EE282
Computer Architecture

Lecture 3:
Instruction-Set Architecture (Part 2)

October 4th, 2001

Marc Tremblay
Stanford University
marctrem@csl.stanford.edu

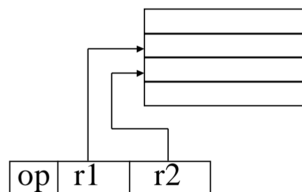
Today

- Instruction-set architecture
  - Register files
  - ISA Design Styles
  - Common instruction types
  - Data Types
  - Addressing Modes
  - Binary Encodings
  - Control Instructions
  - Interrupts & Events

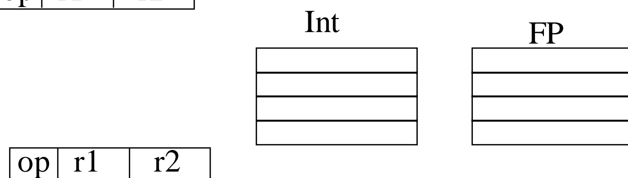
Register Files

- Group of registers having the same characteristics
- Number of registers (e.g. 16, 32, 64, 128, 256 registers)
- Register width (e.g. 32, 64, 80, 128 bits)
- Data type
  - Integer, fixed-point, floating-point, SIMD
  - Data type agnostic -> unified register file
    - FP + SIMD: very common (e.g. IA32, SPARC, PowerPC)
    - Int + FP + SIMD: less common (e.g. MAJC)
- Organization
  - Contiguous/flat
  - Some overlapping in space or time
    - Register windows (RISC-I/II, SPARC)
    - Dribbling stack (IA-64, Java engines)

Register File Organization



- Flat register file
- # of registers = $2^{(\text{\# of reg spec bits})}$
- # of ports is typically # of registers that need to be accessed simultaneously



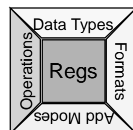
- If op starts with 0x1… -> access integer register file
- If op starts with 0x0… -> access floating-point register file
- Opcode bit effectively doubles name space

Register Files: Implicit functionality

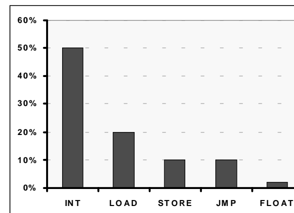
- Some registers often have a special role
- Constant value
  - R0 often is value “0”
  - R1 is sometimes “1”
- Assignment
  - IP may be saved automatically in one specific register upon a function call
- Software convention
  - Registers may be assigned role by software
  - For instance:
    - Stack pointer
    - Returned value
    - Parameters

Principles of Instruction Set Design

- Keep it simple (KIS)
  - complexity
    - increases logic area
    - increases pipe stages
    - increases development time
  - evolution tends to make kludges
- Orthogonality (modularity)
  - simple rules, few exceptions
  - all ops on all registers

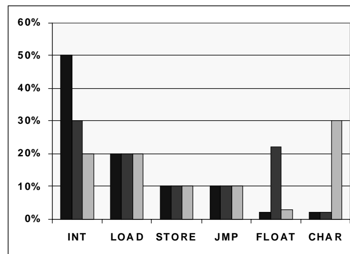


- Frequency
  - make the common case fast
    - some instructions (cases) are more important than others

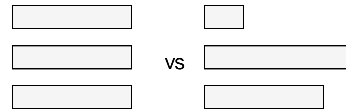


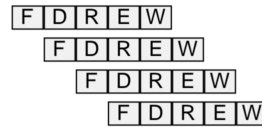
Principles of Instruction Set Design (part 2)

- Generality
  - not all problems need the same features/instructions
  - principle of *least surprise*
  - performance should be easy to predict



- Locality and concurrency
  - design ISA to permit efficient implementation
    - today
    - 10 years from now



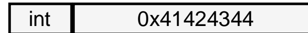


Instruction Types

- Operations
  - arithmetic
  - logical
  - data type conversions
- Data Movement
  - memory reference
  - register to register
- Control
  - what instruction to do next
  - tests (compare)
  - branches and jumps
  - support for procedure call
  - operating system entry
- Misc. Junk

Data Types

- How the contents of memory and registers are interpreted
- Can be identified by
  - tag
  - use
- Driven by application
  - Signal processing
    - 16-bit fixed point (fraction)
    - 32-bit fixed point (fraction)
  - Text processing
    - 8-bit characters
  - Scientific computing
    - 64-bit floating point
  - Multimedia
    - 8-bit vectors of length 3 or 4
- Most *general purpose* computers support several types
  - 8, 16, 32, 64-bit
  - signed and unsigned
  - fixed and floating



