A. Assembly Language Programming

Programming of a computer system:
• Machine code
  – direct execution
• Assembly language
  – tool: assembler
• High level programming language
  – tool: interpreter
  – tool: compiler

Programming
• Computer comprises hardware and software
• Load program on hardware for a particular task

Software
Input Data → Hardware → Output Data

A.1 Machine Code

Machine Code
Input Data → Microcontroller → Output Data

• Specify program in byte codes corresponding to CPU’s instruction set
  → commands, the CPU can understand directly
• Instruction Format is machine-dependent, e.g. Motorola 68,000

Machine Code Example: Motorola ADD

• Find machine code for: Add two values (4 bytes) in register: D4 + D2 → D2
  ADD.L D4, D2
• Destination register is D2, code 010
  Destination mode is 010 (see table) for Long and destination is data register
• Source mode is 000 (see table) for: addressing mode register direct
  Source register is D4, code 100
• Putting all together:
  1101 010 010 010 0100 1000 0100 0100 0100 D4 84
  opcode, dest.-reg., mode, addr-mode source-reg.
  re-group in 4-bit chunks
  write as hex
Machine Code

Programming in machine code is:
• Very low level programming
• Difficult and error-prone
• Not used in practice anymore

Improvement: Use name abbreviations that make sense
• E.g. ADD for adding, CMP for comparing
• This is what assembly language is all about
• Requires translation tool: “Assembler”

Assembler

• Translation of mnemonic codes to machine language
• Machine-dependent e.g. Motorola 68,000: ADD.L D4,D2 → D4 84
• Line by line execution
• Symbolic address calculation, use of labels
• Cross assembler if hardware1 ≠ hardware2

Compiler for C and C++

• Translation of C/C++ source code to machine language
• Machine-independent
• Use of variable and function names
• Cross compiler if Hardware1 ≠ Hardware2

A.2 Programming Tools

Assembler
• We use “gnu” assembler (freeware for Unix and Windows)
• Call: gas68 mydemo.s -o mydemo.hex
• Execution steps:
  mydemo.s assembly file
  ↓ ASM
  mydemo.o object file
  ↓ Link
  mydemo executable
  ↓ (Object Copy)
  mydemo.hex S-record for download (if cross-assembler)
Programming Tools

Debugger
- allows to **stop program** in mid-execution
- allows to **examine** current register and memory contents
- allows to set **breakpoints**: stop program at a certain instruction
- allows to **single-step** through program
- very helpful in testing and debugging of a program

Disassembler
- converts machine code back into assembly code
- used in combination with debugger

Motorola Programming Tools

Atmel Programming Tools

Operating System
- Not always necessary for embedded systems
- Usually small, compact collection of routines in ROM
  - user interface for download of programs (RAM)
  - routines for Flash-ROM update (easy update of operating system itself)
  - mathematical function library
  - predefined input/output routines
    - display
    - keys
    - digital I/O lines
    - analog input lines
    - serial and parallel port
- For EyeBot controller we use RoBIOS (Robot Basic I/O System)
online documentation: http://robotics.ee.uwa.edu.au/eyebot
A.3 Assembly Commands

<table>
<thead>
<tr>
<th>Label (opt.)</th>
<th>Command</th>
<th>Operands</th>
<th>Comment (opt.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>main:</td>
<td>ADD.L</td>
<td>D4, D2</td>
<td></td>
</tr>
<tr>
<td>main:</td>
<td>DEC</td>
<td>R17</td>
<td>; Atmel</td>
</tr>
</tbody>
</table>

Label: Needed for branching / jumps
Command: Op-code for processor
(Motorola: length attribute L: long=4, W: word=2, B: byte=1 byte)
Operands: The arguments the command operates on (e.g. constant, register, memory)
Comments: Should be meaningful and describe what the process accomplishes, rather than restating the command

Motorola Assembly Subset

Note: With 2 operands, result goes to second operand

• **Arithmetic**
  - ADD  addition
  - SUB  subtraction
  - MUL  multiply signed
  - DIV  division signed
  - MOVE copy data
  - NEG  negate value
  - CLR  clear (set to 0)

• **Logic**
  - AND  and
  - OR   or
  - NOT  not

• **Subroutine**
  - JSR  jump subroutine
  - RET  return

• **Branching**
  - JMP  jump (absolute)
  - BRA  branch (relative)
  - BEQ  branch if equal
  - BNE  branch if not eq.
  - BPL  branch if plus
  - BMI  branch if minus

Atmel (1) Assembly Commands

Note: With 2 operands, result goes to first operand

• **Arithmetic**
  - ADD  add registers
  - ADIW add imm. word
  - SUB  subtr. registers
  - SUBI subtr. immediate
  - MUL  multiply unsigned
  - MULS multiply signed
  - MEA move effective address

