

15-213

“The course that gives CMU its Zip!”

Machine-Level Programming I: Introduction Sept. 8, 1998

Topics

- Assembly Programmer’s Execution Model
- Arithmetic Instructions
- Memory Instructions
- Transfers of Control
- Comparison to actual Alpha 21164 processor

Alpha Processors

Reduced Instruction Set Computer (RISC)

- Simple instructions with regular formats
- Key Idea: *make the common cases fast!*
 - infrequent operations can be synthesized using multiple instructions

Assumes compiler will do optimizations

- e.g., scalar optimization, register allocation, scheduling, etc.
- ISA designed for *compilers*, not assembly language programmers

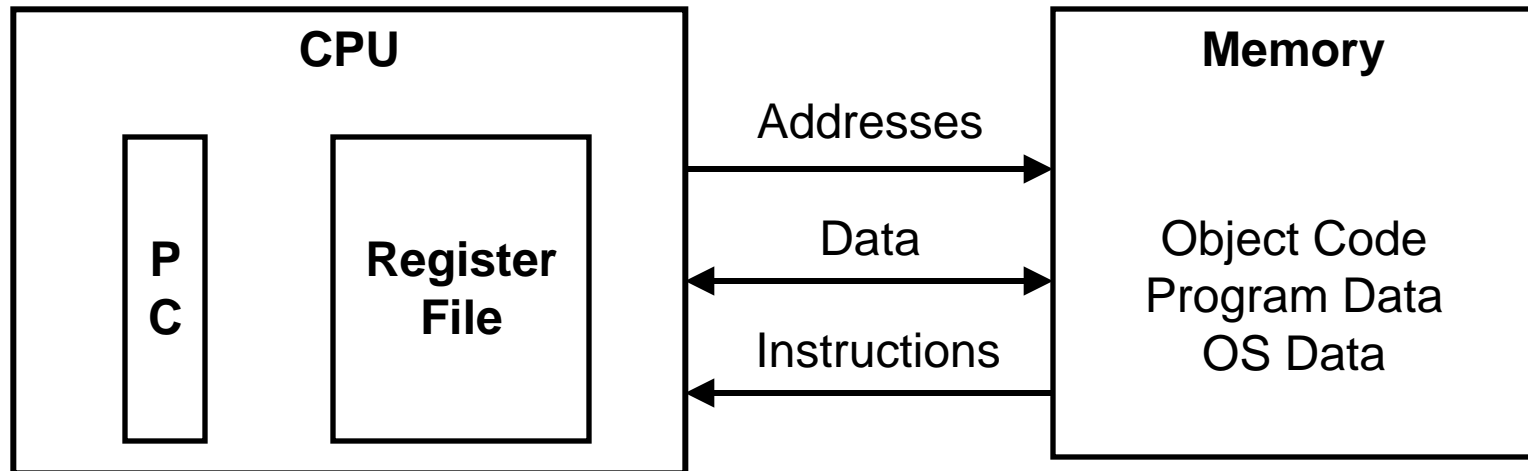
A 2nd Generation RISC Instruction Set Architecture

- Designed for superscalar processors (> 1 inst per cycle)
- Designed as a 64-bit ISA from the start

Very High Performance Machines

- Alpha has been the clear performance leader for many years now

Assembly Programmer's View

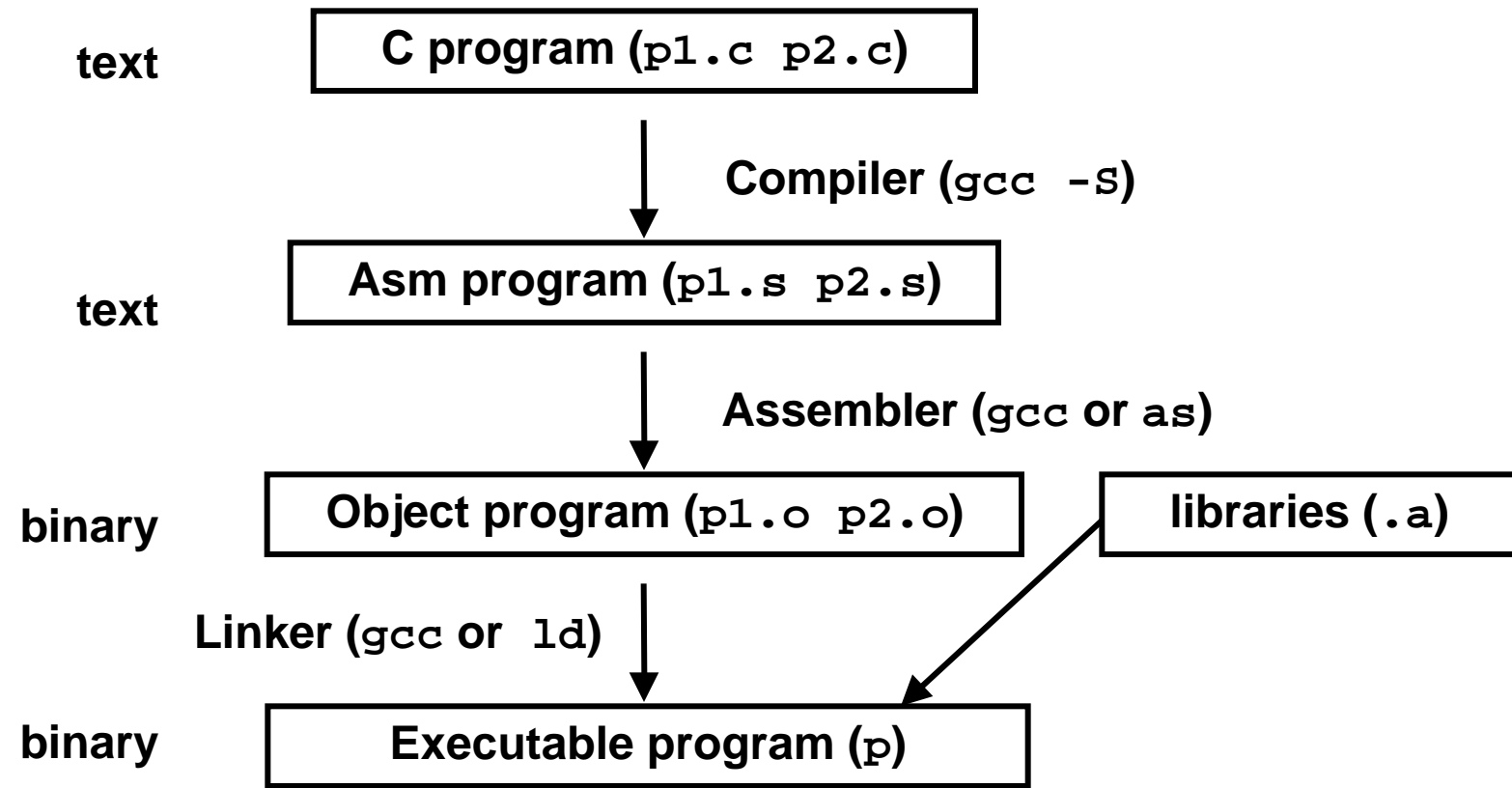


Programmer-Visible State

- **PC** **Program Counter**
 - Address of next instruction
- **Register File**
 - Heavily used program data
- **Memory**
 - Byte addressable array
 - Code, user data, (some) OS data

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -O p1.c p2.c -o p`
 - Use optimizations (`-O`)
 - Put resulting binary in file `p`



Compiling Into Assembly

C Code

```
long int arith(long int x,  
              long int y, long int z)  
{  
    long int t1 = x+y;  
    long int t2 = z+t1;  
    long int t3 = x+4L;  
    long int t4 = y * 48L;  
    long int t5 = t3 + t4;  
    long int rval = t2 * t5;  
    return rval;  
}
```

Obtain with command

```
gcc -O -S code.c
```

Produces file code.s

Generated Assembly

```
arith:  
arith..ng:  
    .frame $30,0,$26,0  
    .prologue 0  
    addq    $16,$17,$1  
    addq    $18,$1,$0  
    addq    $16,4,$16  
    s4subq  $17,$17,$17  
    sll     $17,4,$17  
    addq    $16,$17,$16  
    mulq    $0,$16,$0  
    ret     $31,($26),1  
    .end arith
```

Assembly Characteristics

Minimal Data Types

- **“Integer” data of 1, 2, 4, or 8 bytes**
 - Data values
 - Addresses (untyped pointers)
- **Floating point data of 4 or 8 bytes**
- **No aggregate types such as arrays or structures**
 - Just contiguously allocated bytes in memory

