Classifying Instruction Set Architectures

- Using the type of internal storage in the CPU.

<table>
<thead>
<tr>
<th>Internal Storage in CPU</th>
<th>Explicit operands per ALU instruction</th>
<th>Destination for result</th>
<th>Access operand by</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>0 (operand implicit on stack)</td>
<td>Stack</td>
<td>Push or Pop on Stack</td>
<td>B5500 HP3000/70</td>
</tr>
<tr>
<td>Accumulator</td>
<td>1</td>
<td>Accumulator</td>
<td>Load/Store accumulator</td>
<td>Motorola 6809</td>
</tr>
<tr>
<td>General-purpose registers</td>
<td>2, 3</td>
<td>Register or Memory</td>
<td>Load/Store register</td>
<td>360, VAX</td>
</tr>
</tbody>
</table>

- Using the number of explicit operands named per instructions.
- Using operand location.
  Can ALU operands be located in memory?
  —RISC architecture requires all operands in register.
  —Stack architecture requires all operands in stack.
    (top portion of the stack inside CPU; the rest in memory)
- Using operations provided in the ISA.
- Using types and size of operands.
Metrics for Evaluating Different ISA

- Code size (How many bytes in a program’s executable code?)
- Code density (How many instructions in x K Bytes?)
- Instruction length.
  - Stack architecture has the best code density. (No operand in ALU ops)
- Code efficiency. (Are there limitation on operand access?)
  - Stack can not be randomly accessed.
    - e.g. $M_{top-x} \geq 2$ cannot be directly accessed.
    - $M_{top} - M_{top-1}$ is translated into subtract followed by negate
- Bottleneck in operand traffic.
  - Stack will be the bottleneck;
  - both input operands from the stack and result goes back to the stack.
- Memory traffic (How many memory references in a program (i+d)?)
  - Accumulator Arch: Each ALU operation involves one memory reference.
- Easiness in writing compilers.
  - General-Purpose Register Arch: we have more registers to allocate.
    more choices more difficult to write?
- Easiness in writing assembly programs.
  - Stack Arch: you need to use reverse polish expression.
Comparison of Instruction Set Architectures

In the following let us compare five different ISAs.

- Stack Architecture
- Accumulator Architecture
- Load/Store Architecture: GPR(0,3) Arch. — e.g. MIPS R2000, SPARC
  0 → no memory operand allowed in ALU instruction;
  3 → max. 3 operands in ALU instructions;
  GPR stands for General-Purpose Register
- Register-Memory Architecture: GPR(1,2) Arch. — e.g. M68020
- Memory-Memory Architecture: GPR(3,3) Arch. — e.g. VAX

Let us compare how $f = (a-b)/(c-d*e)$ are translated.
Assume all six variables are 4-bytes integers.
Assume opcodes are 1 bytes except push and pop in stack arch. are 2 bits.
  Memory address is 16 bits and register address is 5 bits.
In reality, VAX and M68020 do not have 32 registers.
We will evaluate the instruction density and memory traffic.
See the actual compiled assembly code of GPR Architectures in
~cs520/src/arch.s*.<mips, sun3, vax>
Compute $f=(a-b)/(c-d*e)$
Using Stack Architecture

Assume that top 4 elements of the stack are inside CPU.

Instructions

<table>
<thead>
<tr>
<th>Stack Content</th>
<th>M_{stacktop} ← M_{stacktop-1}</th>
<th>bop M_{stacktop}</th>
</tr>
</thead>
<tbody>
<tr>
<td>push a[a]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>push b[a, b]</td>
<td>binary operator</td>
<td></td>
</tr>
<tr>
<td>subtract[a-b]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>push c[a-b, c]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>push d[a-b, c, d]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>push e[a-b, c, d, e]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mult[a-b, c, d*e]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subtract[a-b, c-d*e]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>divide[a-b/(c-d*e)]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pop f</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction count = 10

Total bits for the instructions = bits for opcodes + bits for addresses

$=(6*2+4*8)+(6*16)=140$ bits

Memory traffic = $140$ bits + $6*4*8$ bits = $332$ bits
Compute $f = (a-b)/(c-d*e)$

Using Accumulator Architecture

Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>AC temp (a memory location)</th>
<th>AC ← AC bop M_p</th>
</tr>
</thead>
<tbody>
<tr>
<td>load d[d]</td>
<td>temp[d]</td>
<td></td>
</tr>
<tr>
<td>mult e[d*e]</td>
<td>temp[d*e]</td>
<td></td>
</tr>
<tr>
<td>store temp</td>
<td>temp[c-d*e]</td>
<td></td>
</tr>
<tr>
<td>load c[c]</td>
<td>temp[d*e]</td>
<td></td>
</tr>
<tr>
<td>sub temp</td>
<td>temp[d*e]</td>
<td></td>
</tr>
<tr>
<td>load a</td>
<td>temp[d*e]</td>
<td></td>
</tr>
<tr>
<td>sub b</td>
<td>temp[d*e]</td>
<td></td>
</tr>
<tr>
<td>div temp</td>
<td>f</td>
<td></td>
</tr>
</tbody>
</table>

Instruction count = 10

Total bits for the instructions = bits for opcodes + bits for addresses

= (10*8) + (10*16) = 240 bits

Memory traffic = 240 bits + 10*4*8 bits = 560 bits
Compute $f=\frac{(a-b)}{(c-d*e)}$

Using Load/Store Architecture: GPR(0,3)

MIPS R2000 instruction $sp$ is a special register called stack pointer.