• **Logic**
  - AND  and registers
  - ANDI and immediate
  - OR   or registers
  - ORI or immediate
  - EOR  exclusive-or registers
  - COM  not (complement)
  - LSL  logical shift left
  - LSR  logical shift right
  - ROL  rotate left (carry)
  - ROR  rotate right (carry)

• **Subroutine**
  - CALL call subroutine
  - RET  return

• **Stack**
  - PUSH push reg. on stack
  - POP  pop data from stack

• **Memory values (variables)**
  - usually specified by labels

Motorola: ADD.L name, D1
Atmel: LDS R16, name
Atmel (2) Assembly Commands

Note: With 2 operands, result goes to first operand

- **Compare**
  - CP: compare registers
  - CPI: compare with immediate
  - TST: test single register

- **Branching**
  - JMP: jump (absolute)
  - BREQ: branch if equal (=0)
  - BRNE: branch if not equal
  - BRGE: branch if greater than or equal
  - BRLT: branch if less than
  - BRPL: branch if plus
  - BRMI: branch if minus
  - BRCS: branch if carry set
  - BRCC: branch if carry clear

- **Move Data**
  - LDI: load register immediate
  - LD: load indirect
  - LDS: load memory
  - ST: store indirect
  - STS: store memory

- **Bit Operations**
  - SBR: set bit in register
  - CBR: clear bit in register
  - SBI: set bit in I/O register
  - CBI: clear bit in I/O register

---

Atmel Assembly Commands

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDI R, X</td>
<td>Load Register</td>
<td>R = X</td>
</tr>
<tr>
<td>LD R, X</td>
<td>Load Indirect</td>
<td>R = [X]</td>
</tr>
<tr>
<td>LDS R, X</td>
<td>Load Memory</td>
<td>R = [X]</td>
</tr>
<tr>
<td>ST R, X</td>
<td>Store Indirect</td>
<td>[X] = R</td>
</tr>
<tr>
<td>STS R, X</td>
<td>Store Memory</td>
<td>[X] = R</td>
</tr>
<tr>
<td>SBR R, X</td>
<td>Set Bit Register</td>
<td>R</td>
</tr>
<tr>
<td>CBR R, X</td>
<td>Clear Bit Register</td>
<td>R &amp;= X</td>
</tr>
<tr>
<td>SBI R, X</td>
<td>Set Bit I/O Register</td>
<td>R</td>
</tr>
<tr>
<td>CBI R, X</td>
<td>Clear Bit I/O Register</td>
<td>R &amp;= X</td>
</tr>
</tbody>
</table>

---

Atmel Assembly Commands

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R, R</td>
<td>Move Registers</td>
<td>R = R</td>
</tr>
<tr>
<td>IN R</td>
<td>Read Input Data</td>
<td>R = IN</td>
</tr>
<tr>
<td>OUT R</td>
<td>Write Output Data</td>
<td>OUT = R</td>
</tr>
</tbody>
</table>

---

Atmel Assembly Commands

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP R, X</td>
<td>Jump (Absolute)</td>
<td>R + X</td>
</tr>
<tr>
<td>BREQ R, X</td>
<td>Branch if Equal (=0)</td>
<td>R = 0? R</td>
</tr>
<tr>
<td>BRNE R, X</td>
<td>Branch if Not Equal</td>
<td>R ≠ 0? R</td>
</tr>
<tr>
<td>BRGE R, X</td>
<td>Branch if Greater than or Equal</td>
<td>R ≥ 0? R</td>
</tr>
<tr>
<td>BRLT R, X</td>
<td>Branch if Less than</td>
<td>R &lt; 0? R</td>
</tr>
<tr>
<td>BRPL R, X</td>
<td>Branch if Plus</td>
<td>R ≥ 0? R</td>
</tr>
<tr>
<td>BRMI R, X</td>
<td>Branch if Minus</td>
<td>R ≤ 0? R</td>
</tr>
<tr>
<td>BRCS R, X</td>
<td>Branch if Carry Set</td>
<td>R &amp; C? R</td>
</tr>
<tr>
<td>BRCC R, X</td>
<td>Branch if Carry Clear</td>
<td>R &amp;¬ C? R</td>
</tr>
</tbody>
</table>

---

Atmel Assembly Commands

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDI R, X</td>
<td>Load Register</td>
<td>R = X</td>
</tr>
<tr>
<td>LD R, X</td>
<td>Load Indirect</td>
<td>R = [X]</td>
</tr>
<tr>
<td>LDS R, X</td>
<td>Load Memory</td>
<td>R = [X]</td>
</tr>
<tr>
<td>ST R, X</td>
<td>Store Indirect</td>
<td>[X] = R</td>
</tr>
<tr>
<td>STS R, X</td>
<td>Store Memory</td>
<td>[X] = R</td>
</tr>
<tr>
<td>SBR R, X</td>
<td>Set Bit Register</td>
<td>R</td>
</tr>
<tr>
<td>CBR R, X</td>
<td>Clear Bit Register</td>
<td>R &amp;= X</td>
</tr>
<tr>
<td>SBI R, X</td>
<td>Set Bit I/O Register</td>
<td>R</td>
</tr>
<tr>
<td>CBI R, X</td>
<td>Clear Bit I/O Register</td>
<td>R &amp;= X</td>
</tr>
</tbody>
</table>