Example, 32-bit Floating Point

- Type specifies mapping from bits to real numbers (plus symbols)
  - format
    - S, 8-bit exp, 23-bit mantissa
  - interpretation
    - mapping from bits to abstract set

$^{S} \quad ^{(E-127)}$

  - operations
    - add, mult, sub, sqrt, div

Addressing Modes
Driven by Program Usage

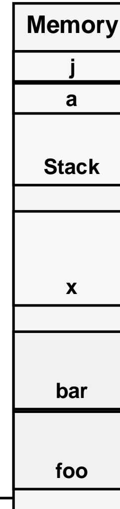
```
double x[100] ;              // global
void foo(int a) {            // argument
  int j ;                    // local
  for(j=0;j<10;j++)
    x[j] = 3 + a*x[j-1] ;
   bar(a);
}
```

procedure

constant

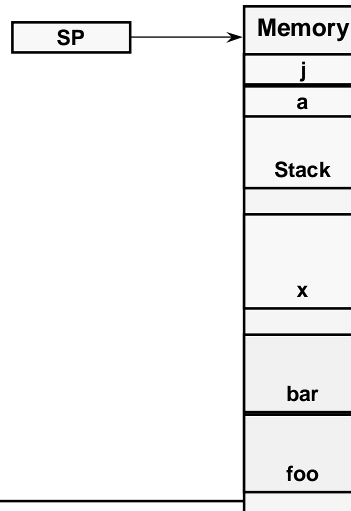
argument

array reference



Addressing Modes

- Stack relative for locals and arguments
  - *(R30+x)
  - a, j
- Short immediates (small constants)
  - 3
- Long immediates (global addressing)
  - &x[0], &bar
  - 0x3ac1e400
- Indexed for array references
  - *(R4+R3)
  - *(R4+R3*S)



VAX-11 Had 27 Addressing Modes (Why?)

`#n`	immediate	`@d(Rn)[Rx]`	
`Rn`	Register	`@#addr[Rx]`	
`(Rn)`	Direct	`(PC)+`	immediate
`-(Rn)`	predecrement	`@(PC)+`	absolute
`(Rn)+`	postincrement	`@d(PC)+`	immediate
`@(Rn)+`	Indirect postincrement		
`d(Rn)`	Displacement (b,w,l)		
`@d(Rn)`			
`(Rn)[Rx]`	Indexed		
`(Rn)+[Rx]`			
`-(Rn)[Rx]`			
`@(Rn)+[Rx]`			
`d(Rn)[Rx]`			

Addressing Mode Summary

1) **Implied addressing - operand is specified within the operation.**

2) **Immediate addressing - Operand data is included in the instruction.**

   `add  r4,#5    r4 ← r4+5`

3) **Register addressing - Operand data is in a register.**

   `add  r4,r5    r4 ← r4+r5`

Memory Addressing

When operands reside in memory - more complex modes may exist.

1) **Direct Addressing.**

```
add  r4,(0x300)     r4 ← r4+M[0x300]
```

2) **Indirect addressing - register contains the operand address.**

```
add  r4,(r5) r4 ← r4+M[r5]
```

3) **Indirect displaced -**

```
add  r4,100(r5)     r4 ← r4+M[r5+100]
```

4) **Indexed -**

```
add  r4,(r5+r6)     r4 ← r4+M[r5+r6]
```

5) **Memory Indirect (or Double Indirect) -**

```
add  r4,@(r5) r4 ← r4+M[M[r5]]
```

WJD/AW/MT

EE282 - Class 3 – 10/04/01

Other Addressing Modes

1) **Autoincrement**

```
add  r4,(r5)+         r4 ← r4+M[r5]   r5 ← r5 + 1
```

2) **Autoincrement (by word size)**

```
add  r4,(r5)+         r4 ← r4+M[r5]   r5 ← r5 + d
```

3) **Autoincrement (by stride)**

```
add  r4,20(r5)        r4 ← r4+M[r5]   r5 ← r5 + 20
```

4) **Autodecrement**

```
add  r4,-(r5)         r5 ← r5 - 1     r4 ← r4+M[r5]
```

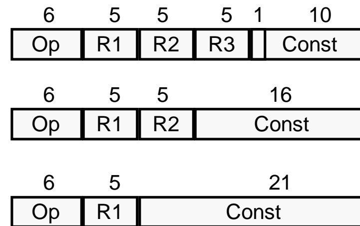
Both Preincrement and Postincrement modes are sometimes available.