Instructions $(op \ dst, src1, src2)$ \hspace{1em} $dst \leftarrow src1 \ bop \ src2$ $n$: nth register

- $lw$ $14, 20(sp)$ \hspace{1em} $a$ was allocated at $M[20+sp]$  
- $lw$ $15, 16(sp)$ \hspace{1em} $b$ was allocated at $M[16+sp]$  
- $subu$ $24, 14, 15$ \hspace{1em} $24=a-b$  
- $lw$ $25, 8(sp)$ \hspace{1em} $d$ was allocated at $M[8+sp]$  
- $lw$ $8, 4(sp)$ \hspace{1em} $e$ was allocated at $M[4+sp]$  
- $mul$ $9, 25, 8$ \hspace{1em} $9=de$  
- $lw$ $10, 12(sp)$ \hspace{1em} $c$ was allocated at $M[12+sp]$  
- $subu$ $11, 10, 9$ \hspace{1em} $11=c-d*e$  
- $div$ $12, 24, 11$  
- $sw$ $12, 0(sp)$ \hspace{1em} $f$ was allocated at $M[0+sp]$  

Instruction count = 10

Total bits for the instructions = bits for opcodes + bits for addresses  
\hspace{1em} = (10*8)+(4*(5+5+5)+6*(5+16)) = 266 bits

Memory traffic = 266 bits + 6*4*8 bits = 458 bits
Compute \( f = \frac{a-b}{c-d*e} \)

Using Register-Memory Architecture: GPR(1,2)

**M68020 instructions**  
\( a6 \) is a stack pointer  
\( d0 \) & \( d1 \) are registers

**Instructions (op src, dst)**  
\( d0 \)  
\( d1 \)  
\( dst \leftarrow dst \ bop \ src \)

- \( \text{movl a6@(-16), d0} \)  
  d was allocated at M\[a6-16\]
- \( \text{mulsl a6@(-20), d0} \)  
  \( d*e \)  
  e was allocated at M\[a6-20\]
- \( \text{movl a6@(-12), d1} \)  
  \( c \)  
  c was allocated at M\[a6-12\]
- \( \text{subl d0, d1} \)  
  \( c-d*e \)
- \( \text{movl a6@(-4), d0} \)  
  \( a \)  
  a was allocated at M\[a6-4\]
- \( \text{subl a6@(-8), d0} \)  
  \( a-b \)  
  b was allocated at M\[a6-8\]
- \( \text{divsl d1, d0} \)
- \( \text{movl d0, a6@(-24)} \)  
  \( f \)  
  f was allocated at M\[a6-24\]

**Instruction count=8**

**Total bits for the instructions=bits for opcodes + bits for addresses**

\[ = (8*8) + (6*(5+16)+2*(5+5)) = 210 \text{ bits} \]

**Memory traffic=210 bits+6*4*8 bits = 402 bits**
Compute \( f = \frac{(a-b)}{(c-d*e)} \)

Using Memory-Memory Architecture: GPR(3,3)

VAX instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Source 1</th>
<th>Source 2</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>subl3</td>
<td>-8(fp), -4(fp), r0</td>
<td>[a-b]</td>
<td></td>
<td>r1</td>
</tr>
<tr>
<td>mull3</td>
<td>-20(fp), -16(fp), r1[d*e]</td>
<td></td>
<td>d,e were allocated at M[fp-16],[fp-20]</td>
<td></td>
</tr>
<tr>
<td>subl3</td>
<td>r1,-12(fp), r1</td>
<td></td>
<td></td>
<td>[c-d*e]c was allocated at M[fp-12]</td>
</tr>
<tr>
<td>divl2</td>
<td>r1, r0</td>
<td></td>
<td></td>
<td>[(a-b)/(c-d*e)]</td>
</tr>
<tr>
<td>movl</td>
<td>r0, -24(fp)</td>
<td></td>
<td></td>
<td>f was allocated at M[fp-24]</td>
</tr>
</tbody>
</table>

Instruction count=5

Total bits for the instructions=bits for opcodes + bits for addresses

\[= (5 \times 8) + (6 \times 16 + 7 \times 5) = 171 \text{ bits} \]

Memory traffic=171 bits + 6 \times 4 \times 8 \text{ bits} = 363 \text{ bits}
### Comparison of ISAs
In terms of Instruction Density and Memory Traffic

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Instruction Count</th>
<th>Code Size (bits)</th>
<th>Memory Traffic (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>10</td>
<td>140</td>
<td>332</td>
</tr>
<tr>
<td>Accumulator</td>
<td>10</td>
<td>240</td>
<td>560</td>
</tr>
<tr>
<td>Load-Store</td>
<td>10</td>
<td>266</td>
<td>458</td>
</tr>
<tr>
<td>Reg.-Mem.</td>
<td>8</td>
<td>210</td>
<td>402</td>
</tr>
<tr>
<td>Mem.-Mem.</td>
<td>5</td>
<td>171</td>
<td>363</td>
</tr>
</tbody>
</table>

How about the speed of execution?
Misalignment causes hardware complications

A misalignment memory access results in multiple aligned memory references.

To get the integer in to the register, it not only requires two memory accesses but also requires CPU to perform shift and or operations when receiving data.
Memory Alignment may require more storage

How many bytes will be allocated for the following structure on SPARC and PC? How about on SPARC and PC?

```c
struct PenRecord{
    unsigned char msgType;
    short size;
    int X;
    int Y;
} pr;
```

Assume `pr` is allocated at memory starting at address 2000. After we executed "pr.msgType=8, pr.size=4, pr.X=1, pr.Y=256;", here are the content of memory on different machines in hexadecimal.

