# Assembly Languages

COMS W4995-02

Prof. Stephen A. Edwards
Fall 2002
Columbia University
Department of Computer Science

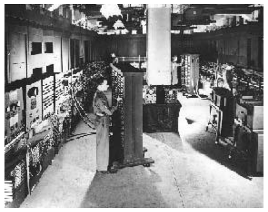
## Assembly Languages

One step up from machine language

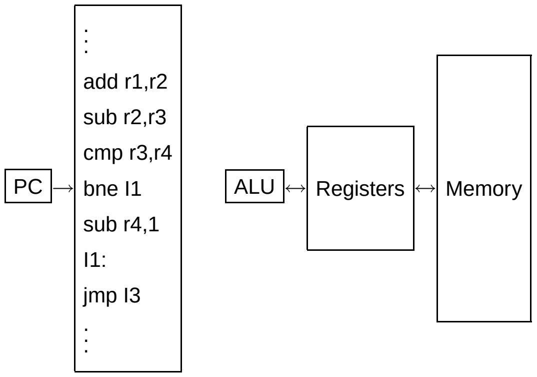
Originally a more user-friendly way to program

Now mostly a compiler target

Model of computation: stored program computer



## Assembly Language Model



## Assembly Language Instructions

Built from two pieces:

$$\underset{\substack{\text{Opcode}\\\text{What to do with the data}}}{\texttt{add}}\ \underset{\substack{\text{Operands}\\\text{Where to get the data}}}{\texttt{R1, R3, 3}}$$

## Types of Opcodes

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

Memory load/store
- ld, st

Control transfer
- jmp
- bne

Complex
- movs

## 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

## 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

## 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

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

## i386 Programmer’s Model

| 31 0 | |
|---|---|
| eax | Mostly |
| ebx | General- |
| ecx | Purpose- |
| edx | Registers |
| esi | Source index |
| edi | Destination index |
| ebp | Base pointer |
| esp | Stack pointer |
| eflags | Status word |
| eip | Instruction Pointer |

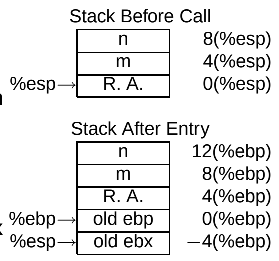
| 15 0 | |
|---|---|
| cs | Code segment |
| ds | Data segment |
| ss | Stack segment |
| es | Extra segment |
| fs | Data segment |
| gs | Data segment |

## Euclid on the i386

```
  .file "euclid.c"          # Boilerplate
  .version "01.01"
gcc2_compiled.:
  .text                     # Executable
  .align 4                  # Start on 16-byte boundary
  .globl gcd                # Make “gcd” linker-visible
  .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 on the i386

```
  .file "euclid.c"
  .version "01.01"
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

```
  jmp .L6          # Jump to local label .L6
.p2align 4,,7      # Skip ≤ 7 bytes to a multiple of 16
.L4:
  movl %ecx,%eax
  movl %ebx,%ecx
.L6:
  cltd               # Sign-extend eax to edx:eax
  idivl %ecx         # Compute edx:eax / ecx
  movl %edx,%ebx
  testl %edx,%edx
  jne .L4
  movl %ecx,%eax
  movl -4(%ebp),%ebx
  leave
  ret
```

## Euclid on the i386

```
  jmp .L6
.p2align 4,,7
.L4:
  movl %ecx,%eax    # m = n
  movl %ebx,%ecx    # n = r
.L6:
  cltd
  idivl %ecx
  movl %edx,%ebx
  testl %edx,%edx   # AND of edx and edx
  jne .L4           # branch if edx was ≠ 0
  movl %ecx,%eax    # Return n
  movl -4(%ebp),%ebx
  leave             # Move ebp to esp, pop ebp
  ret               # Pop return address and branch
```