---

Atmel Assembly Commands

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP R, X</td>
<td>Jump (Absolute)</td>
<td>R + X</td>
</tr>
<tr>
<td>BREQ R, X</td>
<td>Branch if Equal (=0)</td>
<td>R = 0? R</td>
</tr>
<tr>
<td>BRNE R, X</td>
<td>Branch if Not Equal</td>
<td>R ≠ 0? R</td>
</tr>
<tr>
<td>BRGE R, X</td>
<td>Branch if Greater than or Equal</td>
<td>R ≥ 0? R</td>
</tr>
<tr>
<td>BRLT R, X</td>
<td>Branch if Less than</td>
<td>R &lt; 0? R</td>
</tr>
<tr>
<td>BRPL R, X</td>
<td>Branch if Plus</td>
<td>R ≥ 0? R</td>
</tr>
<tr>
<td>BRMI R, X</td>
<td>Branch if Minus</td>
<td>R ≤ 0? R</td>
</tr>
<tr>
<td>BRCS R, X</td>
<td>Branch if Carry Set</td>
<td>R &amp; C? R</td>
</tr>
<tr>
<td>BRCC R, X</td>
<td>Branch if Carry Clear</td>
<td>R &amp;¬ C? R</td>
</tr>
</tbody>
</table>

---

Atmel Assembly Commands

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R, R</td>
<td>Move Registers</td>
<td>R = R</td>
</tr>
<tr>
<td>IN R</td>
<td>Read Input Data</td>
<td>R = IN</td>
</tr>
<tr>
<td>OUT R</td>
<td>Write Output Data</td>
<td>OUT = R</td>
</tr>
</tbody>
</table>

---

Atmel Assembly Commands

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP R, X</td>
<td>Jump (Absolute)</td>
<td>R + X</td>
</tr>
<tr>
<td>BREQ R, X</td>
<td>Branch if Equal (=0)</td>
<td>R = 0? R</td>
</tr>
<tr>
<td>BRNE R, X</td>
<td>Branch if Not Equal</td>
<td>R ≠ 0? R</td>
</tr>
<tr>
<td>BRGE R, X</td>
<td>Branch if Greater than or Equal</td>
<td>R ≥ 0? R</td>
</tr>
<tr>
<td>BRLT R, X</td>
<td>Branch if Less than</td>
<td>R &lt; 0? R</td>
</tr>
<tr>
<td>BRPL R, X</td>
<td>Branch if Plus</td>
<td>R ≥ 0? R</td>
</tr>
<tr>
<td>BRMI R, X</td>
<td>Branch if Minus</td>
<td>R ≤ 0? R</td>
</tr>
<tr>
<td>BRCS R, X</td>
<td>Branch if Carry Set</td>
<td>R &amp; C? R</td>
</tr>
<tr>
<td>BRCC R, X</td>
<td>Branch if Carry Clear</td>
<td>R &amp;¬ C? R</td>
</tr>
</tbody>
</table>

---

Atmel Assembly Commands

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDI R, X</td>
<td>Load Register</td>
<td>R = X</td>
</tr>
<tr>
<td>LD R, X</td>
<td>Load Indirect</td>
<td>R = [X]</td>
</tr>
<tr>
<td>LDS R, X</td>
<td>Load Memory</td>
<td>R = [X]</td>
</tr>
<tr>
<td>ST R, X</td>
<td>Store Indirect</td>
<td>[X] = R</td>
</tr>
<tr>
<td>STS R, X</td>
<td>Store Memory</td>
<td>[X] = R</td>
</tr>
<tr>
<td>SBR R, X</td>
<td>Set Bit Register</td>
<td>R</td>
</tr>
<tr>
<td>CBR R, X</td>
<td>Clear Bit Register</td>
<td>R &amp;= X</td>
</tr>
<tr>
<td>SBI R, X</td>
<td>Set Bit I/O Register</td>
<td>R</td>
</tr>
<tr>
<td>CBI R, X</td>
<td>Clear Bit I/O Register</td>
<td>R &amp;= X</td>
</tr>
</tbody>
</table>
### Atmel Assembly Commands

#### ADD – Add without Carry

**Description:**

Adds two registers without the C Flag and places the result in the destination register Rd.

