Assembly Languages

One step up from machine language
Originally a more user-friendly way to program
Now mostly a compiler target
Model of computation: stored program computer

Assembly Language Model

Types of Opcodes

Arithmetic, logical
- add, sub, mult
- and, or
- Cmp

Memory load/store
- ld, st

Control transfer
- jmp
- bne

Complex
- movs

Assembly Language Instructions

Built from two pieces:

Opcode
What to do with the data

Operands
Where to get the data

add R1, R3, 3

Types of Assembly Languages

Assembly language closely tied to processor architecture
At least four main types:
CISC: Complex Instruction-Set Computer
RISC: Reduced Instruction-Set Computer
DSP: Digital Signal Processor
VLIW: Very Long Instruction Word

CISC Assembly Language

Developed when people wrote assembly language
Complicated, often specialized instructions with many effects
Examples from x86 architecture
- String move
- Procedure enter, leave

Many, complicated addressing modes
So complicated, often executed by a little program (microcode)
Examples: Intel x86, 68000, PDP-11

RISC Assembly Language

Response to growing use of compilers
Easier-to-target, uniform instruction sets
“Make the most common operations as fast as possible”
Load-store architecture:
- Arithmetic only performed on registers
- Memory load/store instructions for memory-register transfers
Designed to be pipelined
Examples: SPARC, MIPS, HP-PA, PowerPC

Operands

Each operand taken from a particular addressing mode:

Examples:
- Register: add r1, r2, r3
- Immediate: add r1, r2, 10
- Indirect: mov r1, (r2)
- Offset: mov r1, 10(r3)
- PC Relative: beq 100

Reflect processor data pathways
DSP Assembly Language

Digital signal processors designed specifically for signal processing algorithms.
Lots of regular arithmetic on vectors.
Often written by hand.
Irregular architectures to save power, area.
Substantial instruction-level parallelism.
Examples: TI 320, Motorola 56000, Analog Devices.

VLIW Assembly Language

Response to growing desire for instruction-level parallelism.
Using more transistors cheaper than running them faster.
Many parallel ALUs.
Objective: keep them all busy all the time.
Heavily pipelined.
More regular instruction set.
Very difficult to program by hand.
Looks like parallel RISC instructions.
Examples: Itanium, TI 320C6000.

Example: Euclid’s Algorithm

```c
int gcd(int m, int n) {
    int r;
    while ((r = m % n) != 0) {
        m = n;
        n = r;
    }
    return n;
}
```

Euclid on the i386

```asm
.file "euclid.c"
.globl gcd
gcc2 Compiled:
.text
.align 4
.globl gcd
.type gcd,@function
gcd:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%eax
    movl 12(%ebp),%ecx
    jmp .L6
    p2align 4,7
```

Euclid in the i386

```asm
jmp .L6
p2align 4,7
.L6:
    movl %ecx,%eax
    movl %ebx,%ecx
    jmp .L6
```

Euclid on the i386

```asm
jmp .L6
p2align 4,7
.L4:
    movl %ecx,%eax
    movl %ebx,%ecx
    jmp .L4
```

Euclid on the i386

```asm
jmp .L6
p2align 4,7
```

SPARC Programmer’s Model

```
31 0 15 0
r0 Always 0
r1 GLOBAL REGISTERS
r24/10 Input Registers
r30/16 Frame Pointer
r31/7 Return Address
r7 Output Registers
r8/00 PSW
r14/06 Status Word
r15/07 Program Counter
r16/0 Next PC
r23/17 Local Registers
```

i386 Programmer’s Model

```
0 31 0 15 0
eax Mostly cs Code segment
ebx General- ds Data segment
ecx Purpose- es Extra segment
dx Source index fs Data segment
esi Destination index gs Data segment
edi Base pointer
ebp Stack pointer
esp Stack pointer
eflags Status word
eip Instruction Pointer
```

Euclid on the i386

```
jmp .L6
p2align 4,7
.L4:
    movl %eax,%ecx
    movl %esi,%edi
    movl 8(%ebp),%edx
    movl -4(%ebp),%ebx
    leave
    ret
```

SPARC Programmer’s Model
SPARC Register Windows

The output registers of the calling procedure become the inputs to the called procedure. The global registers remain unchanged. The local registers are not visible across procedures.