Primitive Operations

- **Perform arithmetic function on register data**
- **Transfer data between memory and register**
 - Load data from memory into register
 - Store register data into memory
- **Transfer control**
 - Unconditional jumps to/from procedures
 - Conditional branches

Object Code

Code for arith

```
0x1200012d0:  
  0x42110401  
  0x42410400  
  0x42009410  
  0x42310571  
  0x4a209731  
  0x42110410  
  0x4c100400  
  0x6bfa8001
```

- Total of 8 instructions
- Each 4 bytes
- Starts at address
0x1200012d0

Assembler

- Translates `.s` into `.o`
- Binary encoding of each instruction
- Nearly-complete image of executable code
- Missing linkages between code in different files

Linker

- Resolves references between files
- Combines with run-time libraries
 - At the very least includes `libc.a`
 - E.g., code for `malloc`, `printf`

Machine Instruction Example

C Code

```
long int t1 = x+y;
```

- Add two signed 64-bit integers

Assembly

```
addq $16,$17,$1
```

- Add 2 8-byte integers
 - “Quad” words in Alpha parlance
 - Same instruction whether signed or unsigned
- Operands & result in registers:
 - \$16: x
 - \$17: y
 - \$1: t1

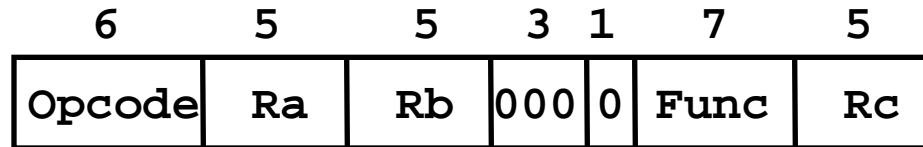
Object Code

```
0x1200012d0: 0x42110401
```

- 32-bit pattern
- Stored at address 0x1200012d0

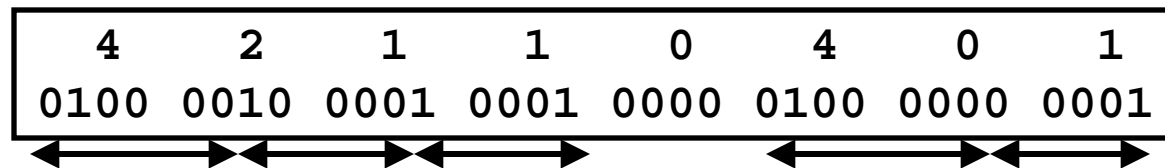
Encoding of Machine Instruction

Format for Register-Register Arithmetic Instructions

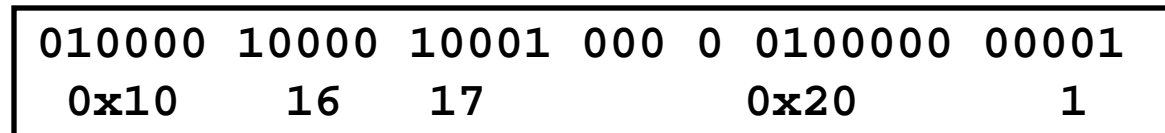


- Opcode + Func fields determine operation <op>
- Effect is $\text{Reg}[\text{Rc}] = \text{Reg}[\text{Ra}] \text{ <op> } \text{Reg}[\text{Rb}]$

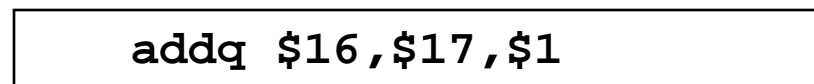
Decoding Hex Format



Regrouping into Instruction Fields



- Opcode 0x10 + Func 0x20 encodes `addq`



Disassembling Object Code

Object

```
0x1200012d0:  
  0x42110401  
  0x42410400  
  0x42009410  
  0x42310571  
  0x4a209731  
  0x42110410  
  0x4c100400  
  0x6bfa8001
```

Disassembled

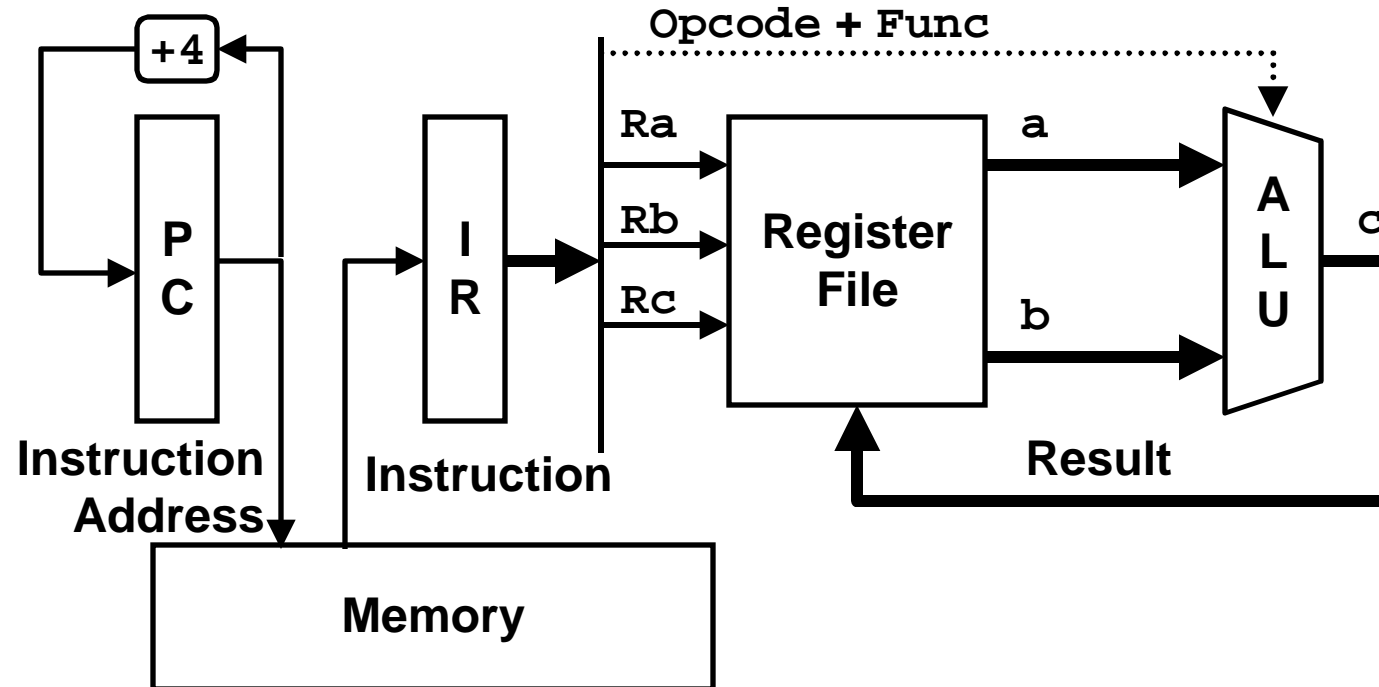
```
0x1200012d0: 42110401 addq   r16, r17, r1  
0x1200012d4: 42410400 addq   r18, r1, r0  
0x1200012d8: 42009410 addq   r16, 0x4, r16  
0x1200012dc: 42310571 s4subq r17, r17, r17  
0x1200012e0: 4a209731 sll   r17, 0x4, r17  
0x1200012e4: 42110410 addq   r16, r17, r16  
0x1200012e8: 4c100400 mulq  r0, r16, r0  
0x1200012ec: 6bfa8001 ret   r31, (r26), 1
```

Disassembler

```
dis -h p
```

- Useful tool for examining object code
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either a .out (complete executable) or .o file