**Operation:**

(i) \( Rd \leftarrow Rd + Rr \)

**Syntax:**

(i) ADD Rd, Rr  \( 0.s \leq s \leq 31, 0.s \leq s \leq 31 \)

**Program Counter:**

\( \text{PC} \leftarrow \text{PC} + 1 \)

**16-bit Opcode:**

<table>
<thead>
<tr>
<th>0000</th>
<th>li</th>
<th>ddss</th>
<th>rrrr</th>
</tr>
</thead>
</table>

### Status Register (BREG) and Boolean Formula:

<table>
<thead>
<tr>
<th>I</th>
<th>T</th>
<th>H</th>
<th>S</th>
<th>V</th>
<th>N</th>
<th>Z</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

### Atmel Branching

#### Conditional Branch Summary

<table>
<thead>
<tr>
<th>Text</th>
<th>Boolean</th>
<th>Mnemonic</th>
<th>Complementary</th>
<th>Boolean</th>
<th>Mnemonic</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Rd + N )</td>
<td></td>
<td>BRNL</td>
<td>( Rd + N )</td>
<td></td>
<td>BRNL</td>
<td></td>
</tr>
<tr>
<td>( Rd + N )</td>
<td></td>
<td>BRLL</td>
<td>( Rd + N )</td>
<td></td>
<td>BRLL</td>
<td></td>
</tr>
<tr>
<td>( Rd + N )</td>
<td></td>
<td>BREQ</td>
<td>( Rd + N )</td>
<td></td>
<td>BREQ</td>
<td></td>
</tr>
<tr>
<td>( Rd + N )</td>
<td></td>
<td>BREQH</td>
<td>( Rd + N )</td>
<td></td>
<td>BREQH</td>
<td></td>
</tr>
<tr>
<td>( Rd + N )</td>
<td></td>
<td>BREQL</td>
<td>( Rd + N )</td>
<td></td>
<td>BREQL</td>
<td></td>
</tr>
<tr>
<td>( Rd + N )</td>
<td></td>
<td>BREQ</td>
<td>( Rd + N )</td>
<td></td>
<td>BREQ</td>
<td></td>
</tr>
<tr>
<td>( Rd + N )</td>
<td></td>
<td>BRF</td>
<td>( Rd + N )</td>
<td></td>
<td>BRF</td>
<td></td>
</tr>
<tr>
<td>( Rd + N )</td>
<td></td>
<td>BRS</td>
<td>( Rd + N )</td>
<td></td>
<td>BRS</td>
<td></td>
</tr>
<tr>
<td>( Rd + N )</td>
<td></td>
<td>BRH</td>
<td>( Rd + N )</td>
<td></td>
<td>BRH</td>
<td></td>
</tr>
<tr>
<td>( Rd + N )</td>
<td></td>
<td>BRC</td>
<td>( Rd + N )</td>
<td></td>
<td>BRC</td>
<td></td>
</tr>
<tr>
<td>( Rd + N )</td>
<td></td>
<td>BRD</td>
<td>( Rd + N )</td>
<td></td>
<td>BRD</td>
<td></td>
</tr>
<tr>
<td>( Rd + N )</td>
<td></td>
<td>BRF</td>
<td>( Rd + N )</td>
<td></td>
<td>BRF</td>
<td></td>
</tr>
<tr>
<td>( Rd + N )</td>
<td></td>
<td>BRF</td>
<td>( Rd + N )</td>
<td></td>
<td>BRF</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** 1. Interchange \( Rd \) and \( Rr \) in the operation before the test, i.e., \( CP \text{ Rd}, \text{Rr} \rightarrow \text{CP} \text{ Rr}, \text{Rd} \).
Atmel Registers

- 32 General purpose registers, 8-bit, memory-mapped $00..$1F
  R0 .. R31
- 6 of those registers (R26..R31) can be combined to form 16-bit reg.,
  X, Y, Z (e.g. for memory addressing)
- 64 Special registers for control and I/O, 8-bit, mem.-map. $20..$5F
  GPIO0, GPIO1, SMCR, SP, ...
- Further extended I/O space, mem.-map. $60..$FF
- Status register SREG

Atmel General Purpose Registers

<table>
<thead>
<tr>
<th>32 Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 I/O Registers</td>
</tr>
<tr>
<td>160 Ext I/O Reg.</td>
</tr>
</tbody>
</table>

Atmel Memory Map

- Data Memory
- 32 Registers
- 64 I/O Registers
- 160 Ext I/O Reg.
- Internal SRAM (1024 x 8)
- 0x0000 - 0x001F
- 0x0020 - 0x005F
- 0x0060 - 0x00FF
- 0x0100 - 0x04FF

Atmel SREG

- Status Register

Note: Not all instructions work with all registers ...
Atmel Single Cycle ALU Operation

Note: Slightly overlapping

Atmel Port Input/Output

PORTA – Port A Data Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
</tr>
<tr>
<td>Initial Value</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

DDRA – Port A Data Direction Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
</tr>
<tr>
<td>Initial Value</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

PINA – Port A Input Pins Address

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
</tr>
<tr>
<td>Initial Value</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Atmel Port Input/Output

Atmel Addressing Modes

Register Direct, Single Register Rd

Figure 1. Direct Single Register Addressing

The operand is contained in register d (Rd).