<table>
<thead>
<tr>
<th>address</th>
<th></th>
<th>size</th>
<th>X</th>
<th>Y</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>On sparc:</td>
<td>08 00</td>
<td>00</td>
<td>04</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>12 bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>On Pentium:</td>
<td>08 fd</td>
<td>04</td>
<td>00</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>12 bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>On old 86:</td>
<td>08 04</td>
<td>00</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>11 bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Bytes with red color are padding bytes. A structure variable is allocated with memory that is multiple of its largest element size.
Byte Ordering

There are two ways to ordering the bytes of short, int, long data in memory.

- Little endian—(e.g., PC) put the byte with the less significant value in the lower address of the allocated memory. Least significant byte sends/saves first.
- Big endian—(e.g., SPARC) put the byte with more significant value in the lower address of the allocated memory. Most significant byte sends first.

<table>
<thead>
<tr>
<th>address</th>
<th>2000</th>
<th>size</th>
<th>X</th>
<th>Y</th>
<th>2011</th>
</tr>
</thead>
<tbody>
<tr>
<td>On sparc:</td>
<td>08 00 00 04 00 00 01 00 00 01 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

big endian machine: For X, it was allocated at addresses 2004-7. The least significant of the four bytes, 01, is allocated at address 2007 (the higher address).

| On Pentium: | 08 fd 04 00 01 00 00 00 00 01 00 00 00 |

little endian machine: For X, it was allocated at addresses 2004-7. The least significant of the four bytes, 01, is allocated at addresses 2004 (the lower address).
Memory Alignment

What happens if we change the declaration by moving the size field to the end?

```c
struct PenRecord{
    unsigned char msgType;
    int X;
    int Y;
    short size;
}
pr;
```

Assume `pr` is allocated at memory starting at address 2000.

After we executed “`pr.msgType=8, pr.size=4, pr.X=1, pr.Y=256`, here are the content of memory on different machines generated by `write2.c`:

<table>
<thead>
<tr>
<th>address</th>
<th>X</th>
<th>Y</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>08</td>
<td>00</td>
<td>04</td>
</tr>
</tbody>
</table>

On Sparc: 08 00 00 00 00 00 00 01 00 00 01 00 00 04 06 38
16 bytes

On Pentium: 08 00 00 00 01 00 00 00 00 01 00 00 04 00 40 00
12 bytes

On Old 86: 08 01 00 00 00 00 01 00 00 04 00
11 bytes.

Bytes with red color are padding bytes. A structure variable is allocated with memory that is multiple of its largest element size. What if we change `int` to `double`
Memory Alignment Exercise

In ~/cs520/alignment, there is a program, called writer.c, that declares a variable of
the type PenRecord, sets its field values to be msgtype=8, X=1, Y=256, size=4,
and print out sizeof(struct PenRecord), save the variable in binary form in a file,
called datafile. Another program, reader.c, reads in the datafile and print the
values of PenRecord fields. Assume you are in ~/cs520 directory.

a) Run "SPARC/writer" on a SPARC, then run "LPC/reader datafile" on a Linux
PC. Observe difference.
b) Run "SPARC/reader datafile" on a SPARC. Observe differences.

In chow.uccs.edu (a Win95 machine), at c:\cs520,
a) telnet to chow.uccs.edu with login=guest and password=cs520
b) run "reader datafile.spa" and see how data are interpreted.

Character does not require alignment; integer, short, and long require alignment.
On Pentium PC, DEC3100, DECalpha and Sparc, int is interpreted as 4 bytes.
On DECalpha, long is 8 bytes; while on other machines, long is 4 bytes.
DEC3100, DECalpha, has stringent aligned accesses of objects.
SUN3, a 680x0 machine, has less stringent in aligned accesses. Both short and
integer only need to align at even address.
Compilers on Pentium PC with Linux and Win32 generate code that requires
memory alignment. The old 386 and 486 compilers generate code without
memory requirement.
#include <stdio.h> ...

struct PenRecord {
    unsigned char msgType;
    short size;
    int X;
    int Y;};

main() {
    int fd, cc;
    int accessible;
    struct PenRecord r;
    if (access("datafile", F_OK) == 0) {
        /* file with same name exist */
        if ((fd=open("datafile", O_WRONLY, 00777)) <= 0) {
            printf("fd=%d\n", fd);
            exit(1);
        }
    } else if ((fd=open("datafile", O_CREAT, 00777)) <= 0) {
        printf("fd=%d\n", fd);
        exit(1);
    }
    r.msgType=8;
    r.X=1;
    r.Y=256;
    r.size=4;
    printf("size of PenRecord =%d\n", sizeof(struct PenRecord));
    cc=write(fd, &(r.msgType), sizeof(struct PenRecord));
    printf("no.of bytes written is %d\nr.msgType=%d, r.X=%d, r.Y=%d, 
        r.size=%d\n", cc, r.msgType, r.X, r.Y, r.size);
    close(fd);
}
```
#include <stdio.h> ....
struct PenRecord {
    unsigned char msgType;
    short size;
    int X;
    int Y;} r;

main() {
    int fd, cc;

    if ((fd=open("datafile", O_RDONLY, 0)) <= 0) exit(1);
    printf("size of PenRecord = %d\n", sizeof(struct PenRecord));
    cc=read(fd, &r, sizeof(struct PenRecord));
    printf("no.of bytes read is %d
r.msgType=%d, r.X=%d, r.Y=%d, r.size=%d\n",
        cc, r.msgType, r.X, r.Y, r.size);
    close(fd);
}

Run writer on SPARC, we get the following output:
    size of PenRecord = 12
    no.of bytes written is 12
    r.msgType=8, r.X=1, r.Y=256, r.size=4

Run reader on DEC3100 with the datafile generated by writer on SPARC, got:
    size of PenRecord = 12
    no.of bytes read is 12
    r.msgType=8, r.X=16777216, r.Y=65536, , r.size=1024

Run reader on SPARC with the datafile generated by writer on SPARC, got:
    size of PenRecord = 12
    no.of bytes read is 12
    r.msgType=8, r.X=1, r.Y=256, r.size=4
```
Examples of Addressing Modes