Implementing Arithmetic Operations



1. Fetch

$IR = Mem[PC]$

$a = Reg[Ra]$

$b = Reg[Rb]$

2. Execute

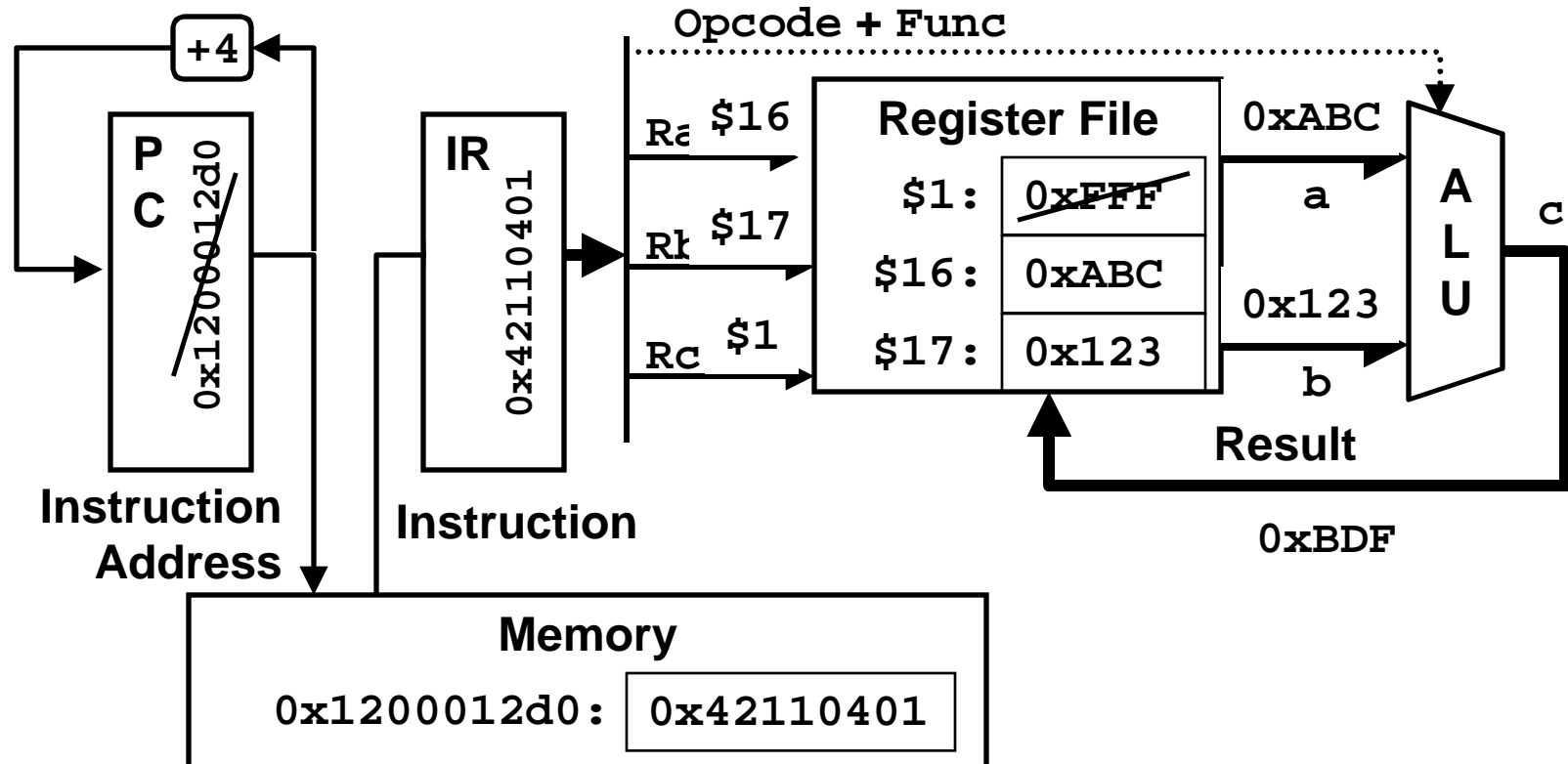
$c = a <op> b$

3. Update

$Reg[Rc] = c$

$PC = PC + 4$

Arithmetic Operation Example



1. Fetch

$IR = Mem[PC]$

$a = Reg[Ra]$

$b = Reg[Rb]$

2. Execute

$c = a <op> b$

3. Update

$Reg[Rc] = c \quad 0xBDF$

$PC = PC + 4 \quad 0x1200012d4$

Executing Arith

```

arith(x, y, z)
{
    t1 = x+y;
    t2 = z+t1;
    t3 = x+4L;
    t4 = y * 48L;
    t5 = t3 + t4;
    rval = t2 * t5;
    return rval;
}

```

Op	a <op> b
addq	a + b
s4subq	4*a - b
sll	a << b
mulq	a * b

```

addq $16,$17,$1
addq $18,$1,$0
addq $16,4,$16
s4subq $17,$17,$17
sll $17,4,$17
addq $16,$17,$16
mulq $0,$16,$0
ret $31,($26),1

```

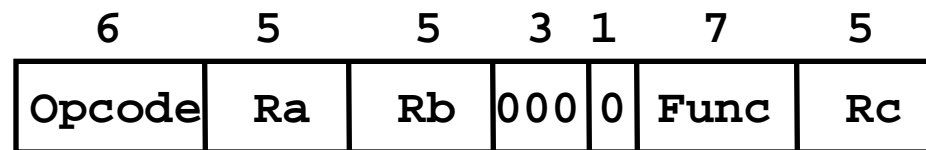
\$0	\$1	\$16	\$17	\$18
-	-	x	y	z
-	t1	x	y	z
t2	t1	x	y	z
t2	t1	t3	y	z
t2	t1	t3	3*y	z
t2	t1	t3	t4	z
t2	t1	t5	t4	z
rval	t1	t5	t4	z

Logical Operation Example

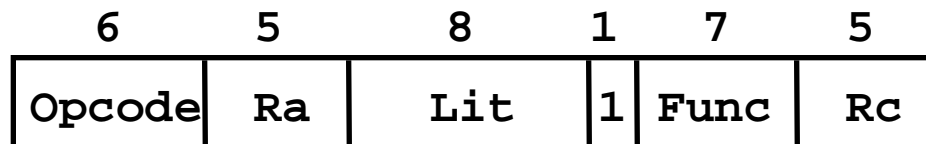
```
long int logical(long int x, long int y)
{
    long int t1 = x^y;
    long int t2 = t1 >> 17;
    long int mask = (1L<<13) - 7L;
    long int rval = t2 & mask;
    return rval;
}
```

- Logical operations similar to arithmetic operations
- Both have two formats

–RR: Operands **a** and **b** from registers



–RI: Operand **a** from register, **b** is 8-bit unsigned “literal”



Executing Logical

```
logical(x, y)
{
  t1 = x^y;
  t2 = t1 >> 17;
  mask = (1L<<13) - 7L;
  rval = t2 & mask;
  return rval;
}
```

Op	a <op> b
xor	a ^ b
sra	a >> b
and	a & b

```
xor $16,$17,$17
sra $17,17,$17
lda $0,8185
and $17,$0,$0
ret $31,($26),1
```

Special trick to get big constant

\$0	\$16	\$17
-	x	y
-	x	t1
-	x	t2
mask	x	t2
rval	x	t2

1L << 13 = 8192
8192 - 7 = 8185

Load & Store Instructions

Load Operation

- Read (load) from memory
- Write to register

`ldq Ra, Offset(Rb)`

- Arguments

Ra Destination Reg.
Rb Base address Reg.
Offset Offset from base
 » Between -32,768 and
 32,767

- Effective Address

$EA = \text{Reg}[Rb] + \text{Offset}$

- Operation

$\text{Reg}[Ra] = \text{Mem}[EA]$

Store Operation

- Read from register
- Write (store) to memory

`stq Ra, Offset(Rb)`

- Arguments

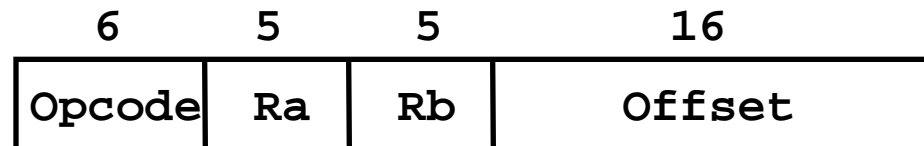
Ra Source Reg.
Rb Base address Reg.
Offset Offset from base
 » Between -32,768 and
 32,767