Euclid on the SPARC

```assembly
.file "euclid.c" # Boilerplate
include "euclid.h"
gcc2.compiled:. # make .rem linker-visible
.section ".text" # Executable code
.align 4
.global gcd # make gcd linker-visible
type gcd, #function
proc 04
gcd:
    save %sp, -112, %sp # Next window, move SP
    mov %i0, %o1 # Move m into o1
    b .LL3 # Unconditional branch
    mov %i1, %i0 # Move n into i0
    .LL5:
        mov %o0, %i0 # n = r
        mov %o1, %o0 # Compute the remainder of call .rem, 0 # m / n, result in o0
        mov %i0, %i1 # m = n (always executed)
bne .LL5 # Return (actually jmp i7 + 8)
    ret # Restore previous window
```

Digital Signal Processor Apps.

- Low-cost embedded systems
  - Modems, cellular telephones, disk drives, printers
- High-throughput applications
  - Halftoning, base stations, 3-D sonar, tomography
- PC based multimedia
  - Compression/decompression of audio, graphics, video

Embedded Processor Requirements

- Inexpensive with small area and volume
- Deterministic interrupt service routine latency
- Low power: ≈50 mW (TMS320C54x uses 0.36 μA/MIPS)

Conventional DSP Architecture

- Harvard architecture
  - Separate data memory/bus and program memory/bus
  - Three reads and one or two writes per instruction cycle
- Deterministic interrupt service routine latency
- Multiply-accumulate in single instruction cycle
- Special addressing modes supported in hardware
  - Modulo addressing for circular buffers for FIR filters
  - Bit-reversed addressing for fast Fourier transforms
- Instructions to keep the pipeline (3-4 stages) full
  - Zero-overhead looping (one pipeline flush to set up)
  - Delayed branches

Conventional DSPs

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Fixed-Point</th>
<th>Floating-Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost/Unit</td>
<td>$5–$79</td>
<td>$5–$381</td>
</tr>
<tr>
<td>Registers</td>
<td>2–4 data, 8 address</td>
<td>8–16 data, 8–16 address</td>
</tr>
<tr>
<td>Data Words</td>
<td>16 or 24 bit</td>
<td>32 bit</td>
</tr>
<tr>
<td>Chip Memory</td>
<td>2–64K data+program</td>
<td>8–64K data+program</td>
</tr>
<tr>
<td>Address Space</td>
<td>16–128K data</td>
<td>16M–4G data</td>
</tr>
<tr>
<td>Compilers</td>
<td>Bad C</td>
<td>Better C, C++</td>
</tr>
<tr>
<td>Examples</td>
<td>TI TMS320C5x</td>
<td>TI TMS320C3x</td>
</tr>
<tr>
<td></td>
<td>Motorola 56000</td>
<td>Analog Devices SHARC</td>
</tr>
</tbody>
</table>

Example

Finite Impulse Response filter (FIR)

Market share: 95% fixed-point, 5% floating-point
Each processor comes in dozens of configurations
- Data and program memory size
- Peripherals: A/D, D/A, serial, parallel ports, timers

Drawbacks
- No byte addressing (needed for image and video)
- Limited on-chip memory
- Limited addressable memory on most fixed-point DSPs
- Non-standard C extensions to support fixed-point data

y_n = \sum_{i=0}^{k} a_i x_{n+i}

where
a_0, \ldots, a_k are filter coefficients,
x_n is the n-th input sample, y_n is the n-th output sample.
56000 Programmer’s Model

