Building the Beta
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

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

Millions of Instructions per Second

PUSHING PERFORMANCE ...

TODAY: 1 cycle/inst.
NEXT TIME: more MHz via pipelining
NEXT NEXT TIME: fixing various pipelining issues
The Beta ISA

### OPCODE

<table>
<thead>
<tr>
<th>10 x x x x</th>
<th>Rc</th>
<th>Ra</th>
<th>Rb</th>
<th>(unused)</th>
</tr>
</thead>
</table>

Operate class: \(\text{Reg}[Rc] \leftarrow \text{Reg}[Ra] \text{ op Reg[Rb]}\)

### OPCODE

<table>
<thead>
<tr>
<th>11 x x x x</th>
<th>Rc</th>
<th>Ra</th>
<th>literal C (signed)</th>
</tr>
</thead>
</table>

Operate class: \(\text{Reg}[Rc] \leftarrow \text{Reg}[Ra] \text{ op SXT}(C)\)

Opcodes, both formats:
- ADD
- SUB
- MUL
- DIV
- CMPEQ
- CMPLE
- CMPLT
- AND
- OR
- XOR
- SHL
- SHR
- SRA

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

### LD:
\(\text{Reg}[Rc] \leftarrow \text{Mem}[\text{Reg}[Ra] + \text{SXT}(C)]\)

### ST:
\(\text{Mem}[\text{Reg}[Ra] + \text{SXT}(C)] \leftarrow \text{Reg}[Rc]\)

### JMP:
\(\text{Reg}[Rc] \leftarrow \text{PC}+4; \text{PC} \leftarrow \text{Reg}[Ra]\)

### BEQ:
\(\text{Reg}[Rc] \leftarrow \text{PC}+4; \text{if \text{Reg}[Ra]}=0 \text{ then } \text{PC} \leftarrow \text{PC}+4+4\times\text{SXT}(C)\)

### BNE:
\(\text{Reg}[Rc] \leftarrow \text{PC}+4; \text{if \text{Reg}[Ra]}=0 \text{ then } \text{PC} \leftarrow \text{PC}+4+4\times\text{SXT}(C)\)

### LDR:
\(\text{Reg}[Rc] \leftarrow \text{Mem}[\text{PC} + 4 + 4\times\text{SXT}(C)]\)
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:
- Registers
- Muxes
- "Black box" ALU
- Register File (3-port)
- Instruction Memory
- Data Memory
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:

```
100000001000011000000000000000
```

OpCode   Rc   Ra   Rb   (unused)

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

Instruction

Memory

RA1 RA2
RD1 RD2
WA WD

Control Logic

ALU

ALUFN
WERF

Rc: <25:21>

Ra: <20:16>

Rb: <15:11>

1 0 XXXX

Rc Ra Rb unused

OP: Reg[Rc] ← Reg[Ra] op Reg[Rb]

ALUFN
WERF

32

32

32
ALU Operations (w/constant)

Instruction Memory

Rc: <25:21>

PC +4

Register File

Ra: <20:16>
Rb: <15:11>
Rc: <25:21>
C: SXT(<15:0>)

Control Logic

ALU

BSEL
ALUFN
WERF

 OPC: Reg[Rc] ← Reg[Ra] op SXT(C)

1 1 XXXX  Rc  Ra  literal C (signed)
Load Instruction

LD: Reg[Rc] ← Mem[Reg[Ra]+SXT(C)]
Store Instruction
JMP Instruction

JMP: Reg[Rc] ← PC+4; PC ← Reg[Ra]
BEQ/BNE Instructions

**BEQ (Branch if Equal)**
- **Instruction:** 0111 01  
- **Description:** If Reg[Ra] = 0, then PC ← PC + 4 + 4*SXT(C)
- **Diagram:**
  - Control Logic
  - Instruction Memory
  - ALU
  - Register File
  - Data Memory
  - Control Logic

**BNE (Branch if Not Equal)**
- **Instruction:** 0111 10  
- **Description:** If Reg[Ra] ≠ 0, then PC ← PC + 4 + 4*SXT(C)
- **Diagram:**
  - Control Logic
  - Instruction Memory
  - ALU
  - Register File
  - Data Memory
  - Control Logic

**Notes:**
- Rb: <15:11>
- Ra: <20:16>
- Rc: <25:21>
- SXT(<15:0>)
- Wr: Write Enable
- WE: Write Enable
- WERF: Write Enable Register File
- PCSEL: Program Counter Select
- RA2SEL: Register File Select A
- BSEL: Branch Enable Select
- WDSEL: Write Data Enable Select
- ALUFN: ALU Function
- ALU: Arithmetic Logic Unit
- RA1: Register File A
- RA2: Register File B
- RD1: Register File D
- RD2: Register File E
- PC: Program Counter
- JT: Jump Table
- 4*SXT(<15:0>): 4 times SXT of the first 15 bits
- PC+4: Program Counter + 4
- PC+4+4*SXT(C): Program Counter + 4 + 4*SXT of the first 15 bits
Load Relative Instruction

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)
  ...

cl:  LONG(123456)
```
LDR Instruction

```
| 011111 | Rc | Ra | literal C (signed) |

LDR: Reg[Rc] ← Mem[PC + 4 + 4*SXT(C)]

PCSEL → 4 3 2 1 0

Control Logic

Instruction Memory

Data Memory

Register File

ALU

PC+4 +4*SXT(C)

JT

 +/-

RA2SEL

ASEL

BSEL

ALUFN

RD

owe

WA WD WE

0 1 0

Rc: <25:21>

Rc: <25:21>

Ra: <20:16>

Rb: <15:11>

0

Rc: <25:21>

C:SXT(<15:0>)

JT

ASEL

1 0

ALUFN

A B

ALU

Data Memory

Adr

RD

PC4

WE

0 1 2

WR

WDSEL
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

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

IllOp:

```asm
PUSH(XP)
```

Forced by hardware

Fetch inst. at Mem[Reg[XP]–4]

check for DIV opcode, get reg numbers

perform operation in SW, fill result reg

```asm
POP(XP)
JMP(XP)
```
Exceptions

Bad Opcode:  Reg[XP] ← PC+4; PC ← “IllOp”
Other:       Reg[XP] ← PC+4; PC ← “Xadr”
# Control Logic

<table>
<thead>
<tr>
<th></th>
<th>OP</th>
<th>OPC</th>
<th>LD</th>
<th>ST</th>
<th>JMP</th>
<th>BEQ</th>
<th>BNE</th>
<th>LDR</th>
<th>Il/op</th>
<th>trap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ALUFN</strong></td>
<td>F(op)</td>
<td>F(op)</td>
<td>“+”</td>
<td>“+”</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>“A”</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><strong>WERF</strong></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>BSEL</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><strong>WDSEL</strong></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>WR</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>RA2SEL</strong></td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><strong>PCSEL</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>$Z \ ? 1 : 0$</td>
<td>$Z \ ? 0 : 1$</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td><strong>ASEL</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><strong>WASEL</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Implementation choices:

- ROM indexed by opcode, external branch & trap logic
- PLA
- “random” logic (eg, standard cell gates)
Beta: Our “Final Answer”
Next Time: Pipelined Betas

Hey, building a computer is not really all that difficult

So let’s run down to the 6.004 lab and put Intel out of business!