## SPARC Programmer’s Model

| 31 0 | |
|---|---|
| r0 | Always 0 |
| r1 | Global Registers |
| ⋮ | |
| r7 | |
| r8/o0 | Output Registers |
| ⋮ | |
| r14/o6 | Stack Pointer |
| r15/o7 | |
| r16/l0 | Local Registers |
| ⋮ | |
| r23/l7 | |

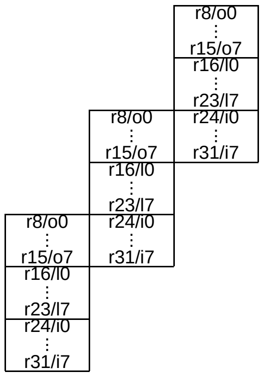
| 31 0 | |
|---|---|
| r24/i0 | Input Registers |
| ⋮ | |
| r30/i6 | Frame Pointer |
| r31/i7 | Return Address |
| PSW | Status Word |
| PC | Program Counter |
| nPC | Next PC |

## 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

```
  .file    "euclid.c"      # Boilerplate
gcc2_compiled.:
  .global .rem             # 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
```

## Euclid on the SPARC

```
   mov  %i0, %o1
   b    .LL3
   mov  %i1, %i0
.LL5:
   mov  %o0, %i0   # n = r
.LL3:
   mov  %o1, %o0   # Compute the remainder of
   call .rem, 0    # m / n, result in o0
   mov  %i0, %o1

   cmp  %o0, 0
   bne  .LL5
   mov  %i0, %o1   # m = n (always executed)
   ret             # Return (actually jmp i7 + 8)
   restore         # 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: $\approx$50 mW (TMS320C54x uses 0.36 $\mu$A/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

| | Fixed-Point | Floating-Point |
|---|---|---|
| **Cost/Unit** | $5–$79 | $5–$381 |
| **Architecture** | Accumulator | load-store |
| **Registers** | 2–4 data, 8 address | 8–16 data, 8–16 address |
| **Data Words** | 16 or 24 bit | 32 bit |
| **Chip Memory** | 2–64K data+program | 8–64K data+program |
| **Address Space** | 16–128K data | 16M–4G data |
| | 16–64K program | 16M–4G program |
| **Compilers** | Bad C | Better C, C++ |
| **Examples** | TI TMS320C5x | TI TMS320C3x |
| | Motorola 56000 | Analog Devices SHARC |

## Conventional DSPs

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

## Example

Finite Impulse Response filter (FIR)

Can be used for lowpass, highpass, bandpass, etc.

Basic DSP operation

For each sample, computes

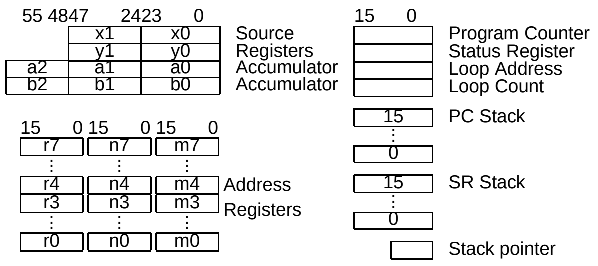
$$y_n = \sum_{i=0}^{k} a_i x_{n+i}$$

where

$a_0, \ldots, a_k$ are filter coeffecients,

$x_n$ is the $n$th input sample, $y_n$ is the $n$th output sample.

## 56000 Programmer's Model



## 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 n0 . . . n7 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 |
| r0 | (r0)+ | (r0 + 1) mod m0 |
| r0 - 1 | -(r0) | r0 - 1 mod m0 |
| r0 | (r0)- | (r0 - 1) mod m0 |
| r0 | (r0)+n0 | (r0 + n0) mod m0 |
| r0 | (r0)-n0 | (r0 - n0) mod m0 |

## FIR Filter in 56001

```
n       equ 20     # Define symbolic constants
start   equ $40
samples equ $0
coeffs  equ $0
input   equ $ffe0 # Memory-mapped I/O
output  equ $ffe1