...
Atmel Addressing Modes

Register Direct, Two Registers Rrd and Rr

Figure 2. Direct Register Addressing, Two Registers

Operands are contained in register r (Rr) and d (Rd). The result is stored in register d (Rd).

Atmel Addressing Modes

Figure 3. I/O Direct Addressing

Operand address is contained in 6 bits of the instruction word. n is the destination or source register address.

Atmel Addressing Modes

Data Direct

Figure 4. Direct Data Addressing

A 16-bit Data Address is contained in the 16 LSBs of a two-word instruction. Rd/Rr specify the destination or source register.

Atmel Addressing Modes

Figure 5. Data Indirect with Displacement

Operand address is the result of the Y- or Z-register contents added to the address contained in 6 bits of the instruction word. Rd/Rr specify the destination or source register.
Atmel Addressing Modes

Data Indirect

Figure 6. Data Indirect Addressing

Operand address is the contents of the X-, Y-, or the Z-register. In AVR devices without SRAM, Data Indirect Addressing is called Register Indirect Addressing. Register Indirect Addressing is a subset of Data Indirect Addressing since the data space from 0 to 31 is the Register File.

Atmel Addressing Modes

Data Indirect with Pre-decrement

Figure 7. Data Indirect Addressing with Pre-decrement

The X-, Y-, or the Z-register is decremented before the operation. Operand address is the decremented contents of the X-, Y-, or the Z-register.

Atmel Addressing Modes

Data Indirect with Post-increment

Figure 8. Data Indirect Addressing with Post-increment

The X-, Y-, or the Z-register is incremented after the operation. Operand address is the content of the X-, Y-, or the Z-register prior to incrementing.

Atmel Addressing Modes

Figure 9. Program Memory Constant Addressing

Constant byte address is specified by the Z-register contents. The 15 LSB bits select word address. For LPM, the LSB selects low byte if cleared (LSB = 0) or high byte if set (LSB = 1). For SPM, the LSB should be cleared. If ELPM is used, the RAMP2Z Register is used to extend the Z-register.
Atmel Addressing Modes

Program Memory with Post-Increment using the LPM Z+ and ELPM Z+ Instruction

Figure 10. Program Memory Addressing with Post-Increment

Constant byte address is specified by the Z-register contents. The 15 MSBs select word address. The LSB selects low byte if cleared (LSB = 0) or high byte if set (LSB = 1). If ELPM Z+ is used, the RAMPZ register is used to extend the Z-register.

Bräunl 2007

Atmel Addressing Modes

Figure 11. Direct Program Memory Addressing

Program execution continues at the address immediate in the instruction word.

Bräunl 2007

Atmel Addressing Modes

Indirect Program Addressing, IJMP and ICALL

Figure 12. Indirect Program Memory Addressing

Program execution continues at address contained by the Z-register (i.e., the PC is loaded with the contents of the Z-register).

Bräunl 2007

Atmel Addressing Modes

Figure 13. Relative Program Memory Addressing

Program execution continues at address PC + k + 1. The relative address k is from -2048 to 2047.

Bräunl 2007
M68,000 CPU Registers

Unlike our simple CPUs, which had only 1 data register (accumulator) and 1 address register, the 68,000 has

- 8 data registers: D0 .. D7
- 8 address registers: A0 .. A7 (A7 is stack pointer, therefore should not be used in user program)
- Each register (data and address) is 4 bytes long
- The command length (L, W, B) determines whether the whole register is used or just right half or right byte

Motorola Memory Data

- Data in memory is declared by using labels
- Memory data
  datal: DC.L 12 | declare constant “datal”, 4 bytes, value 12
table1: DS.L 12 | declare storage “table1”, 12*4 bytes, no init.

- Equivalence (define constant name, see “#define” in C)
  ClearScreen = 12

- Data in memory can be used via its label
  MOVE.L datal, D1 | copy memory value to register
  MOVE.L D6, datal | copy register value to memory

- String constants
  str1: .ascii “hello” | string “hello”, 5 bytes, not terminated
  str2: .asciz “hello” | string “hello/0”, 6 bytes, terminated with 0 byte (C convention)

- Even start address .even | required for subsequent words/longs/programs
Motorola Jump and Branch

- After arithmetic or logic commands the **program counter** (PC) is always incremented to the next instruction
- Jump and branching instructions can be used to change the control flow