VAX instructions

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>$3: i = 3; j = i+4; $3: Immediate address mode</td>
</tr>
<tr>
<td>Displacement</td>
<td>$4: p = &amp;i; $-4(fp): Displacement address mode</td>
</tr>
<tr>
<td>Memory Indirect</td>
<td>*p += 1; *-12(fp): Memory indirect address mode</td>
</tr>
<tr>
<td>Direct</td>
<td>$L16: c = buff[i]; L16: Direct address mode</td>
</tr>
<tr>
<td>Scaled</td>
<td>(r0)[r1]: (r0)[r1]: Scaled address mode</td>
</tr>
<tr>
<td></td>
<td>for (j=1; j&lt;10; j++) { buf[j] = c; }</td>
</tr>
</tbody>
</table>

Similar compiled codes for DEC3100 on ~cs520/src/addrmode.s*
## Design Consideration of Addressing Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct/Immediate</td>
<td>Full Size or Short Literal</td>
<td>ADD R4, #3</td>
</tr>
<tr>
<td>Indirect</td>
<td>Register vs. Memory</td>
<td>ADD R4, (R1)</td>
</tr>
<tr>
<td></td>
<td>as intermediate location</td>
<td>ADD R4, @(R1)</td>
</tr>
<tr>
<td>Indexed</td>
<td>Full Size vs. Short Displacement</td>
<td>LW R12, (R11+R2)</td>
</tr>
<tr>
<td></td>
<td>Single vs. Multiple Index Registers</td>
<td>ADD R1, 100(R2)[R3]</td>
</tr>
<tr>
<td>Self Modifying</td>
<td>Pre- vs. Post-Increment</td>
<td>ADD R1, (R2)+</td>
</tr>
<tr>
<td></td>
<td>Pre- vs. Post-Decrement</td>
<td>ADD R1, -(R2)</td>
</tr>
<tr>
<td></td>
<td>Fixed vs. Data Dependent Increment (Decrement)</td>
<td></td>
</tr>
</tbody>
</table>

Powerful instruction set architectures such as VAX were designed to translate the constructs of programming languages in very few instructions, 
→ providing high level hardware support for languages.

But they also made the design of those processors very difficult and caused the delay of the delivery.
→ In early 1980s, the direction of computer architecture starts the shift to a simpler architecture and RISC was born.
Trade-off in Encoding Address Modes

1. The desire to have as many registers and addressing modes as possible.
2. The impact of the size of the register and addressing mode fields on the average instruction size and hence on the average program size.
3. A desire to have instructions encode into lengths that will be easy to handle in the implementation, e.g. instructions in multiples of bytes, rather than arbitrary length. Many architects have chosen to use a fixed-length instruction.
   → Most RISC architectures use 32-bit fixed-length instruction.
## Operations in the Instruction Set

<table>
<thead>
<tr>
<th>Operator type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic and Logical</td>
<td>Integer arithmetic and logical operations: add, and, subtract, or</td>
</tr>
<tr>
<td>Data Transfer</td>
<td>Load/Store (move instructions on machines with memory addressing)</td>
</tr>
<tr>
<td>Control</td>
<td>Branch, jump, procedure call and return, traps</td>
</tr>
<tr>
<td>System</td>
<td>Operating system call, virtual memory management instructions</td>
</tr>
<tr>
<td>Floating point</td>
<td>Floating-point operations: add, multiply</td>
</tr>
<tr>
<td>Decimal</td>
<td>Decimal add, decimal multiply, decimal-to-character conversions</td>
</tr>
<tr>
<td>String</td>
<td>String move, string compare, string search</td>
</tr>
</tbody>
</table>
Statistics about Control Flow Change

- Branches every 6 to 20 instructions
- Procedure calls every 70 to 300 instructions
- Branches 12 to 14 times more frequent than procedure calls.
- 0 is the most frequent used immediate value in compared (83%).
- Most backward-going branches are loop branches
- Loop branches are taken with 90% probability
- Branch behavior is application dependent and sometimes compiler-dependent.
Procedure Call/Return

Two basic approaches to save registers in procedure calls:
- Caller-saving
- Callee-saving

Where to save registers in procedure calls:
- Memory area pointed by the stack pointer
- Register Windows in Sparc architecture
ISA and Compiler Design

Compiler Writer’s goals:
- Generate correct code for valid program.
- Speed of compiled code.
- Fast compilation, debugging support, interoperability among different languages.

Recent compiler Structure

Dependencies

- Language dependent; machine independent
- Highly machine dependent; language independent

Function

- Transform language to common intermediate form
- Procedure inlining and loop transformation
- Global+local optimization register allocation
- Detailed instruction selection and machine-dependent optimization; may include or be followed by assembler
Procedure Inlining/Integration

```c
int gv;
int
addn(int src, int n) {
    return src+n;
}
main() {
    int i, j;
    gv = 1;
    i = 3;
    j = addn(gv, i); /* replace this statement with j=gv+i */
    printf("j=%d\n", j);
}
```

- Avoid the overhead of procedure call/return by replacing the procedure call with the procedure body (with proper variable substitution).
- Trade-off between the size of the procedure body and the frequencies of the procedure call. The code size of expanded program vs. call/return overhead.
- Some modern programming languages such as C++ have explicit `inline` language construct.
Common Subexpression Elimination