- Effective Address

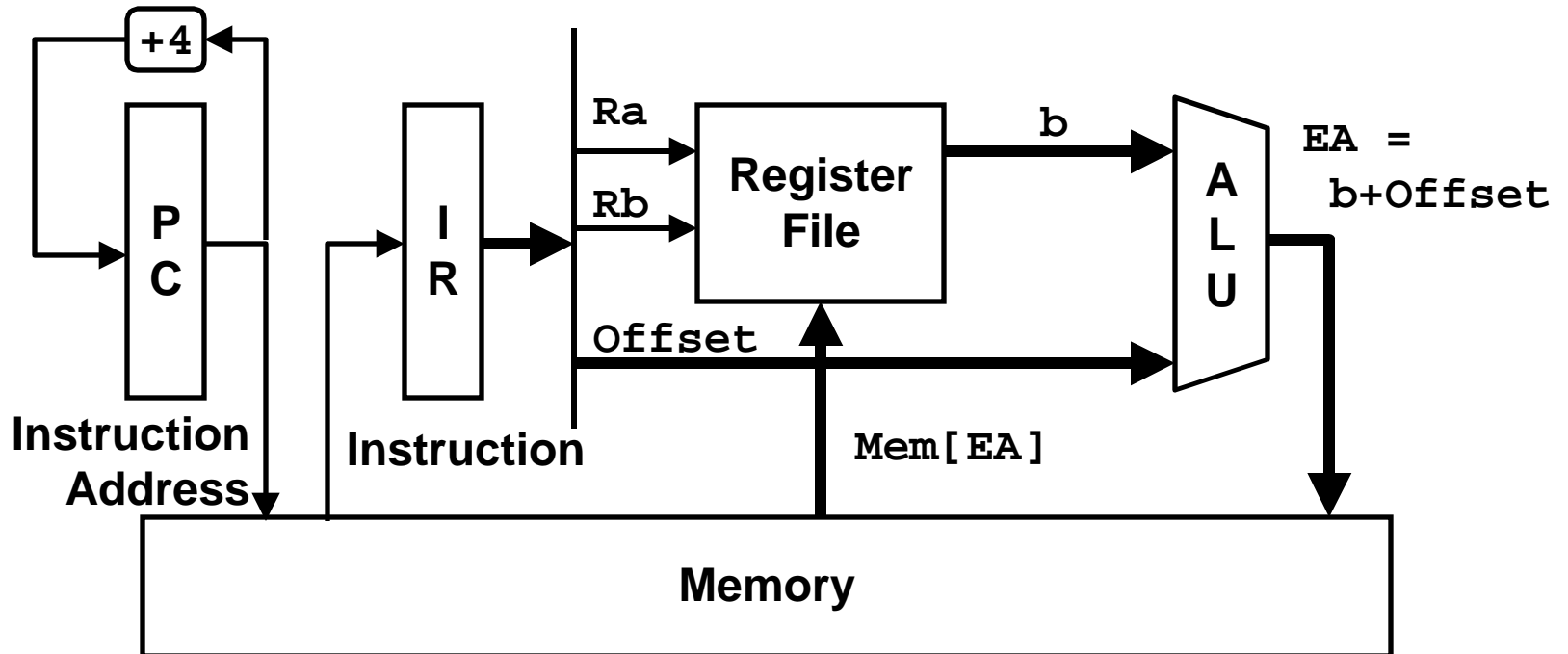
$EA = \text{Reg}[Rb] + \text{Offset}$

- Operation

$\text{Mem}[EA] = \text{Reg}[Ra]$



Implementing Load Operation



1. Fetch

$IR = Mem[PC]$

$b = Reg[Rb]$

2. Execute

$EA = b + Offset$

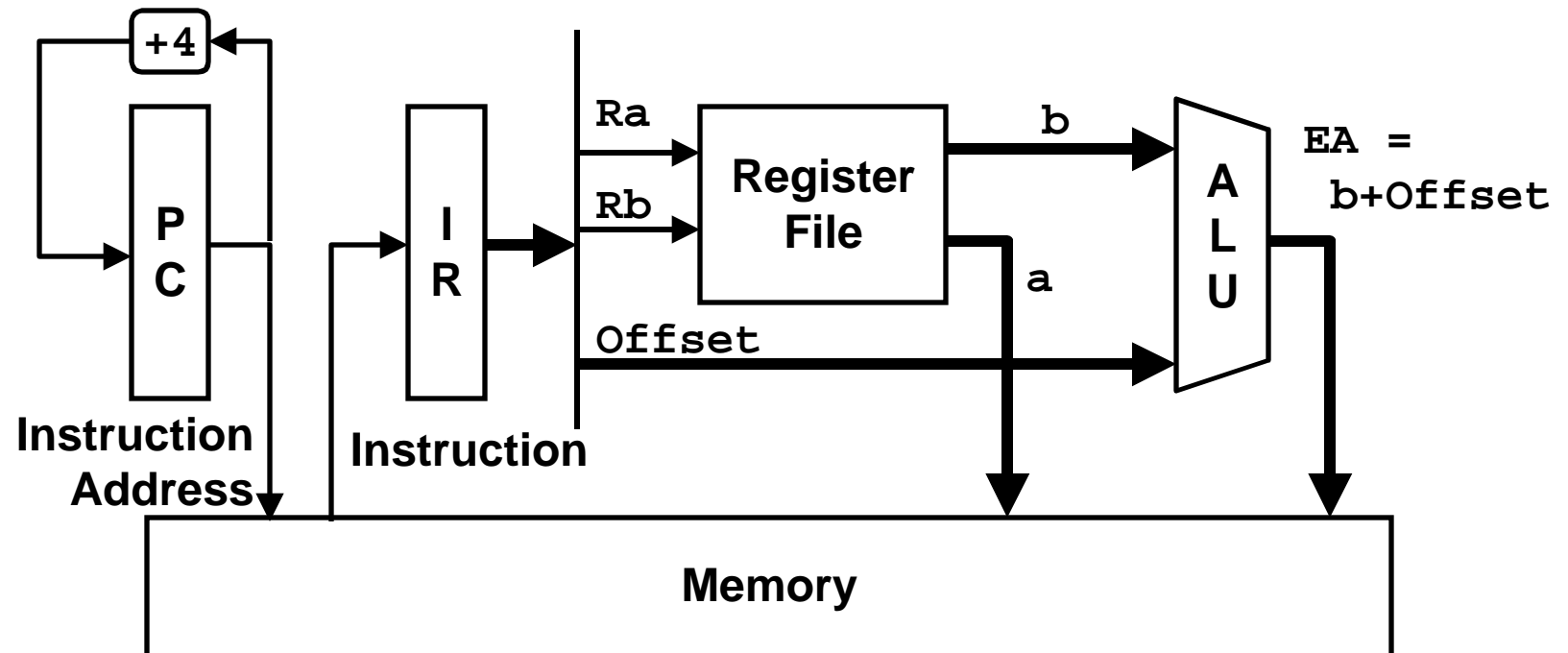
$Result = Mem[EA]$

3. Update

$Reg[Ra] = Result$

$PC = PC + 4$

Implementing Store Operation



1. Fetch

$IR = \text{Mem}[PC]$

$a = \text{Reg}[Ra]$

$b = \text{Reg}[Rb]$

2. Execute

$EA = b + \text{Offset}$

3. Update

$\text{Mem}[EA] = a$

$PC = PC + 4$

Load & Store Example

```
void swap(long int *xp, long int *yp)
{
    long int t0 = *xp;
    long int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Realization of C Pointers

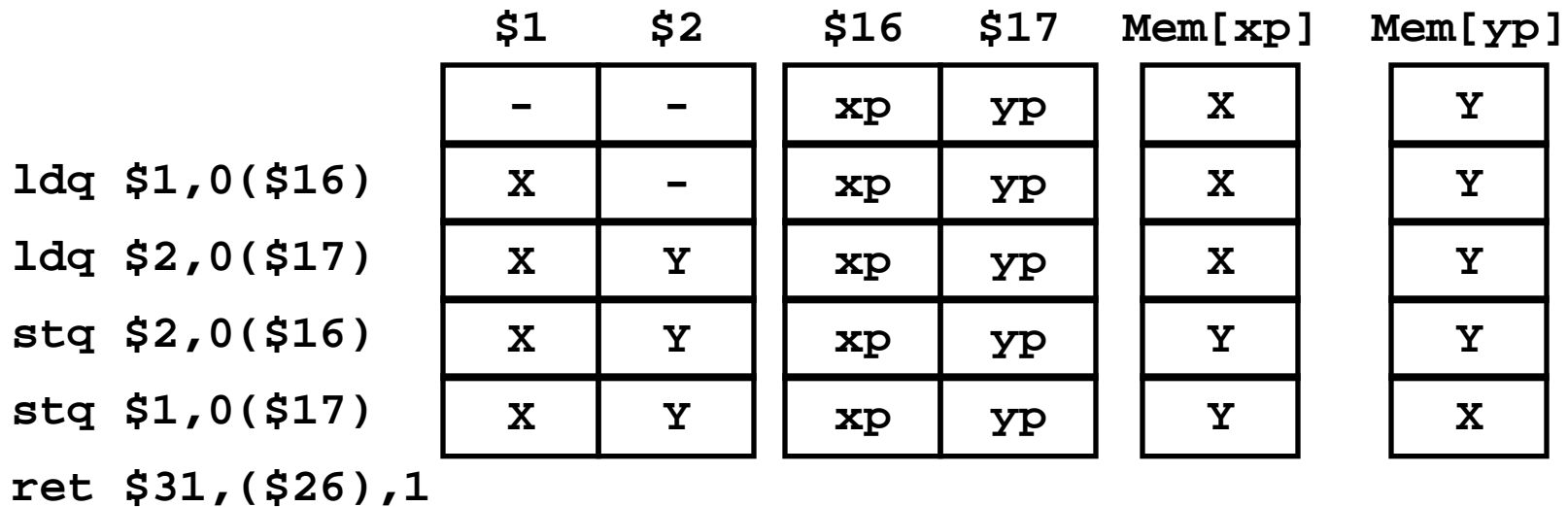
- **Pointer is address of object**
- **Manipulated as 64-bit signed integer**
- **Machine has no notion of pointer type**
 - Does not distinguish `(char *)`, `(int *)`, `(int **)`, etc.

Executing Swap

