CPE/EE 421 Microcomputers

Instructor: Dr Aleksandar Milenkovic
Lecture Notes S03

*Material used is in part developed by Dr. D. Raskovic and Dr. E. Jovanov

Review: Assembler Directives

Addressing Modes

Addressing modes are concerned with how the CPU accesses the operands used by its instructions

Register Transfer Language (RTL)

Unambiguous notation to describe information manipulation

Registers are denoted by their names (eg. D1-D7, A0-A7)

Square brackets mean "the contents of"

Base number noted by a prefix (%-binary, $-hex)

Backward arrow indicates a transfer of information (←)

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ai</td>
<td>Address register i (i = 0 to 7)</td>
</tr>
<tr>
<td>Di</td>
<td>Data register i (i = 0 to 7)</td>
</tr>
<tr>
<td>Xi</td>
<td>General register i</td>
</tr>
<tr>
<td>[M]</td>
<td>The contents of memory location M</td>
</tr>
<tr>
<td>[X]</td>
<td>The contents of register X</td>
</tr>
<tr>
<td>[Di(0:7)]</td>
<td>Bits 0 to 7 inclusive of register Di</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Enclose a parameter required by an expression</td>
</tr>
<tr>
<td>ea</td>
<td>The effective address of an operand</td>
</tr>
<tr>
<td>{M}</td>
<td>The contents of a memory location specified by ea</td>
</tr>
<tr>
<td>d8</td>
<td>An 8-bit signed offset (-128 to 127)</td>
</tr>
<tr>
<td>d16</td>
<td>A 16-bit signed offset (-32K to 32K -1)</td>
</tr>
<tr>
<td>d32</td>
<td>A 32-bit signed offset (-2G to 2G -1)</td>
</tr>
</tbody>
</table>

ADD <source>,<destination>

MOVE <source>,<destination>

[destination] = [source] + [destination]
Register Direct Addressing

Register direct addressing is the simplest addressing mode in which the source or destination of an operand is a data register or an address register. The contents of the specified source register provide the source operand. Similarly, if a register is a destination operand, it is loaded with the value specified by the instruction. The following examples all use register direct addressing for source and destination operands.

- **MOVE.B D0, D1**
  - The instruction indicates the data register
  - The destination operand is data register D1
  - The effect of this instruction is to copy the contents of data register D0 in to data register D1

Immediate Addressing

- In *immediate addressing* the actual operand forms part of the instruction. An immediate operand is also called a literal operand. Immediate addressing can be used only to specify a source operand.
- Immediate addressing is indicated by a # symbol in front of the source operand.
- For example, **MOVE.B #24, D0** uses the immediate source operand 24.
Immediate Addressing

The instruction MOVE.B #4,D0 uses a literal source operand and a register direct destination operand.

- **The literal source operand**, 4, is part of the instruction.
- The destination operand is a data register.

The effect of this instruction is to copy the literal value 4 to data register D0.

Immediate Addressing Example

- Typical application is in setting up control loops:
  
  ```
  for(i=0; i<128; i++)
  A(i) = 0xFF;
  ```

- 68000 assembly language implementation:

  ```
  MOVE.L #001000,A6  Load A6 with the address of the array
  MOVE.B #128, D0   D0 is the element counter
  LOOP MOVE.B #1FF,(A6)+ Store FF in this elem. and incr. pointer
                 DSEQ.B #1, D0   Decrement element counter
                 BNE LOOP        Repeat until all the elements are set
  ```

Direct (or Absolute) Addressing

- In direct or absolute addressing, the instruction provides the address of the operand in memory.
- Direct addressing requires two memory accesses. The first is to access the instruction and the second is to access the actual operand.
- For example, CLR.B 1234 clears the contents of memory location 1234.
Direct (or Absolute) Addressing

This instruction has a direct source operand

The source operand is in memory

The destination operand uses data register direct addressing

The address of the operand forms part of the instruction

Once the CPU has read the operand address from the instruction, the CPU accesses the actual operand.

The effect of MOVE.B 20,D0 is to read the contents of memory location 20 and copy them to D0

Problem: \[ D0 \leftarrow [M(1001)] + [D0] \] (or: \( D0 \leftarrow [M(1001)] + D0 \))

Instruction:

\[ \begin{array}{cccccccc}
    1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
\end{array} \]

Effective Address: \[ 0000 \ 1001 \]

Summary of Fundamental Addressing Modes

- Consider the high-level language example: \( Z = Y + 4 \)
- The following fragment of code implements this construct:

```
ORG $400  ; Start of code
MOVE.B Y, D0
ADD #4, D0
MOVE.B D0, Z

ORG $600  ; Start of data area
Y DC.B 27  ; Store the constant 27 in memory
Z DS.B  1  ; Reserve a byte for Z
```