# 2  int A[128], B[128];
# 3  int i, j, k;
# 4  B[i+j*k] = A[i+j*k];

lw $14, 8($sp) # load j
lw $15, 4($sp) # load k
mul $24, $14, $15 # $24=j*k
lw $25, 12($sp) # load i
addu $8, $25, $24 # $8 = i+j*k
mul $9, $8, 4 # each integer is 4 bytes,
addu $10, $sp, 1040
addu $11, $9, $10 # add the base address of the stack
lw $12, -512($11) # -512 = the offset from base address to A[0]
sw $12, -1024($11) # -1024 = the offset from base address to B[0]

- Note that the common subexpression, i+j*k, is only calculated once.
- This is MIPS code compiled on DEC3100 using cc -S -O1 commsubexp.c
Constant Propagation

```
# 2 int i, j, k, l;
# 3
# 4 i = 3;
li $14, 3
sw $14, 12($sp)
# 5 j = i+4;
li $15, 7 # i+4 is optimized to constant 7
sw $15, 8($sp)
# 6 k = i;
li $24, 3
sw $24, 4($sp)
# 7 l = k; # const 3 is propagated to k
li $25, 3
sw $25, 0($sp)
```

- Note that the constant propagation will be stopped when the value of the variable is replaced by the result of a non-constant expression.
Code Motion

```c
void codemotion(int k, int j)
{
    int i, base;
    int A[128];
    for (i=1; i<10; i++) {
        base = k*j;
        A[base+i] = i;
    }
}
```

M68020 code

```assembly
#PROLOGUE# 0
link a6,#-524
moveml #192,sp@

#PROLOGUE# 1
moveq #1,d6
movl a6@(8),d0
mulsl a6@(12),d0
moveq #4,d7
asll #2,d0
movl d0,a0
addl d7,a0
lea a6@(-512,a0:l),a0

L77003:
movl d6,a0@+
addql #1,d6
addql #4,d7
moveq #40,d1
cmpl d1,d7
jlt L77003

#PROLOGUE# 2
moveml a6@(-524),#192
unlk a6

#PROLOGUE# 3
rts
```

- `k* j` is moved out of the loop
- Auto-increment addressing mode
```c
int fib(int n) {
    if (n<0) return -1;
    if (n==0) return 0;
    if (n==1) return 1;
    return fib(n-1)+fib(n-2);
}
main() {
    fib(10);
}
```

### Register Saving in Procedure Call/Return

#### fib:

- `subu $sp, 40`
- `sw $31, 28($sp)`
- `sw $4, 40($sp)`
- `sw $17, 24($sp)`
- `sw $16, 20($sp)`

#### Return ADDR:

- `$31` (SP old)
- `$16` (SP new)
- `$12` (SP new)
- `$8` (SP new)
- `$4` (SP new)
- `$0` (SP old)

#### Temporary Variables:

- `$4$SP new` + 40
- `$31$SP new` + 36
- `$17$SP new` + 32
- `$16$SP new` + 28
- `$12$SP new` + 24
- `$8$SP new` + 20
- `$4$SP new` + 16
- `$0$SP new` + 12
- `$4$SP old` + 8
- `$0$SP old` + 4
- `$0$SP old` + 0

#### jal:

- `jal: save PC+4 (next instruction address) to $31, them jump to the target address`

#### Callee-Saving:

- $6$ if (n<0) return -1;
- `lw $14, 40($sp)`
- `bge $14, 0, $32`
- `li $2, -1`
- `b $35`

#### $32:

- # 7 if (n==0) return 0;
- `lw $15, 40($sp)`
- `bne $15, 0, $33`
- `move $2, $0`
- `b $35`

#### $33:

- # 8 if (n==1) return 1;
- `lw $24, 40($sp)`
- `bne $24, 1, $34`
- `li $2, 1`
- `b $35`

#### $34:

- # 9 return fib(n-1)+fib(n-2);
- `lw $4, 40($sp)`
- `addu $4, $4, -1`
- `jal fib`
- `move $16, $2`
- `lw $4, 40($sp)`
- `addu $4, $4, -2`
- `jal fib`
- `move $17, $2`
- `addu $2, $16, $17`
- `lw $16, 20($sp)`
- `lw $17, 24($sp)`
- `lw $31, 28($sp)`
- `addu $sp, 40`
- `j $31`
Performance effect of various levels of optimization

<table>
<thead>
<tr>
<th>Optimizations performed</th>
<th>Percent faster</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedure integration only</td>
<td>10%</td>
</tr>
<tr>
<td>Local optimizations only</td>
<td>5%</td>
</tr>
<tr>
<td>Local optimizations + register allocations</td>
<td>26%</td>
</tr>
<tr>
<td>Global and local optimizations</td>
<td>14%</td>
</tr>
<tr>
<td>Local and global optimizations + register allocation</td>
<td>63%</td>
</tr>
<tr>
<td>Local and global optimizations + procedure integration + register allocation</td>
<td>81%</td>
</tr>
</tbody>
</table>

measurements from Chow[1983] for 12 small FORTRAN and PASCAL programs.
Impact of Compiler Technology on ISA

How are variables allocated and addressed? How many registers are needed to allocate variable appropriately?

The three areas in which high level languages allocate their data:

- The stack (not the “stack” in stack architecture for evaluating expression)—is an area of the memory that are used to allocate local variables.
  —Objects on the stack are addressed relative to the stack pointer. (using displacement addressing mode)
  —The stack is used for activation records.
  —The stack is grown on procedure call, and shrunk on procedure return.

- The global data area—used to allocate statically declared objects.