```

swap(*xp, *yp)
{
    t0 = *xp;
    t1 = *yp;
    *xp = t1;
    *yp = t0;
}

```



XOR-Based Swap

```
void xor_swap(long int *xp,  
             long int *yp)  
{  
    *xp = *xp ^ *yp;  
    *yp = *xp ^ *yp;  
    *xp = *xp ^ *yp;  
}
```

```
ldq $2,0($16)  
ldq $1,0($17)  
xor $2,$1,$2  
stq $2,0($16)  
ldq $1,0($17) ← = $1?  
xor $2,$1,$2  
stq $2,0($17)  
ldq $1,0($16) ← Reuse  
xor $1,$2,$1    earlier  
stq $1,0($16)   result?  
ret $31,($26),1
```

Very Inefficient Code

- 4 loads, 3 stores, 3 ALU ops
- vs. 2 loads, 2 stores for swap

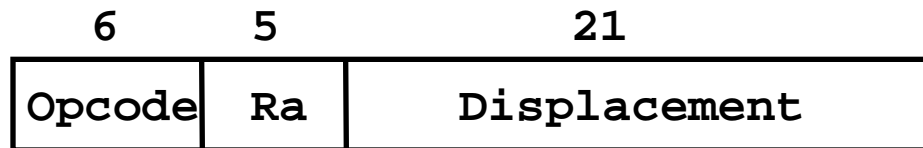
Apparent Non-Optimality

- Why does it need to keep reloading operand?
- Hint: required to preserve behavior for special argument combination

Conditional Branch Instructions

Format

`bCOND Ra, target`



- *COND* describes branch condition
- `target` computed from `PC+4` and `Displacement`
- Assembler allows use of symbolic labels

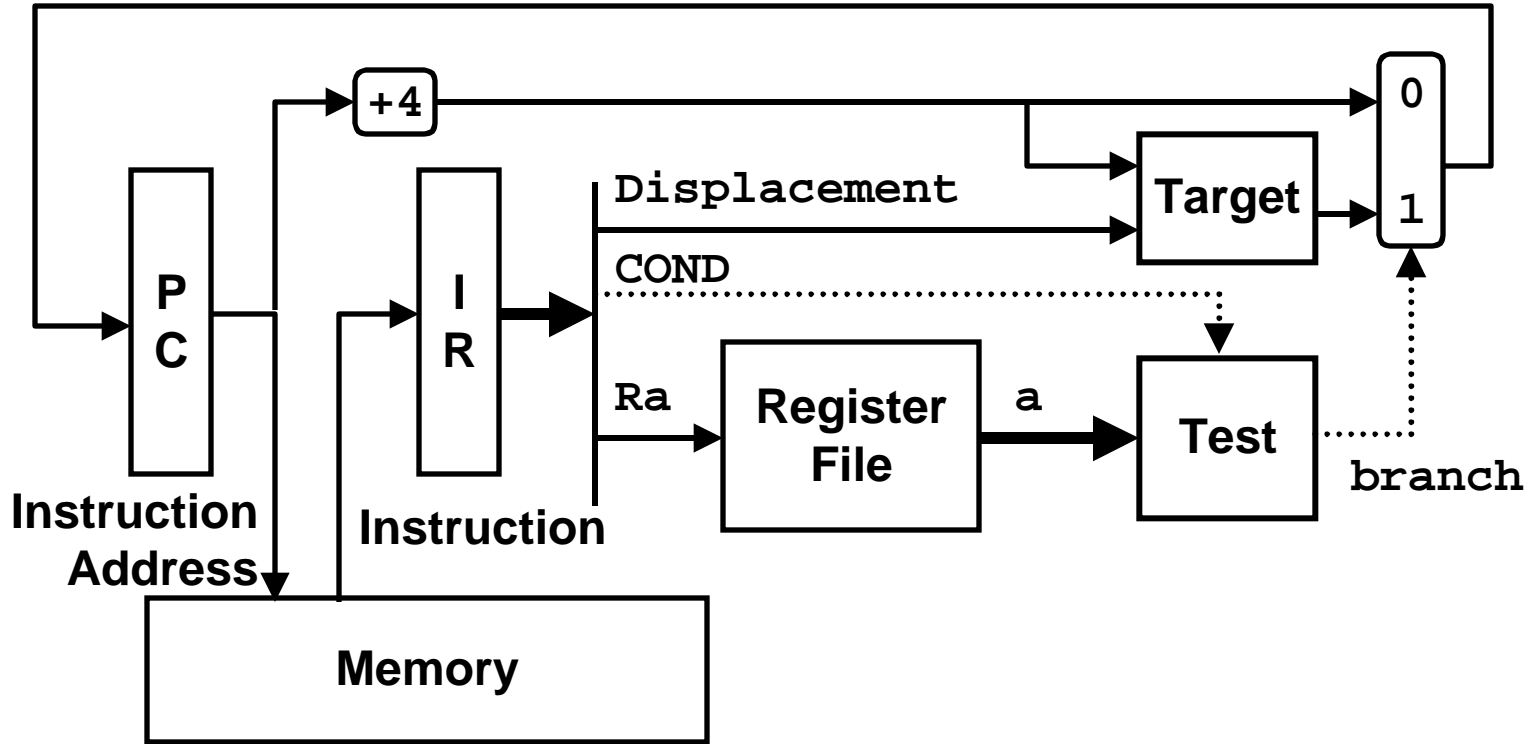
Operation

- Compare `Reg[Ra]` to 0
- Go to `target` if condition satisfied
- Otherwise proceed with next instruction

Possible Conditions

<code>eq</code>	<code>a == 0</code>
<code>ne</code>	<code>a != 0</code>
<code>gt</code>	<code>a > 0</code>
<code>ge</code>	<code>a >= 0</code>
<code>lt</code>	<code>a < 0</code>
<code>le</code>	<code>a <= 0</code>
<code>lbc</code>	<code>a&1 == 0</code>
<code>lbs</code>	<code>a&1 != 0</code>

Implementing Branch Operation



1. Fetch

```
IR = Mem[PC]
a = Reg[Ra]
```

2. Execute

```
branch = Cond(a)
target =
    Target(PC+4,
           Displacement)
```

3. Update

```
if (branch)
    PC = target
else
    PC = PC+4
```

Conditional Branch Example

```
long int deref(long int *xp)
{
    long int rval = 0;
    if (xp != 0) {
        rval = *xp;
    }
    return rval;
}
```

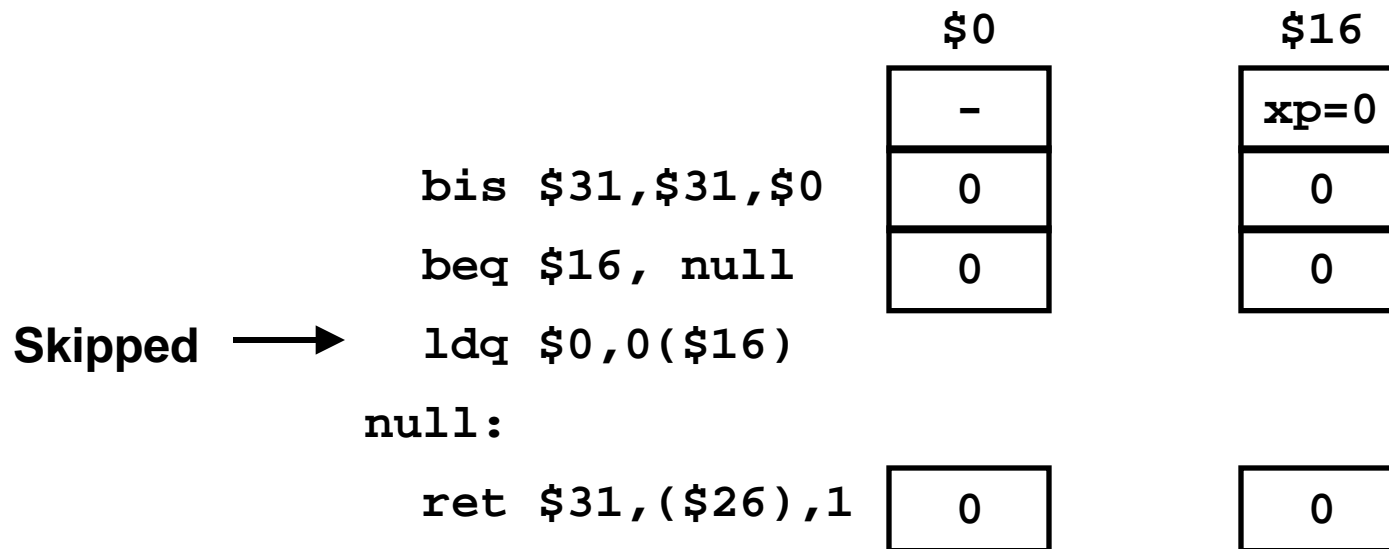
Dereferencing Invalid Pointer

- If address valid, will return whatever is at that address
- If address is invalid, will signal error
 - “Segmentation fault”
 - Attempt to access portion of virtual address space that hasn’t been allocated yet

Executing Deref (0)