Assembler:

```
ADD.B Y, D0
```
The Assembled Program

1 000000400 ORG $400
2 000000400 103900000600 MOVE.B Y, D0
3 000000406 06000018 ADD.B #24, D0
4 00000040A 13C000000601 MOVE.B D0, Z
5 000000410 48F2700 STOP #$2700
6 *
7 000000600 ORG $600
8 000000600 1B Z: DC.B 27
9 000000601 00000001 Y: DS.B 1
10 000000600 END $400

Memory Map of the Program

Y is a variable accessed via the direct address 000600
Z is a variable accessed via the direct address 000601
This is a literal operand stored as part of the instruction

Moving data from a 32-bit register to memory using the MOVEP instruction

Register direct addressing is used for variables that can be held in registers
Literal (immediate) addressing is used for constants that do not change
Direct (absolute) addressing is used for variables that reside in memory
The only difference between register direct addressing and direct addressing is that the former uses registers to store operands and the latter uses memory

Summary

The Assembled Program

1 000000400 ORG $400
2 000000400 103900000600 MOVE.B Y, D0
3 000000406 06000018 ADD.B #24, D0
4 00000040A 13C000000601 MOVE.B D0, Z
5 000000410 48F2700 STOP #$2700
6 *
7 000000600 ORG $600
8 000000600 1B Z: DC.B 27
9 000000601 00000001 Y: DS.B 1
10 000000600 END $400

Memory Map of the Program

Y is a variable accessed via the direct address 000600
Z is a variable accessed via the direct address 000601
This is a literal operand stored as part of the instruction
Moving data from a 32-bit register to memory using the MOVEP instruction

NOTE: The instruction takes 24 clock cycles to execute.

Address Register Indirect Addressing

- In address register indirect addressing, the instruction specifies one of the 68000’s address registers; for example, MOVE.B (A0), D0.
- The specified address register contains the address of the operand.
- The processor then accesses the operand pointed at by the address register.

Address Register Indirect Addressing

This instruction means load D0 with the contents of the location pointed at by address register A0.

The instruction specifies the source operand as (A0).

The address register in the instruction specifies an address register that holds the address of the operand.

Finally, the contents of the address register pointed at by A0 are copied to the data register.
Auto-incrementing

If the addressing mode is specified as (A0)+, the contents of the address register are incremented after they have been used.

Address register A0 is used to access memory location 1000 and the contents of this location (i.e., 57) are added to D0.

After the instruction has been executed, the contents of A0 are incremented to point at the next location.

Use of Address Register Indirect Addressing

The following fragment of code uses address register indirect addressing with post-incrementing to add together five numbers stored in consecutive memory locations.

```
MOVE.B #$05,D0     Five numbers to add
LEA Table,A0      A0 points at the numbers
CLR.B D1          Clear the sum
Loop ADD.B (A0)+,D1 REPEAT Add number to total
     SUB.B D0,1      UNTIL all numbers added
     BNE Loop
STOP #$2700

* Table DC.B 1,4,2,6,5 Some dummy data
```

We are now going to trace through part of this program, instruction by instruction.
Use of Address Register Indirect Addressing

This instruction adds the contents of the location pointed at by A0 to D1.

Because the operand was (A0)+, the contents of A0 are incremented.

ADD.B (A0)+, D1

On the next cycle, the instruction ADD.B (A0)+, D1 uses A0 as a source operand and then increments the contents of A0.

Problem

Identify the source addressing mode used by each of the following instructions.

ADD.B (A5),(A4)  Address register indirect addressing. The address of the source operand is in A5.

MOVE.B #12, D2  Literal addressing. The source operand is the literal value 12.

ADD.W TIME, D4  Memory direct addressing. The source operand is the contents of the memory location whose symbolic name is "TIME".

MOVE.B D6, D4  Data register direct. The source operand is the contents to D6.

MOVE.B (A6)+, TEST  Address register indirect with post-incrementing. The address of the source operand is in A6. The contents of A6 are incremented after the instruction.

Other ARI Addressing Modes

- Address Register Indirect with Predecrement Addressing

  MOVE.L - (A0), D3  (A0 is first decremented by 4)

  Combination: MOVE.B (A0)+, (A1)+

- Register Indirect with Displacement Addressing

  d16(Ai) RTL: ea=d16+[Ai]

- Register Indirect with Index Addressing

  d8(Ai,Xj,W) or d8(Ai,Xj,L) RTL: ea=d8+[Ai]+[Xj]
Summary
Addressing Modes
- Register direct addressing is used for variables that can be held in registers: \textit{ADD.B D1,D0}
- Literal (immediate) addressing is used for constants that do not change: \textit{ADD.B #24,D0}
- Direct (absolute) addressing is used for variables that reside in memory: \textit{ADD.B 1000,D0}
- Address Register Indirect: \textit{ADD.B (A0),D0}
- Autoincrement: \textit{ADD.B (A0)+,D0}

Address Register Indirect with Pre-decrement Addressing
\begin{itemize}
\item MOVE.L -(A0),D3 (A0 is first decremented by \textit{4})
\item Combination: MOVE.B (A0)+,(A1)+
\item MOVE.B -(A1),(A0)+
\end{itemize}