- The heap—used to allocated dynamic objects, that do not adhere to a stack discipline. These objects will still exist after the procedure which creates them disappear.
  —These dynamic objects are created using malloc or new procedures and released using free or delete procedure calls. These procedures implement memory management.
  —Objects in the heap are accessed with pointers (in the form of memory addresses).
Reducing Memory References Using Registers

- Register Allocation is more effective for stack-allocated objects than for global variables.
- *Aliased variables* (there are multiple ways to reference such variables) can not be register-allocated. Is it possible to allocate variable a in a register?
  
  ```
  p = &a;
  a = 2;
  *p = 3;  // How to compile this statement?
  i = a + 4;
  ```

- p typically contains a memory address, not a register address.
  
  ```
  *p = 3;  // can be implemented as ADDI R1, R0, #3; LW R2, p; SW 0(R2), R1
  ```

- After register allocation, the remaining memory traffic consists of the following five types:
  1. Unallocated reference—potential register-allocable references
  2. Global scalar
  3. Save/restore memory reference
  4. A required stack reference—due to aliasing, or caller-saved variables
  5. A computed reference—such as heap references, or reference to a stack variable via a pointer or array index.
How the Architecture Can Help Compiler Writers

- Regularity—make operations, data types, and addressing mode orthogonal. This helps simplify code generation. Try not to restrict a register with a certain class of operations.
- Provide primitives not solutions—e.g. the overloaded CALLS instructions in VAX. (It tries to do too much.)
  1. Align the stack if needed.
  2. Push the argument count on the stack.
  3. Save the registers indicated by the procedure call mask on the stack.
  4. Push the return address on the stack, then push the top and base of the stack pointers for the activation record.
  5. Clear the condition codes
  6. Push a word for status information and a zero word on the stack.
  7. Update the two stack pointers.
  8. Branch to the first instruction of the procedure.
- Simplify trade-offs (decision making) among alternative instruction sequences.
Hardware Description Language

Extend C language for explaining functions of the instructions.

- $\leftarrow_n$ means transfer an $n$-bit quantity to the left-hand side object.
- Adopt the big endian notation. A subscript indicates the selection of bit(s).
  - $R3_{24..31}$ represents the least significant byte of $R3$.
- A superscript represents replication of data. $0^{24}$ represents 24 zero bits.
- Variable $M$ used as an array representing main memory. The array is indexed with byte address.
- ## is used to concatenate two fields.

Example: In DLX,

- LB R1, 40(R3) “load byte” means $R1 \leftarrow _{32} (M[40+R3]_0)^{24}##M[40+R3]$
- SB 41(R3), R2 “store byte” means $M[41+R3] \leftarrow _8 R2_{24..31}$

Exercise: Interpret the following description of instructions:

1. $R1 \leftarrow _{32} (M[40+R3]_0)^{16}##M[40+R3]##M[41+R3]$
2. $R1 \leftarrow _{32} 0^{24}##M[40+R3]$
3. $M[502+R2] \leftarrow _{16} R3_{16..31}$
## Hardware Description Language

### Assembly Instruction

**CVTBBW**  
**R10, (R8)**

### Meaning

- **R10**[^1]_{16..31} ← **16** (M[R8])[^2]_{8}  
  - **M[R8]** is the memory address to be accessed.  
  - The replicate count is **8**.  
  - The range of bits is **16**, which means the operation affects bits **0** to **15** of the memory content.  
  - The **no. of bits transferred** is **16**.  
  - The content is **Big Endian**.  

### Registers

- **R15**
- **R10**
- **R8**
- **R3**
- **R1**
- **R0**

### Memory

- **0**  
- **7**  
- **15**  
- **23**  
- **31**

- **1000**  
- **0x8F**

[^1]: **R10**
[^2]: **M[R8]**
DLX Architecture

32 32-bit GPRs, R0, R1, ..., R31. 
R0 always 0.

32 32-bit FPRs (Floating Point Registers), F0, F1, ..., F31. Each of them can store a single precision floating point value.
The register pairs, (F0, F1), (F2, F3), ..., (F30, F31), serve as double precision FPRs. They are named, F0, F2, F30.
They are instructions for moving or converting values between FPRs and GPRs.

Data types include 8-bit bytes, 16-bit half word, 32-bit word for integer
32-bit single precision and 64-bit double precision floating point.

Only immediate and displacement address modes exist.

Byte addressing, Big Endian, with 32 bit address.
A load-store architecture.
## DLX Instruction Format

### I - type instruction

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>5</th>
<th>5</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Encodes: Loads and stores of bytes, words, half words
- All immediates (rd \(\leftarrow\) rs1 op immediate)
- Conditional branch instructions (rs1 is register, rd unused)
  - Jump register, jump and link register
    \(\text{rd} = 0, \text{rs} = \text{destination, immediate} = 0\)

### R - type instruction

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>func</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Register–register ALU operations: \(\text{rd} \leftarrow \text{rs1 func rs2}\)
- Function encodes the data path operation: Add, Sub, ...
- Read/write special registers and moves

### J - type instruction

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset added to PC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Jump and jump and link
- Trap and return from exception
### Load and Store Instructions