WJD/AW/MT

EE282 - Class 3 – 10/04/01

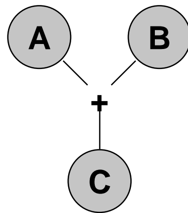
Instruction Formats

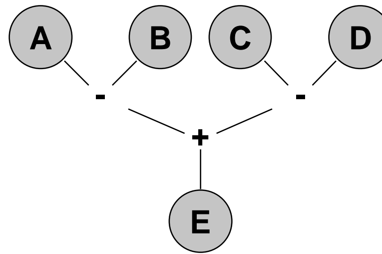
- Different instructions need to specify different information
  - return
  - increment R1
  - R3 ← R1 + R2
  - jump to 64-bit address
- Frequency varies
  - instructions
  - constants
  - registers
- Can encode
  - fixed format
  - small number of formats
  - byte/bit variable



Fixed-Format (Alpha)

Side Effects





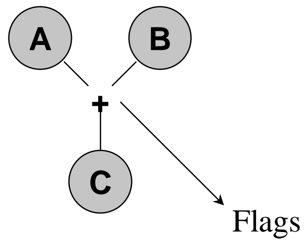

```
C=A+B;

add R3,R1,R2
```

```
C=(A-B)+(C-D);

sub R5,R1,R2
sub R6,R3,R4
add R5,R5,R6
```

Side Effects (2)




```
C=A+B;
add R3,R1,R2
```

- Side Effects are additional state changed by an operation
  - Flags registers
  - Changes to source registers
  - Multiple results
  - Exception conditions
- Disturb clean dataflow models
- Particularly vile when side effects change special-purpose registers in a general-purpose register machine.

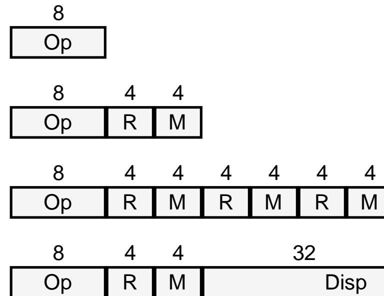
Side Effects (3)

Advantages
- Implied concurrency
- Reduced instruction count
- Reduced program size
- Powerful atomic operations

Disadvantages
- Hard to optimize programs
- Complex compiler structures
- Multiple register writes
  - can limit issue rate
- Special-purpose registers

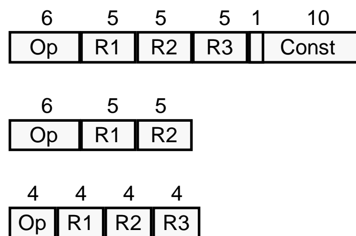
Variable-Length Instructions

- Variable-length instructions give more efficient encodings
  - no bits to represent unused fields/operands
  - can frequency code operations, operands, and addressing modes
  - Examples
    - VAX-11, Intel x86 (byte variable)
    - Intel 432 (bit variable)
- But - can make fast implementation difficult
  - sequential determination of location of each operand



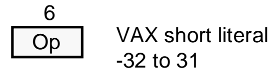
Compromise: A Few Good Formats

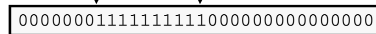
- Gives much better code density than fixed-format
  - important for embedded processors
- Simple to decode



Constant Encoding

- Integer constants
  - mostly small
  - positive or negative
- Bit fields
  - contiguous field of 1s within 32bits (64 bits)
- Other
  - addresses, characters, symbols
- A good architecture
  - uses a few bits to encode the most common.
  - allows any constant to be generated (table reference)





Control Instructions

- Implicit control on each instruction
  - $IP \leftarrow IP + 4$
- Unconditional jumps
  - $IP \leftarrow X$
  - $IP \leftarrow IP + X$
  - X can be constant or register
- Conditional jumps (branches)
  - $IP \leftarrow IP + ((cond)\ ?\ X : 4)$
- Predicated instructions
- Conditions
  - flags
  - in a register
  - fused compare and branch

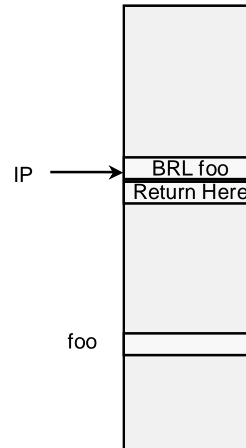
```
LOOP:   LOAD    R1 <- *(R5+R2)
        ADD     R3 <- R3 + R1
        ADD     R2 <- R2 + 4
        CMP     R4 <- R2 == 8
        JNE     R4, LOOP
```