**Unconditional jump**
Change program counter to specified address
JMP label
[PC := label]

Example:
```
JMP next ; always jump
next: MOVE.L .. | destinat.
```

**Conditional branch**
Program counter is only changed if a condition is true
BEQ label
[if equal then PC := label]

Example:
```
CMP.L D1,D2 | compare regist.
BEQ next ; branch if equal
CLR.L D1 | else do this
next: MOVE.L ... | branch destin.
```

Atmel Jump and Branch

- After arithmetic or logic commands the **program counter** (PC) is always incremented to the next instruction
- Jump and branching instructions can be used to change the control flow

**Unconditional jump**
Change program counter to specified address
JMP label
[PC := label]

Example:
```
JMP next ; always jump
next: ADD .. ; destination
```

**Conditional branch**
Program counter is only changed if a condition is true
BREQ label
[if equal then PC := label]

Example:
```
CMP R1,R2 ; compare regist.
BREQ next ; branch if equal
CLR R7 ; else do this
next: ADD ... ; destination
```

Motorola Branch Commands

- Branching can be made on several different **conditions**
- **Conditions** are the result of the previous arithmetic or logic operation and are stored in the condition code register
- Most important conditions:
  - Z  zero
  - M  minus (negative number)
  - C  carry (e.g. in addition)
- Most important branches:
  - BEQ branch on equal (0)
  - BNE branch on not equal
  - BGT branch on greater than
  - BGE branch on greater or equal
  - BLT branch on less than
  - BLE branch on less or equal
  - BMI branch on minus
  - BPL branch on plus
  - BCS branch on carry set
  - BCC branch on carry clear
- if condition then PC := label
- else PC := PC + inc. (next instr.)

Atmel Branch Commands

- Branching can be made on several different **conditions**
- **Conditions** are the result of the previous arithmetic or logic operation and are stored in the condition code register
- Most important conditions:
  - Z  zero
  - M  minus (negative number)
  - C  carry (e.g. in addition)
- Most important branches:
  - BREQ branch on equal (0)
  - BRNE branch on not equal
  - BRGE branch on gr. or equal (signed)
  - BRLT branch on less than (signed)
  - BRPL branch on plus
  - BRMI branch on minus
  - BRCS branch on carry set
  - BRCC branch on carry clear
- if condition then PC := label
- else PC := PC + inc. (next instr.)
Motorola Compare and Test

- Condition codes are set **automatically** by arithmetic or logic commands (e.g. ADD, SUB, MOVE, AND, OR, NOT, ...)
- Condition codes can also be set **explicitly** by a number of compare and test commands:
  - **CMP.L D2, D4** | compare two values, set condition codes accordingly
  - **TST.L D3** | test single value, set Zero and Minus condition flags
  - **BTST.L #16, D5** | bit test single bit in a register [specify bit position 31 (MSB) .. 0 (LSB)], set zero flag if tested bit is zero
  - **CMP2.L low,D1** | double compare if low ≤ D1 ≤ high
    - set carry flag (and zero flag) accordingly
  - **low: DC.L 0**
  - **high: DC.L 9**

Motorola Branch Examples

- **CMP.L #5, D1** | compare register with constant
- **BGT label1** | branch to label 1 if D1 > 5
- **SUB.L D5, D6** | subtract two registers
- **BMI label2** | branch if result is negative
- **MOVE.L D0, D7** | copy data
- **BEQ label3** | branch if D7 = 0
- **TST.L D7** | set condition codes for D7
- **BEQ label3** | branch if D7 = 0
- **CMP2.L low,D1** | compare register against 2 bounds
- **BCS out** | branch if NOT (low ≤ D1 ≤ high)
  - **low: DC.L 0**
  - **high: DC.L 9**

Subroutines

- Command **JSR** (jump subroutine) is similar to JMP, but preserves current program counter on stack, so execution can return later using **RTS** (return from subroutine)
  - **main: JSR sub1** | CALL sub1
  - **sub1: ...**
  - **RTS**
- Most subroutines have to pass parameters
  - e.g. like math function: \( z := f(x,y) \)
- Passing parameters in registers is one method

Motorola Subroutine Example

Compute: \( f(x) = |x| \) absolute value
Assume: Input value in D1
Output value in D0

Subroutine: | move data (also tests D0 vs. 0)
- **abs:** | done if input value is positive
- **PFL done** | else (value is neg.), negate it
- **NEG.L D0** | negate it
- **done:** | RTS

Call from main program:
- **main: MOVE.L #5, D1** | input value
- **JSR abs** | call subroutine
- **...**
- **MOVE.L #-5, D1** | input value
- **JSR abs** | call subroutine
**Atmel Subroutine Example**

Compute: \( f(x) = |x| \) absolute value  
Assume: Input value in R1  
Output value in R0

Subroutine:
```
abs:
  MOVE R0, R1 ; move data (also tests R1 vs. 0)
  BRPL done ; done if input value is positive
  NEG R0 ; else (value is neg.), negate it
done:
  RTS
```