<table>
<thead>
<tr>
<th>Example instruction</th>
<th>Instruction name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R1,30(R2)</td>
<td>Load word</td>
<td>$\text{Regs}[R1] \leftarrow 32 \text{ Mem}[30+\text{Regs}[R2]]$</td>
</tr>
<tr>
<td>LW R1,1000(R0)</td>
<td>Load word</td>
<td>$\text{Regs}[R1] \leftarrow 32 \text{ Mem}[1000+0]$</td>
</tr>
<tr>
<td>LB R1,40(R3)</td>
<td>Load byte</td>
<td>$\text{Regs}[R1] \leftarrow 32 (\text{Mem}[40+\text{Regs}[R3]]_0)^{24} ## \text{Mem}[40+\text{Regs}[R3]]$</td>
</tr>
<tr>
<td>LBU R1,40(R3)</td>
<td>Load byte unsigned</td>
<td>$\text{Regs}[R1] \leftarrow 32 0^{24} ## \text{Mem}[40+\text{Regs}[R3]]$</td>
</tr>
<tr>
<td>LH R1,40(R3)</td>
<td>Load half word</td>
<td>$\text{Regs}[R1] \leftarrow 32 (\text{Mem}[40+\text{Regs}[R3]]_0)^{16} ## \text{Mem}[40+\text{Regs}[R3]]##\text{Mem}[41+\text{Regs}[R3]]$</td>
</tr>
<tr>
<td>LF F0,50(R3)</td>
<td>Load float</td>
<td>$\text{Regs}[F0] \leftarrow 32 \text{ Mem}[50+\text{Regs}[R3]]$</td>
</tr>
<tr>
<td>LD F0,50(R2)</td>
<td>Load double</td>
<td>$\text{Regs}[F0]##\text{Regs}[F1] \leftarrow 64 \text{ Mem}[50+\text{Regs}[R2]]$</td>
</tr>
<tr>
<td>SW 500(R4),R3</td>
<td>Store word</td>
<td>$\text{Mem}[500+\text{Regs}[R4]] \leftarrow 32 \text{ Regs}[R3]$</td>
</tr>
<tr>
<td>SF 40(R3),F0</td>
<td>Store float</td>
<td>$\text{Mem}[40+\text{Regs}[R3]] \leftarrow 32 \text{ Regs}[F0]$</td>
</tr>
<tr>
<td>SD 40(R3),F0</td>
<td>Store double</td>
<td>$\text{Mem}[40+\text{Regs}[R3]] \leftarrow 32 \text{ Regs}[F0] ; \text{Mem}[44+\text{Regs}[R3]] \leftarrow 32 \text{ Regs}[F1]$</td>
</tr>
<tr>
<td>SH 502(R2),R3</td>
<td>Store half</td>
<td>$\text{Mem}[502+\text{Regs}[R2]] \leftarrow 16 \text{ Regs}[R31]_{16..31}$</td>
</tr>
<tr>
<td>SB 41(R3),R2</td>
<td>Store byte</td>
<td>$\text{Mem}[41+\text{Regs}[R3]] \leftarrow 8 \text{ Regs}[R2]_{24..31}$</td>
</tr>
</tbody>
</table>
## DLX ALU instructions

<table>
<thead>
<tr>
<th>Example instruction</th>
<th>Instruction name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD $R1, R2, R3</td>
<td>Add</td>
<td>$\text{Regs}[R1] \leftarrow \text{Regs}[R2] + \text{Regs}[R3]$</td>
</tr>
<tr>
<td>ADDI $R1, R2, #3</td>
<td>Add immediate</td>
<td>$\text{Regs}[R1] \leftarrow \text{Regs}[R2] + 3$</td>
</tr>
<tr>
<td>LHI $R1, #42</td>
<td>Load high immediate</td>
<td>$\text{Regs}[R1] \leftarrow 42#0^{16}$</td>
</tr>
<tr>
<td>SLLI $R1, R2, #5</td>
<td>Shift left logical immediate</td>
<td>$\text{Regs}[R1] \leftarrow \text{Regs}[R2] \ll 5$</td>
</tr>
</tbody>
</table>
| SLT $R1, R2, R3     | Set less than                     | if (\text{Regs}[R2] < \text{Regs}[R3])
                                      | \quad \text{Regs}[R1] \leftarrow 0 \\ 
                                      | else \quad \text{Regs}[R1] \leftarrow 0       |

**FIGURE 2.23** Examples of arithmetic/logical instructions on DLX, both with and without immediates.
### DLX Control Flow Instructions

<table>
<thead>
<tr>
<th>Example instruction</th>
<th>Instruction name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>J. name</td>
<td>Jump</td>
<td>PC ← name; ((\text{PC}+4)-2^{25}) ≤ name &lt; ((\text{PC}+4)+2^{25})</td>
</tr>
<tr>
<td>JAL name</td>
<td>Jump and link</td>
<td>R31 ← PC + 4; PC ← name; ((\text{PC}+4)-2^{25}) ≤ name &lt; ((\text{PC}+4)+2^{25})</td>
</tr>
<tr>
<td>JALR R2</td>
<td>Jump and link register</td>
<td>Regs[R31] ← PC + 4; PC ← Regs[R2]</td>
</tr>
<tr>
<td>JR R3</td>
<td>Jump register</td>
<td>PC ← Regs[R3]</td>
</tr>
<tr>
<td>BEQZ R4, name</td>
<td>Branch equal zero</td>
<td>if (Regs[R4] == 0) PC ← name; ((\text{PC}+4)-2^{15}) ≤ name &lt; ((\text{PC}+4)+2^{15})</td>
</tr>
<tr>
<td>BNEZ R4, name</td>
<td>Branch not equal zero</td>
<td>if (Regs[R4] != 0) PC ← name; ((\text{PC}+4)-2^{15}) ≤ name &lt; ((\text{PC}+4)+2^{15})</td>
</tr>
</tbody>
</table>

**FIGURE 2.24 Typical control-flow instructions in DLX.** All control instructions, except jumps to an address in a register, are PC-relative. If the register operand is R0, BEQZ will always branch, but the compiler will usually prefer to use a jump with a longer offset over this “unconditional branch.”