<table>
<thead>
<tr>
<th>55 4847</th>
<th>2423 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Registers</td>
</tr>
<tr>
<td></td>
<td>Accumulator</td>
</tr>
<tr>
<td></td>
<td>Accumulator</td>
</tr>
<tr>
<td>15 0</td>
<td>Program Counter</td>
</tr>
<tr>
<td>15 0</td>
<td>Status Register</td>
</tr>
<tr>
<td>15 0</td>
<td>Loop Address</td>
</tr>
<tr>
<td>15 0</td>
<td>Loop Count</td>
</tr>
<tr>
<td></td>
<td>PC Stack</td>
</tr>
<tr>
<td></td>
<td>SR Stack</td>
</tr>
<tr>
<td></td>
<td>Address</td>
</tr>
<tr>
<td></td>
<td>Registers</td>
</tr>
<tr>
<td></td>
<td>Stack pointer</td>
</tr>
</tbody>
</table>

56001 Memory Spaces

Three memory regions, each 64K:
- 24-bit Program memory
- 24-bit X data memory
- 24-bit Y data memory

Idea: enable simultaneous access of program, sample, and coefficient memory

Three on-chip memory spaces can be used this way
One off-chip memory pathway connected to all three memory spaces
Only one off-chip access per cycle maximum

56001 Address Generation

Addresses come from pointer register r0 … r7
Offset registers r0 … r7 can be added to pointer
Modifier registers cause the address to wrap around
Zero modifier causes reverse-carry arithmetic

Address | Notation | Next value of r0 |
---------|----------|-----------------|
| r0      | (r0)     | r0              |
| r0 + n0 | (r0+n0)  | (r0 + 1) mod m0 |
| r0      | (r0)+    | (r0 - 1) mod m0 |
| r0      | (r0)-    | (r0 - 1) mod m0 |
| r0      | (r0)-n0  | (r0 - n0) mod m0 |

FIR Filter in 56001

\[ n \text{ equ 20} \] # Define symbolic constants
\[ \text{start equ $40} \]
\[ \text{samples equ $0} \]
\[ \text{coeffs equ $0} \]
\[ \text{input equ $ffe0} \] # Memory-mapped I/O
\[ \text{output equ $ffe1} \]

\[ \text{org p:start} \] # Locate in prog. memory
\[ \text{move \#samples, r0} \] # Pointers to samples
\[ \text{move \#coeffs, r4} \] # and coefficients
\[ \text{move \#n-1, m0} \] # Prepare circular buffer
\[ \text{move m0, m4} \]

FIR Filter in One ’C6 Assembly Instruction

```
FIRLOOP:
LDH .D1 *A1++, A2     ; Fetch next sample
LDH .D2 *B1++, B2     ; Fetch next coeff.
[BO] SUB .L2 B0, 1, B0 ; Decrement count
[BO] B .S2 FIRLOOP     ; Branch if non-zero
MFY .MIX A2, B2, A3   ; Sample \times \ Coeff.
ADD .L1 A4, A3, A4    ; Accumulate result
```

TI TMS320C6000 VLIW DSP

Eight instruction units dispatched by one very long instruction word
Designed for DSP applications
Orthogonal instruction set
Big, uniform register file (16 32-bit registers)
Better compiler target than 56001
Deeply pipelined (up to 15 levels)
Complicated, but more regular, datapath

Pipelining on the C6

One instruction issued per clock cycle
Very deep pipeline
- 4 fetch cycles
- 2 decode cycles
- 1-10 execute cycles
Branch in pipeline disables interrupts
Conditional instructions avoid branch-induced stalls
No hardware to protect against hazards
- Assembler or compiler’s responsibility

FIR in One ’C6 Assembly Instruction

```
LDH .D1 *A1++, A2     ; Fetch next sample
LDH .D2 *B1++, B2     ; Fetch next coeff.
```

Run these instruction in parallel

Peripherals

Often the whole point of the system
Memory-mapped I/O
- Magical memory locations that make something happen or change on their own
Typical meanings:
- Configuration (write)
- Status (read)
- Address/Data (access more peripheral state)
Example: 56001 Port C

Nine pins each usable as either simple parallel I/O or as part of two serial interfaces.