Call from main program:
```
main:
  LDI R1,5 ; input value
  CALL abs       ; call subroutine
  ...LDI R1,-5     ; input value CALL abs       ; call subroutine
```

**A.4 Addressing Modes**

All assembly commands need operands:
- constant
- register
- memory

**IMMEDIATE** constant  
**Mot:** ADD.L #17, D0  
**Atm:** ADI1 W, H, L, 17

**DIRECT** register  
**Mot:** ADD.D R1, D0 ; R0 := R0 + R1

**INDIRECT** memory referenced via register  
**Mot:** ADD.L (A1), D0 ; D0 := D0 + Mem[A1]

**ABSOLUTE** memory reference  
**Mot:** ADD.L name, D0 ; D0 := D0 + Mem[name]  
**Atm:** LDS R0, name ; R0 := Mem[name]  
(no add)

**Advanced Addressing Modes**

**INDIRECT WITH POSTINCREMENT OR PREDECREMENT**  
ADD.L (A5)+, D0  
ADD.L (A5)-, D0  
- very useful for tables  
- similar to C: i++, --i

**INDIRECT WITH DISPLACEMENT (16 bit)**  
ADD.L (20, A5), D0  
D0 := D0 + Mem[A5+20]

**INDIRECT WITH INDEX AND DISPLACEMENT (8 bit or 32 bit)**  
ADD.L (10, A5, D1.L*8), D0  
**scale factor:** 1, 2, 4, 8

- all added up to determine address  
- not necessarily always so complex, e.g.:  
  ADD.L (A5, D1.L), D0

- very helpful for array processing
Advanced Addressing Modes

**PROGRAM COUNTER INDIRECT (8 bit, 16 bit, or 32 bit)**

- `ADD.L (0x10FF, PC), D0` | program counter w. displacem.
- `ADD.L (50, PC, D2.L*8), D0` | PC with index and displacem.

- data is referenced with respect to program counter
- can point to program or data area
- allows generation of **relocatable** code
  → program can be moved to other memory area without re-assembly

---

A.5 Stack

- A data structure with elements of the same type
- Variable size
- Can be implemented with an array
- Adding new elements and deleting existing elements on the **same side**
- Typical application: **Subroutines**
  → LIFO (last in - first out)

---

Stack Implementation using Array

- Use array of fixed size
- Use address register as Stack-Pointer
- Simpler than list!

- Stack is defined as array elements between Bottom-of-Stack (fixed) and Stack-Pointer (SP), all other array elements are **empty**
- **SP is decremented** if new element is inserted
- **SP is incremented** if existing element is deleted

---

Motorola Stacks in use for Subroutines

- A stack is required for every subroutine call.

Close up look into operation of JSR and RTS:

- **JSR address**
  1. `SP ← SP - 4`
  2. `(SP) ← PC`
  3. `PC ← address`

- **RTS**
  1. `PC ← (SP)`
  2. `SP ← SP + 4`
Atmel Stacks in use for Subroutines

- A stack is required for every subroutine call.

Close up look into operation of JSR and RTS:

```
CALL address
1. (SP) ← PC + 2
2. SP ← SP - 2
3. PC ← address
```

```
RTS
1. SP ← SP + 2
2. PC ← (SP)
```

Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>PC</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>$30000</td>
<td>JSR</td>
<td>$30000</td>
<td>$30006</td>
</tr>
<tr>
<td>$30006</td>
<td>calc</td>
<td></td>
<td>push</td>
</tr>
<tr>
<td>$40000</td>
<td></td>
<td>$40000</td>
<td>$40000</td>
</tr>
<tr>
<td>$40100</td>
<td>RTS</td>
<td></td>
<td>pop</td>
</tr>
</tbody>
</table>

Subroutines and Stack

- It is now obvious that a storage place is needed for the return address
- But why a stack?
  Wouldn’t a single memory space be sufficient?
- Example:

```
main: ...

sub1: ...
JSR sub1 ...

sub2: ...
JSR sub2 ...

sub3: ...
JSR sub3 ...

RTS ...
```

Further Use of Stack:

- Save and restore registers within subroutines.

A common problem:

```
MOVE.L ____, D5
JSR xyz
MOVE.L D5, ___
```

→ Good approach: Save and restore registers.
Subroutines and Stack

Example: move multiple

sub: MOVEM.L D0-D7/A0-A6, -(SP) | Save ALL registers
     ; register contents changes
     MOVEM.L (SP)+, D0-D7/A0-A6 | Restore ALL registers
     RTS

If required save/restore condition codes as well:

sub: MOVE.W CCR, -(SP)
     MOVEM.L D0-D7/A0-A6, -(SP) | Save
     MOVEM.L (SP)+, D0-D7/A0-A6 | Restore
     RTS

A.6 Parameter Passing

Methods for parameter passing to subroutines:

1) Registers → + Fast
   - Maximum of 7 values and 1 result (4 Bytes)
2) Memory → + No size limit: pass an address/pointer
to global memory (e.g. start address of block)
   - Recursive call not supported
3) Inline → Store parameters in program immediately
   after JSR-command (PC Relative)
   - Needs special return routine, not RTS
   - Does not work in ROM
4) Stack → Most common and most versatile solution

Motorola Parameter Passing: Registers

main: MOVE.L #21, D1
       MOVE.L #47, D2
       JSR add
       ; result is now in D0
       RTS

add:   MOVE.L D1, D0
        ADD.L D2, D0
        RTS

Atmel Parameter Passing: Registers

main: LDI R1,21
       LDI R2,47
       CALL add
       ; result is now in R0
       RTS

add:   MOV R0, R1
        ADD R0, R2
        RTS
**Motorola Parameter Passing: Stack**

- Use existing system stack
- In calling routine:
  1. Push parameters onto stack
     - MOVE.L xxx, -(SP)
  2. JSR
  3. Pop parameters from stack
     - ADD.L #yy, SP

- In called routine:
  1. Read parameters from stack with offset from stack pointer
     - MOVE.L zz(SP), ddd

---

**Atmel Parameter Passing: Stack**

- Use existing system stack
- In calling routine:
  1. Push parameters onto stack
  2. CALL
  3. Pop parameters from stack

- In called routine:
  1. Read parameters from stack with offset from stack pointer

---

**Motorola Parameter Passing: Stack**

```
main:    MOVE.L #21, -(SP)
         MOVE.L #47, -(SP)
         JSR add
         ADD.L #8, SP ; pop 2*4 for param.
         ; result is now in D0
         RTS

add:     MOVE.L 4(SP), D0 ; offset 4 for ret. add.
          ADD.L 8(SP), D0
          RTS
```

---

**Atmel Parameter Passing: Stack**

```
main:    LDI R1,21 ; push first value
          PUSH
          LDI R1,47 ; push second value
          PUSH
          CALL add
          POP POP
          ; result is now in R0
          RTS

add:     LD  R0, ??
          ADD R0, ??
          RTS
```
Parameter Passing: Stack

In subroutine:
(1) Read parameters from stack with offset from stack pointer.

Please note: subroutine cannot easily pop parameters itself, since stack looks like:

```
parameters
return address
... SP
```

A.7 Motorola Complete Example

```
.include "eyebot.i"
.section .text.globl main
main: MOVE.L #16, D6            | print 16 characters
    labels loop
label: MOVE.L #'*', -(SP)      | load char to be printed
    JSR LCDPutChar             | call system LCD routine
    ADD.L #4, SP               | take 1 parameter off stack
    SUB.L #1, D6               | decrement counter
    BNE label                  | while not finished (?0) go to loop
    MOVE.L #string, -(SP)      | load string start address
    JSR LCDPutString           | call system LCD routine
    ADD.L #4, SP               | take 1 parameter off stack
    RTS                        | return to operating system
string: .asciz "Hello"
```

A.7 Atmel Complete Example

```
.include "eyebot.i"
.section .text.globl main
main: MOVE.L #16, D6            | print 16 characters
    labels loop
label: MOVE.L #'*', -(SP)      | load char to be printed
    JSR LCDPutChar             | call system LCD routine
    ADD.L #4, SP               | take 1 parameter off stack
    SUB.L #1, D6               | decrement counter
    BNE label                  | while not finished (?0) go to loop
    MOVE.L #string, -(SP)      | load string start address
    JSR LCDPutString           | call system LCD routine
    ADD.L #4, SP               | take 1 parameter off stack
    RTS                        | return to operating system
string: .asciz "Hello"
```
Motorola Complete Example

- Previous example contained a counting loop
- Very basic and very useful construct

\[
\text{MOVE.L } \#16, D6 \quad | \quad \text{init counter} \\
\text{loop: } ... \quad | \quad \text{loop contents} \\
\text{SUB.L } \#1, D6 \quad | \quad \text{dec. counter} \\
\ldots \text{BNE loop} \quad | \quad \text{branch if } \neq 0 \\
\ldots \quad | \quad \text{after loop}
\]

- Counter (here D6) **must not be changed** inside loop contents e.g. by subroutine call

Atmel Complete Example

- Previous example contained a counting loop
- Very basic and very useful construct

\[
\text{LDI R6,16} \quad ; \quad \text{init counter} \\
\text{loop: } ... \quad ; \quad \text{loop contents} \\
\text{SUBI R6,1} \quad ; \quad \text{dec. counter} \\
\ldots \text{BRNE loop} \quad ; \quad \text{branch if } \neq 0 \\
\ldots \quad ; \quad \text{after loop}
\]

- Counter (here R6) **must not be changed** inside loop contents e.g. by subroutine call