```
deref(*xp)
{
  rval = 0;
  if (xp != 0) {
    rval = *xp;
  }
  return rval;
}
```

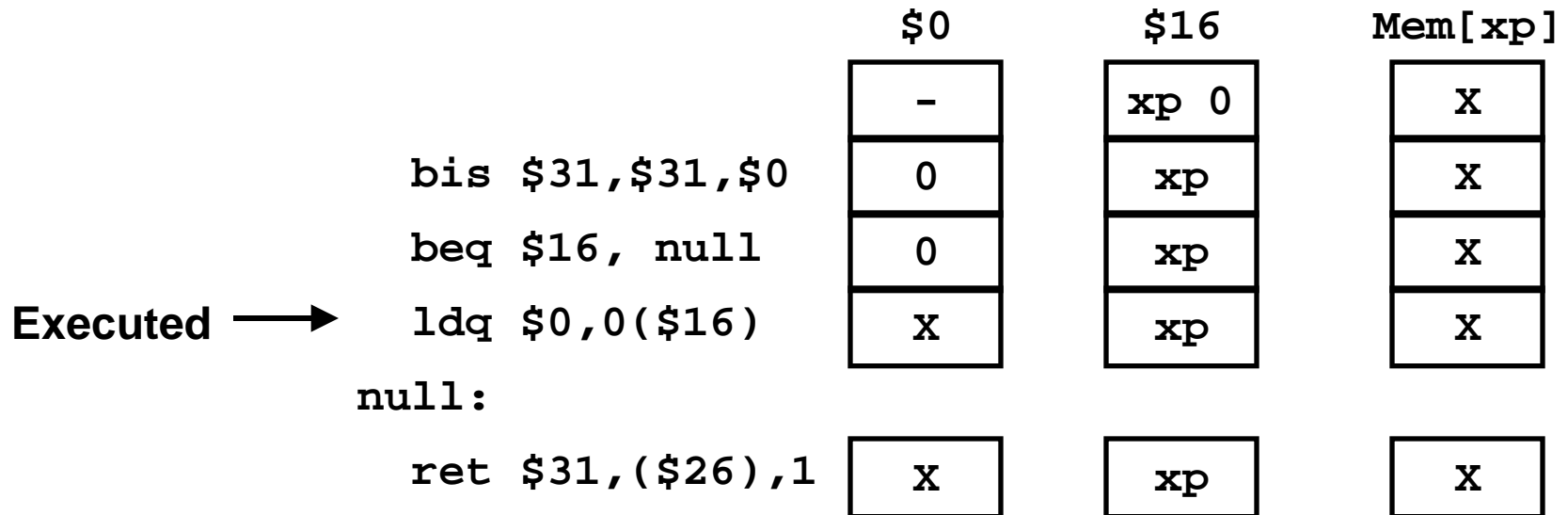
Op	a <op> b
bis	a b
Special Register	
\$31	== 0



Executing Deref (0)

```
deref(*xp)
{
  rval = 0;
  if (xp != 0) {
    rval = *xp;
  }
  return rval;
}
```

```
Op      a <op> b
bis     a | b
Special Register
$31    == 0
```



Conditional Branch Example #2

```
long int absval(long int x)
{
    long int rval = x;
    if (x < 0) {
        rval = -x;
    }
    return rval;
}
```

Assembly

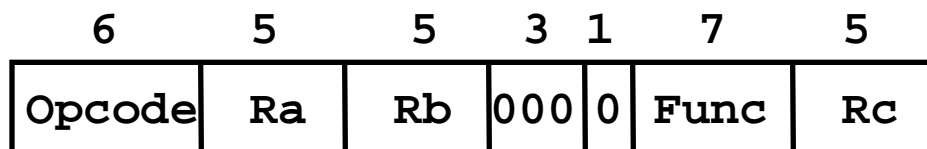
- x in \$16
- rval in \$0
- \$31 always 0

```
subq $31,$16,$0    # rval = -x
blt  $16, neg      # if x < 0 goto neg
bis  $16, $16, $0  # else rval = x
neg:
ret  $31,($26),1   # return
```

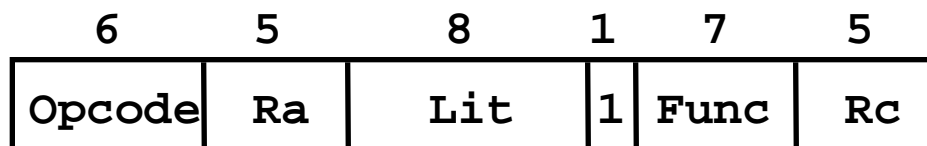
Conditional Move Instructions

Formats

`cmoveCOND Ra, Rb, Rc`



`cmoveCOND Ra, Lit, Rc`



- Same formats as arithmetic instruction
- *COND* describes update condition
- Opcode + Func encode *COND*

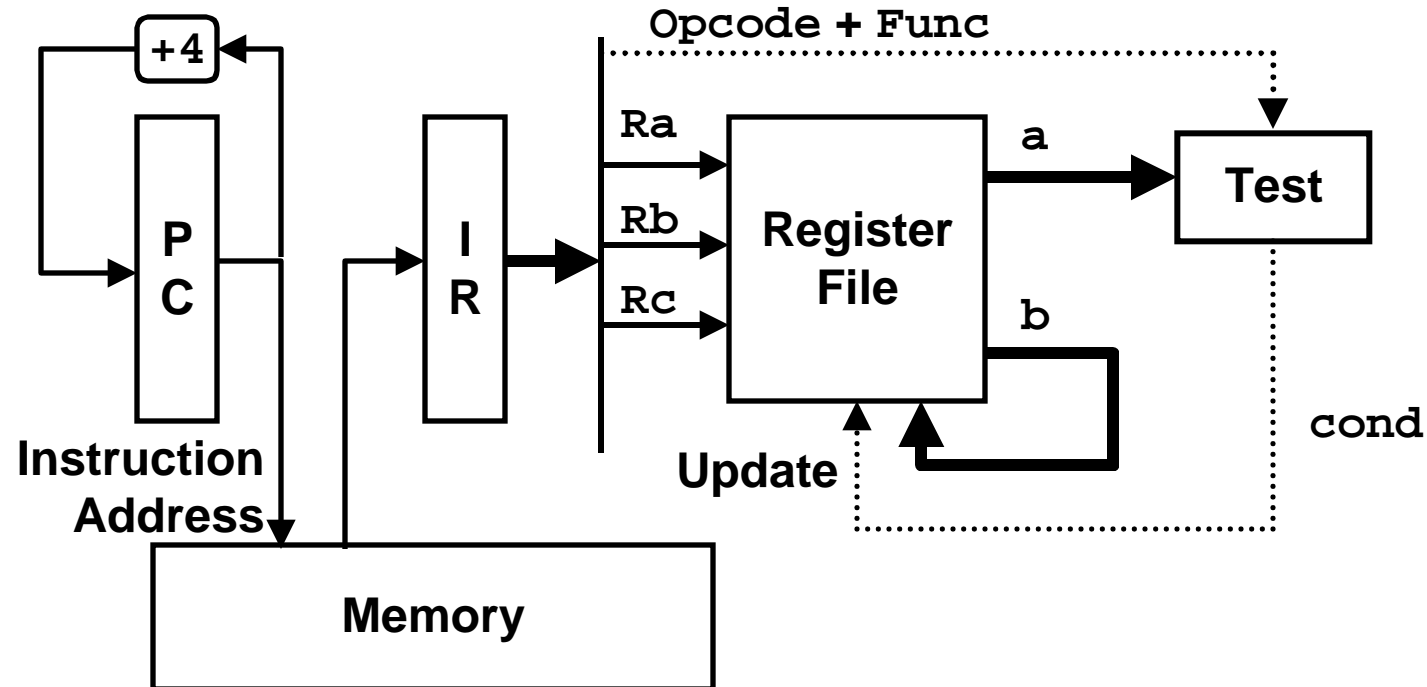
Possible Conditions

<code>eq</code>	<code>a == 0</code>
<code>ne</code>	<code>a != 0</code>
<code>gt</code>	<code>a > 0</code>
<code>ge</code>	<code>a >= 0</code>
<code>lt</code>	<code>a < 0</code>
<code>le</code>	<code>a <= 0</code>
<code>lbc</code>	<code>a&1 == 0</code>
<code>lbs</code>	<code>a&1 != 0</code>

Operation

- Compare `Reg[Ra]` to 0
- if (*cond*) `Reg[Rc] = Reg[Rb]`
- Else register `Rc` unchanged

Implementing Conditional Move



1. Fetch

$IR = Mem[PC]$

$a = Reg[Ra]$

$b = Reg[Rb]$

2. Execute

$cond = Cond(a)$

3. Update

if (cond)

$Reg[Rc] = b$

$PC = PC + 4$

Conditional Move Example