Support for Procedures

- Branch and Link
  - store return address in reg and jump

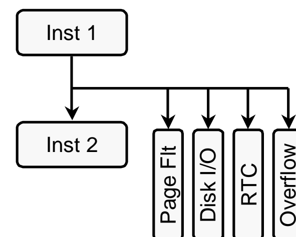
    Rx ← IP + 4

    IP ← Dest
- Subroutine call
  - push return address on stack and jump

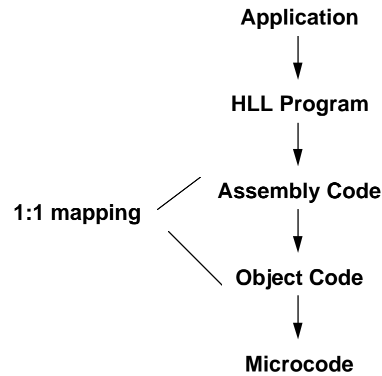


Events

- Implied multi-way branch after every instruction
  - external events (interrupts)
    - completion of I/O operations
  - internal events (faults or exceptions)
    - arithmetic overflow
    - page fault

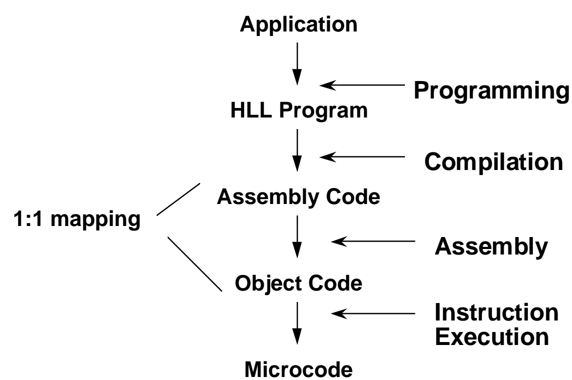


Mapping Applications to Hardware



- **Each mapping translates a more abstract description of what is to be done at execution time to a less abstract description.**

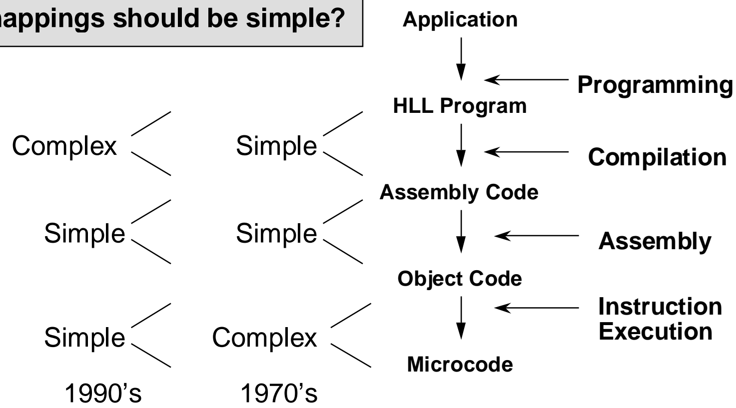
Common Terms for Mappings



Why do so many levels exist?

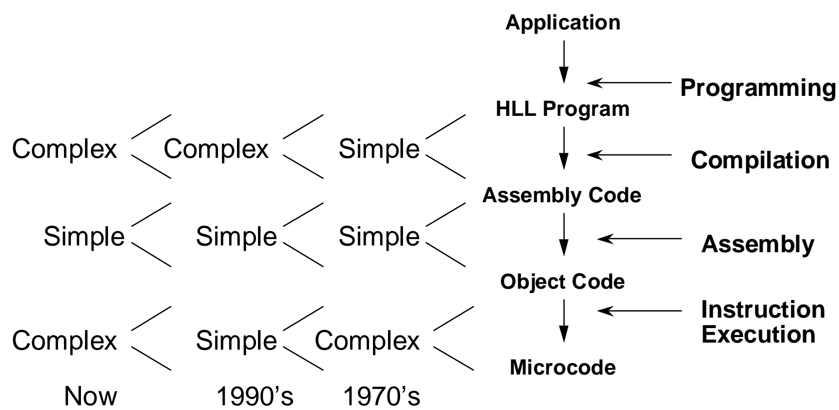
The "Semantic Gap"

Which mappings should be simple?



Complex mappings are opportunities for optimization

The Performance Challenge



Next Time

- Implementation
  - building blocks
  - simple implementation of the DLX