Pins:

<table>
<thead>
<tr>
<th>Parallel</th>
<th>Serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC0</td>
<td>RXD</td>
</tr>
<tr>
<td>PC1</td>
<td>TXD</td>
</tr>
<tr>
<td>PC2</td>
<td>SCLK</td>
</tr>
<tr>
<td>PC3</td>
<td>SC0</td>
</tr>
<tr>
<td>PC4</td>
<td>SC1</td>
</tr>
<tr>
<td>PC5</td>
<td>SC2</td>
</tr>
<tr>
<td>PC6</td>
<td>SCK</td>
</tr>
<tr>
<td>PC7</td>
<td>SRD</td>
</tr>
<tr>
<td>PC8</td>
<td>STD</td>
</tr>
</tbody>
</table>

Port C SCI Registers

SCI Control Register

X: $FFFO

<table>
<thead>
<tr>
<th>Bits</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–2</td>
<td>Word select bits</td>
</tr>
<tr>
<td>3</td>
<td>Shift direction</td>
</tr>
<tr>
<td>4</td>
<td>Send break</td>
</tr>
<tr>
<td>5</td>
<td>Wakeup mode select</td>
</tr>
<tr>
<td>6</td>
<td>Receiver wakeup enable</td>
</tr>
<tr>
<td>7</td>
<td>Wired-OR mode select</td>
</tr>
<tr>
<td>8</td>
<td>Receiver enable</td>
</tr>
<tr>
<td>9</td>
<td>Transmitter enable</td>
</tr>
<tr>
<td>10</td>
<td>Idle line interrupt enable</td>
</tr>
<tr>
<td>11</td>
<td>Receive interrupt enable</td>
</tr>
<tr>
<td>12</td>
<td>Transmit interrupt enable</td>
</tr>
<tr>
<td>13</td>
<td>Timer interrupt enable</td>
</tr>
<tr>
<td>15</td>
<td>Clock polarity</td>
</tr>
</tbody>
</table>

SCI Status Register (Read only)

X: $FFF1

<table>
<thead>
<tr>
<th>Bits</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Transmitter Empty</td>
</tr>
<tr>
<td>1</td>
<td>Transmitter Reg Empty</td>
</tr>
<tr>
<td>2</td>
<td>Receive Data Full</td>
</tr>
<tr>
<td>3</td>
<td>Idle Line</td>
</tr>
<tr>
<td>4</td>
<td>Overrun Error</td>
</tr>
<tr>
<td>5</td>
<td>Parity Error</td>
</tr>
<tr>
<td>6</td>
<td>Framing Error</td>
</tr>
<tr>
<td>7</td>
<td>Received bit 8</td>
</tr>
</tbody>
</table>

SCI Clock Control Register

X: $FFF2

<table>
<thead>
<tr>
<th>Bits</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>11–0</td>
<td>Clock Divider</td>
</tr>
<tr>
<td>12</td>
<td>Clock Output Divider</td>
</tr>
<tr>
<td>13</td>
<td>Clock Prescaler</td>
</tr>
<tr>
<td>14</td>
<td>Receive Clock Source</td>
</tr>
<tr>
<td>15</td>
<td>Transmit Clock Source</td>
</tr>
</tbody>
</table>

Port C SSI

Intended for synchronous, constant-rate protocols

Easy interface to serial ADCs and DACs

Many more operating modes than SCI

Six Pins (Rx, Tx, Clk, Rx Clk, Frame Sync, Tx Clk)

8, 12, 16, or 24-bit words

Port C SSI Registers

SSI Control Register A $FFEC

Prescaler, frame rate, word length

SSI Control Register B $FFED

Interrupt enables, various mode settings

SSI Status/Time Slot Register $FFEE

Sync, empty, overrun

SSI Receive/Transmit Data Register $FFEF

8, 16, or 24 bits of read/write data.