```
long int absval(long int x)
{
    long int rval = x;
    if (x < 0) {
        rval = -x;
    }
    return rval;
}
```

Assembly

- x in \$16
- rval in \$0
- \$31 always 0

```
subq $31,$16,$0    # rval = -x
cmovge $16,$16,$0  # if (x >= 0) rval = x
ret $31,($26),1    # return
```

Why Use Conditional Moves?

- Transfer of control disrupts flow of instructions through pipeline
- Especially when cannot reliably predict test outcome

Jump Instructions

Format

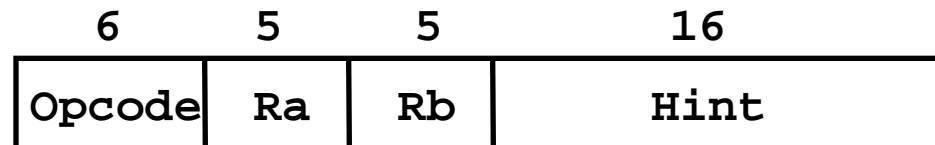
`jmp Ra, (Rb), Hint` **Jump**
`jsr Ra, (Rb), Hint` **Jump to Subroutine**
`ret Ra, (Rb), Hint` **Return from Subroutine**

- **Ra** **Register to store return pointer: PC+4**
 - Usually \$31 for `jmp` and `ret`
 - » Don't store return pointer
 - Usually \$26 for `jsr`
- **Rb** **Jump destination**
 - Usually \$26 for `ret`
- **Hint** **Hint to help predict jump target address**
 - Don't worry about this
- **Assembler allows use of symbolic labels**

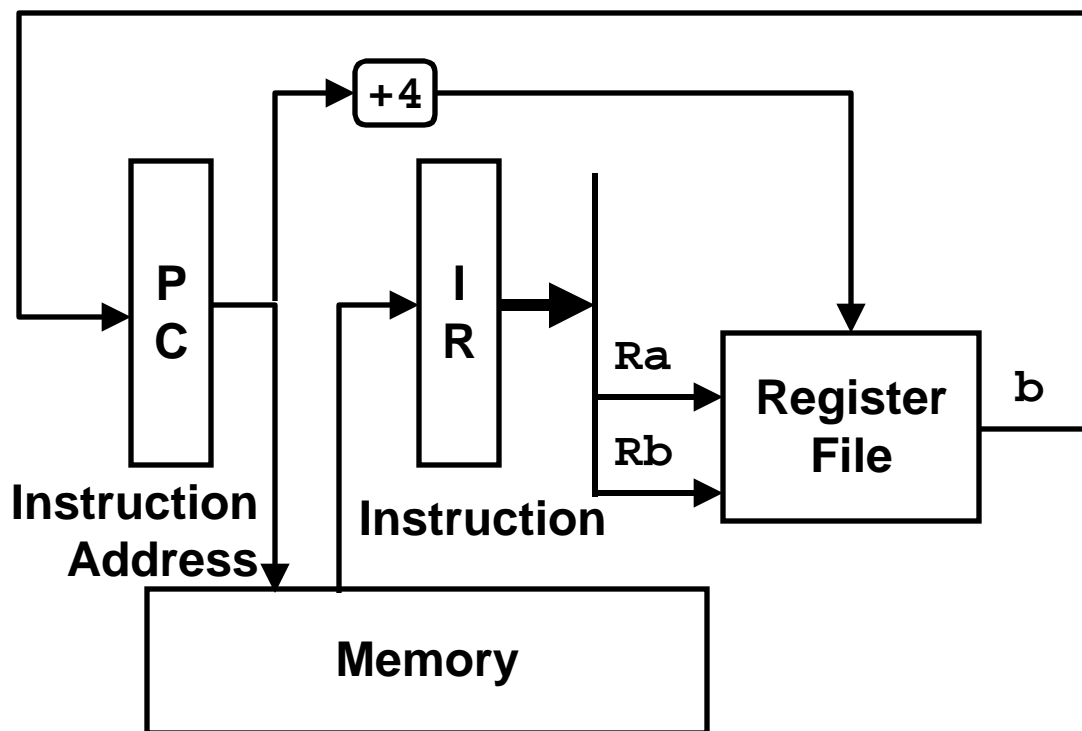
Operation

$\text{Reg}[\text{Ra}] = \text{PC}+4$

$\text{PC} = \text{Reg}[\text{Rb}]$



Implementing Jump Operations



1. Fetch

$IR = Mem[PC]$

$b = Reg[Rb]$

2. Execute

3. Update

$Reg[Ra] = PC+4$

$PC = b$

Procedure Call/Return Example

```
long int abs_deref(long int *xp)
{
    long int x = deref(xp);
    long int rval = absval(x);
    return rval;
}
```

Data Passing Conventions

- Procedure arguments in registers \$16, \$17, ...
- Return result in \$0

Executing Abs_deref