J and JAL use 26 bit signed offset added to the PC+4 (the instruction after J/JAL). The other use registers (32 bits) for destination addresses.
1. Byte ordering and memory alignment:
   struct CircleRecord {
       unsigned char size;
       short X;
       int time;
   } cr;

   a) How many bytes will be allocated for the following cr structure on PC?
      Ans: 8 bytes. There will be a padding byte after size, since X field needs to align with an even address. The address of the available memory happens to be multiple of 4 when we allocation time field, a 4 byte integer.

   b) How about on SUN SPARC?
      Ans: same 8 bytes.

   c) Show the content of memory area allocated to cr, generated by a program compiled on SPARC with cr.size=8; cr.X=3; cr.time=256. Assume cr was allocated with the memory starting at address 2000. Use hexadecimal to represent the byte content and follow the same format in pages 11-13 of the handout.

   d) If the same data was read for cr by a program compiled on PC, what will be the decimal values of cr.size, cr.X, cr.time? Assume the padding characters for the structure are all zeros.
2. For the following DLX code,

```
ADDI R1, R0, #1 ;; keep the value of i in register R1
LW R2, 1500(R0) ;; keep the value of C in R2
ADDI R3, R0, #100 ;; keep the loop count, 200, in R3
```

```
L1: SGT R4, R1, R3
   BNEZ R4, L2
   SLLI R5, R1, #2 ;; multiply i by 4
   LW R6, 5000(R5) ;; calculate address of B[i]
   ADD R6, R6, R2 ;; B[i]+C
   SW 0(R5), R6 ;; A[i]=B[i]+C
   ADDI R1, R1, #1 ;; i++
   J L1
```

```
L2: SW 2000(R0), R1 ;; save final value of i to memory
```

(a) What is the total number of memory references (including data references and instruction references)?

(b) How many instructions are dynamically executed (including those outside of the loop)?
Solution to hw#5

1. Byte ordering and memory alignment:
   a) How many bytes will be allocated for the following cr structure on DEC3100?
      
      ```c
      struct CircleRecord {
          unsigned char size;
          short X;
          short Y;}
      cr;
      ```
      
      Ans: 6 bytes with one byte padding between the size field and the X field.
   b) How about on SUN SPARC?
      Ans: 6 bytes.
   c) How about on PC?
      Ans: 5 bytes.
   d) Show the sequence of bytes (according to the byte ordering of DEC3100) w.r.t. cr generated by a program compiled on DEC3100 with cr.size=8; cr.X=3, cr.Y=256. (See page 11 format)
      
      Ans:

      ![Diagram showing byte ordering on DEC3100 and PC]
e) If the same data was read for cr by a program compiled on PC, what will be the values of cr.size, cr.X, cr.Y? Assume the padding characters for the structure are all zeros

**Ans:** Note that only five bytes will be read in as indicated in the figure above.

- **cr.size =** 8; 2001 and 2002 will be interpreted as cr.X = 3*256 = 768.
- 2003 and 2004 will be interpreted as cr.Y = 0.

2. For the following DLX code,

```
ADDI R1, R0, #1 ;; keep the value of i in register R1
LW R2, 1500(R0) ;; keep the value of C in R2
ADDI R3, R0, #100 ;; keep the loop count, 100, in R3
L1: SGT R4, R1, R3
    BNEZ R4, L2
    SLLI R5, R1, #2 ;; multiply i by 4
    LW R6, 5000(R5) ;; calculate address of B[i]
    ADD R6, R6, R2 ;; B[i]+C
    SW 0(R5), R6 ;; A[i]=B[i]+C
    ADDI R1, R1, #1 ;; i++
    J L1
L2: SW 2000(R0), R1 ;; save final value of i to memory
```

a) What is the total number of memory references?

**Ans:** The loop between SGT and J instructions contains 8 instructions.
The loop will be executed 100 times. Therefore 800 instructions will be executed in this loop.
There are 3 instructions before L2. SGT, BNEZ, and SW will be executed as the last three instructions. Totally, there are 800+3+3=806 instruction references.

There are 2*100 data references inside the loop. There are 2 data reference outside the loop. Totally, there are 202 data references.
Therefore, the total number of memory references is 1008. Since the data and instructions are all 32 bits. There are 32*1008 bits referenced.

b) How many instructions are dynamically executed?
Ans: There are 806 instructions dynamically executed.
Solution to Homework #5

1. a) 6 bytes. b) 6 bytes. c) 
   
   d) cr.size = 8, cr.X=768, cr.Y=1. 
   Note that if the larger field in the struct declaration is a 8byte double precision field, the struct will be allocated with size of multiple 8, which dictates how many padding bytes for the struct.

2. See the figures in the next page. Three registers are required.

3. a) The number of memory reference is 
   instruction reference + operand reference
   = 4+9*100+3+2+2*200+1 = 1110.
   
   b) The instructions dynamically executed is 4+9*100+3 = 907.
Step 1

A = B = C = D = ...
...C_L...
...D_L...
...B_L...
...A_L...

X_L: The last appearance of variable X

Step 2: Interference Graph

Step 3: Colored Graph

The live ranges of C and D are not overlapped.
They can share the same register.

Step 4: register-allocated code

R1 = R2 = R3 = ...
...R3...
...R3...
...R2...
...R1...

The live range of A
live range of B
live range of C
live range of D
Step 1

A = live range of A
B = live range of B
...B_L...
C = live range of C
...A_L...
D = live range of D
...D_L...
...C_L...

X_L: The last appearance of variable X

Step 2: Interference Graph

A –> B

Step 3: Colored Graph

A –> B

Step 4: register-allocated code

R1 =
R2 =
...
...R2...
R2 =
...R1...
R1 =.
...R1...
...R2...

The live ranges of A and D are not overlapped. They can share the same register.