- Prohibit user programs from using XP. Why?

Example: DIV unimplemented

```
LD(R31,A,R0)
LD(R31,B,R1)
DIV(R0,R1,R2)
ST(R2,C,R31)
```

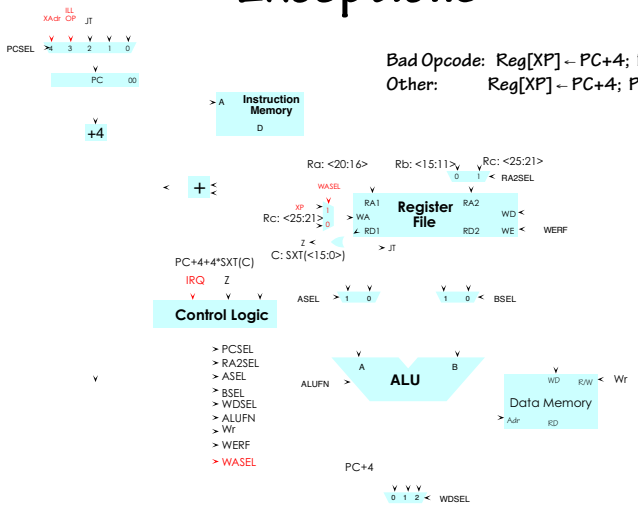
Forced by hardware

```
IllOp:
    PUSH(XP)
```

Fetch inst. at Mem[Reg[XP]-4]
check for DIV opcode, get reg numbers
perform operation in SW, fill result reg

```
    POP(XP)
    JMP(XP)
```

Exceptions



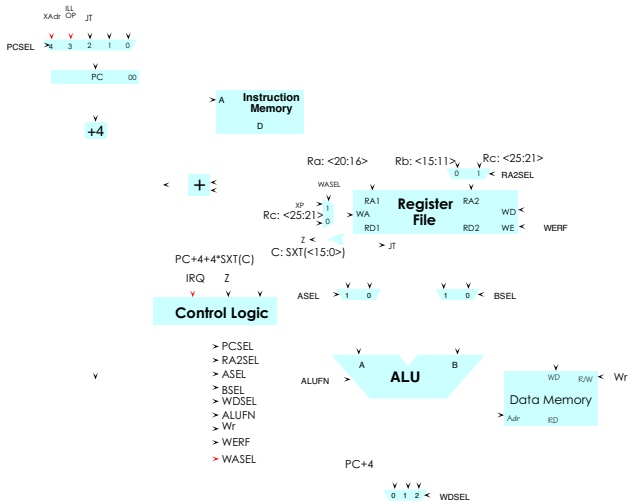
Control Logic

	OP	OPC	LD	ST	JMP	BEQ	BNE	LDR	Illop	IRQ
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

Implementation choices:

- ROM indexed by opcode, external branch & trap logic
- PLA
- "random" logic (eg, standard cell gates)

Beta: Our "Final Answer"



Next Time: Tackling the Memory Bottleneck