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

## FIR Filter in 56001

```
movep y:input, x:(r0)  # Load sample into memory
      # Clear accumulator A
                # Load a sample into x0
                             # Load a coefficient
clr   a         x:(r0)+, x0  y:(r4)+, y0

rep   #n-1      # Repeat next instruction n-1 times
      # a = x0 × y0
                # Next sample
                             # Next coefficient
mac   x0,y0,a   x:(r0)+, x0  y:(r4)+, y0

macr  x0,y0,a  (r0)-
movep a, y:output  # Write output sample
```

## 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

Load a halfword (16 bits)

Do this on unit D1

```
FIRLOOP:
        LDH .D1  *A1++, A2   ; Fetch next sample
||      LDH .D2  *B1++, B2   ; Fetch next coeff.
|| [B0] SUB .L2  B0, 1, B0   ; Decrement count
|| [B0] B   .S2  FIRLOOP     ; Branch if non-zero
||      MPY .M1X A2, B2, A3  ; Sample × Coeff.
||      ADD .L1  A4, A3, A4  ; Accumulate result
```

Use the cross path

Predicated instruction (only if B0 non-zero)

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:

| Parallel | Serial | |
|---|---|---|
| PC0 | RxD | Serial Communication Interface (SCI) |
| PC1 | TxD | |
| PC2 | SCLK | |
| PC3 | SC0 | Synchronous Serial Interface (SSI) |
| PC4 | SC1 | |
| PC5 | SC2 | |
| PC6 | SCK | |
| PC7 | SRD | |
| PC8 | STD | |

## Port C Registers for Parallel Port

**Port C Control Register**

Selects mode (parallel or serial) of each pin

X: $FFE1 Lower 9 bits: 0 = parallel, 1 = serial

**Port C Data Direction Register**

I/O direction of parallel pins

X: $FFE3 Lower 9 bits: 0 = input, 1 = output

**Port C Data Register**

Read = parallel input data, Write = parallel data out

X: $FFE5 Lower 9 bits

## Port C SCI

Three-pin interface

422 Kbit/s NRZ asynchronous interface (RS-232-like)

3.375 Mbit/s synchronous serial mode

Multidrop mode for multiprocessor systems

Two Wakeup modes

- Idle line
- Address bit

Wired-OR mode

On-chip or external baud rate generator

Four interrupt priority levels

## Port C SCI Registers

**SCI Control Register**

| X: $FFF0 | Bits | Function |
|---|---|---|
| | 0–2 | Word select bits |
| | 3 | Shift direction |
| | 4 | Send break |
| | 5 | Wakeup mode select |
| | 6 | Receiver wakeup enable |
| | 7 | Wired-OR mode select |
| | 8 | Receiver enable |
| | 9 | Transmitter enable |
| | 10 | Idle line interrupt enable |
| | 11 | Receive interrupt enable |
| | 12 | Transmit interrupt enable |
| | 13 | Timer interrupt enable |
| | 15 | Clock polarity |

## Port C SCI Registers

**SCI Status Register** (Read only)

| X: $FFF1 | Bits | Function |
|---|---|---|
| | 0 | Transmitter Empty |
| | 1 | Transmitter Reg Empty |
| | 2 | Receive Data Full |
| | 3 | Idle Line |
| | 4 | Overrun Error |
| | 5 | Parity Error |
| | 6 | Framing Error |
| | 7 | Received bit 8 |

## Port C SCI Registers

**SCI Clock Control Register**

| X: $FFF2 | Bits | Function |
|---|---|---|
| | 11–0 | Clock Divider |
| | 12 | Clock Output Divider |
| | 13 | Clock Prescaler |
| | 14 | Receive Clock Source |
| | 15 | Transmit Clock Source |

## 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, oerrun

**SSI Receive/Transmit Data Register** $FFEF

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