```
abs_deref(*xp)
{
    x = deref(xp);
    rval = absval(x);
    return rval;
}
```

Instruction Addresses

0x080:	abs_deref	call
0x200:	deref	entry
0x300:	absval	entry

	\$0	\$1	\$16	\$26
... # Stack stuff ...	-	-	xp	-
0x100: lda \$1, deref	-	0x200	xp	-
0x104: jsr \$26,(\$1),1	-	-	xp	0x108
# On return:	x	-	-	0x108
0x108: bis \$0,\$0,\$16	x	-	x	0x108
0x10c: lda \$1,absval	x	0x300	x	0x108
0x110: jsr \$26,(\$1),1	-	0x300	x	0x114
# On return:	rval	-	-	0x114
... # Stack stuff ...				
0x11c: ret \$31,(\$26),1	rval	-	-	0x084

Alpha 21164 Block Diagram

Four Caches

- Most recently accessed instructions, data, address translations

Two Integer Pipelines

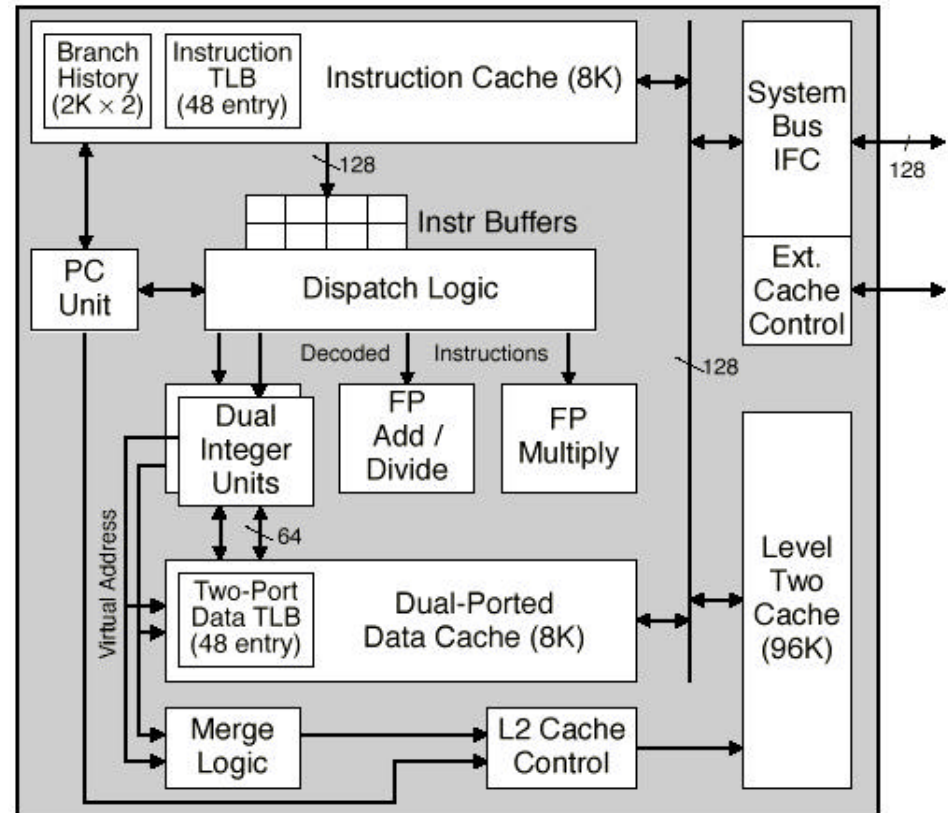
- Perform integer instructions, load/store addresses, branch conditions

Two Floating Point Pipelines

- Floating point operations

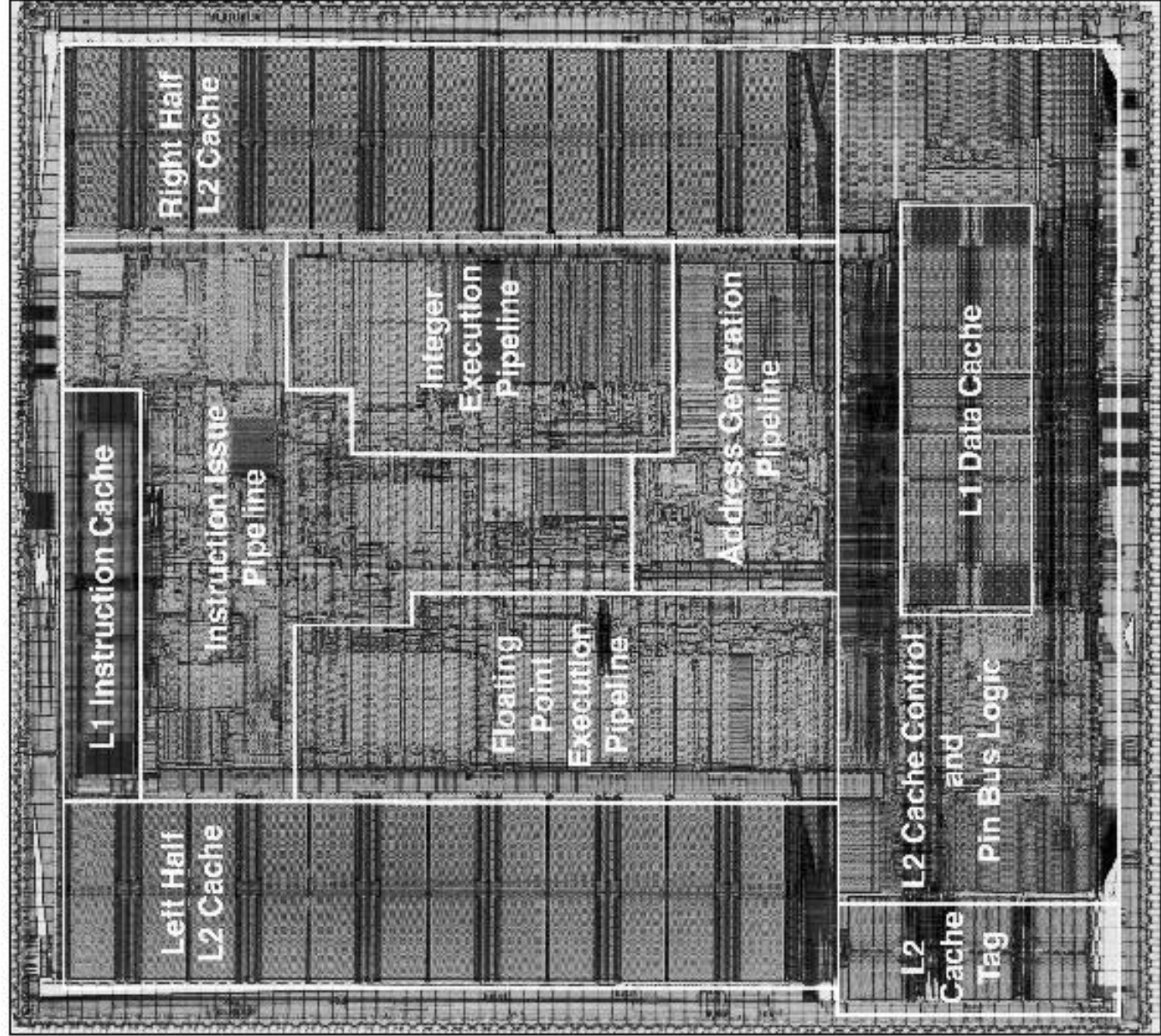
Attempts to Predict Branches

- Whether or not taken
- Target address



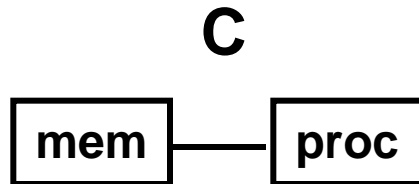
Microprocessor Report '94

21164 Die Photo



Summary: Abstract Machines

Machine Models



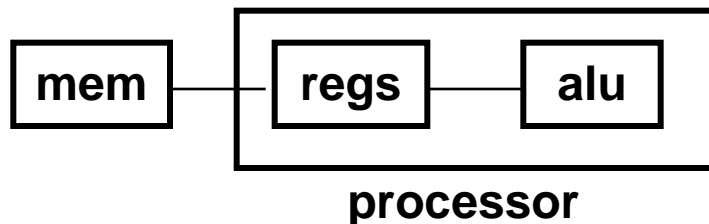
Data

- 1) char
- 2) int, float
- 3) double, long
- 4) struct, array
- 5) pointer

Control

- 1) loops
- 2) conditionals
- 3) goto
- 4) Proc. call
- 5) Proc. return

Assembly

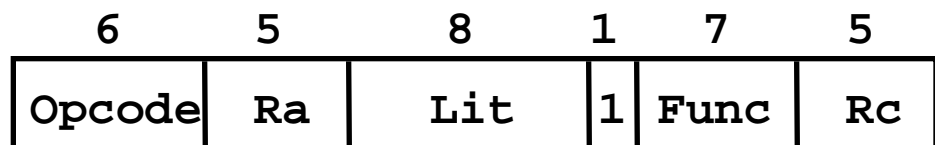
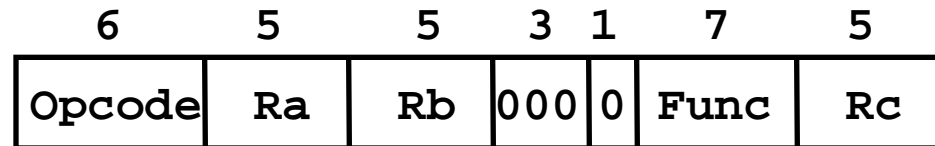


- | | |
|-------------------------------|----------------|
| 1) byte | 3) branch/jump |
| 2) 4-byte long word | 4) jsr |
| 3) 8-byte quad word | 5) ret |
| 4) contiguous word allocation | |
| 5) address of initial byte | |

Summary: Instruction Formats

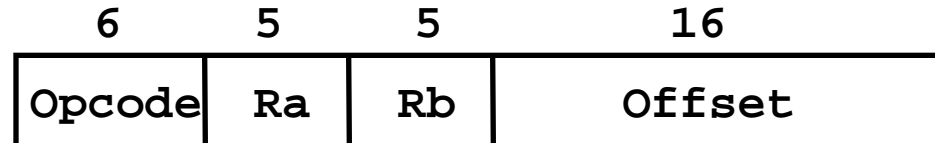
Arithmetic Operations:

- all register operands
 - `addq $1, $7, $5`
- with a literal operand
 - `addq $1, 15, $5`



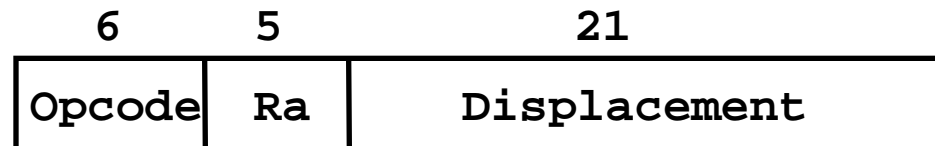
Loads & Stores:

- `ldq $1, 16($30)`



Branches:

- a single source register
 - `bne $1, label`



Jumps:

- one source, one dest reg
 - `jsr $26, ($1), hint`