Address Register Indirect with Displacement Addressing
\begin{itemize}
\item d16(Ai) RTL: ea=d16+[Ai]
\end{itemize}

Address Register Indirect with Index Addressing
\begin{itemize}
\item d8(Ai,Xj.N) or d8(Ai,Xj.L) RTL: ea=d8+[Ai]+[Xj]
\end{itemize}

PC can be used only for SOURCE OPERANDS
\begin{itemize}
\item MOVE.B TABLE(PC),D2
\item \textit{TABLE DC.B Value1...Value2}
\end{itemize}

Stack Pointer
- First-in-last-out
- SP points to the element at the top of the stack
- Up to eight stacks simultaneously
- A7 used for subroutines
- A7 automatically adjusted by 2 or 4 for L or W ops.
- Push/pull implementation:
  \begin{itemize}
  \item MOVE.W Dn,−(A7) <-PUSH
  \item MOVE.W (A7)+,Dn <-PULL
  \item SSP/USP
  \end{itemize}

Data Movement Operations
- Copy information from source to destination
- Comprises 70% of the average program
\begin{itemize}
\item \textit{MOVE/MOVEA}
\item MOVE to CCR
  \begin{itemize}
  \item MOVE <ear>,CCR - word instruction
  \item MOVE to/from SR
  \end{itemize}
\item MOVE USP
  \begin{itemize}
  \item to/from User Stack Pointer
  \item MOVE USP,A3 - transfer the USP to A3
  \item MOVEQ
  \end{itemize}
\item MOVEM
  \begin{itemize}
  \item to/from multiple registers (W/L)
  \item MOVEM.D 00-05/A0-A5, - (A7)
  \item MOVEM.L (A7)+,00-05/A0-A5
  \end{itemize}
\item MOVEP
  \begin{itemize}
  \item Move Quick(8b #value to 32b reg)
  \end{itemize}

Important NOTE: The contents of the CC byte of the SR are updated after the execution of an instruction. Refer to Table 2.2
Data Movement Operations, cont'd

Moving data from a 32-bit register to memory using the \texttt{MOVEP} instruction

\textit{NOTE:} The instruction takes 24 clock cycles to execute.

<table>
<thead>
<tr>
<th>Address</th>
<th>Low</th>
<th>Medium</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>29+3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29+2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29+1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\textit{Bytes from the register are stored in every other memory byte.}

\textbf{LEA}

\textit{Calculates an effective address and loads it into an address register – LEA \texttt{<ea>,An}}

\textit{Can be used only with 32-bit operands}

\textbf{Example:}

\textbf{a)} \texttt{ORG \$9000}

\texttt{LEA TABLE1(PC),A5}

\begin{verbatim}
1 00009000 ORG \$9000
2 00009000 4BFA0FFE LEA TABLE1(PC),A5
\end{verbatim}

\textbf{b)} \texttt{LEA 6(A0,D6.W),A2}

\textit{EA = \$00009000 + 2 + \$0FFE = \$0000A000}

\textit{A5=\$0000A000, CC: Not affected (NA)}

\textit{EA = 6 + \$00007020 + \$0003 = \$00007029}

\textit{A2=\$00007029 CC: NA}

\textbf{PUSH EFFECTIVE ADDRESS (PEA)}

\textit{Calculates an effective address and pushes it onto the stack pointed at by A7 – PEA \texttt{<ea>}}

\textit{Can be used only with 32-bit operands}

\textbf{EXG}

\textit{Exchanges the entire 32-bit contents of two registers}

\textit{EXG Xi,Xj}

\textbf{SWAP}

\textit{Exchanges the upper- and lower-order words of a DATA register}

\textit{SWAP Di}

\textbf{DATA MOVEMENT OPERATIONS, CONT'D}

\textbf{INTEGER ARITHMETIC OPERATIONS}

\textit{Float-point operations not directly supported}

\textit{Except for division, multiplication, and if destination is \texttt{Ai}, all act on 8-, 16-, and 32-bit values}

\textit{ADD/ADDA} (no mem-to-mem additions, if destination is \texttt{Ai}, use ADDA)

\textit{ADDQ} (adds a small 3-bit literal quickly)

\textit{ADDI} (adds a literal value to the destination)

\textit{ADDX} (adds also the contents of X bit to the sum)

\textit{CLR} (clear specified data register or memory location) equivalent to \texttt{MOVE #0, <ea>}

\textit{for address registers use SUB.L An,An
Integer Arithmetic Operations, cont’d

- **DIVU/DIVS** – unsigned/2’s-complement numbers
  - DIVU <ea>,Dn or DIVS <ea>,Dn
  - 32-bit longword in Dn is divided by the 16-bit word at <ea>
  - 16-bit quotient is deposited in the lower-order word of Dn
  - The remainder is stored in the upper-order word of Dn
- **MULU/MULS** – unsigned/2’s-complement numbers
  - Low-order 16-bit word in Dn is multiplied by the 16-bit word at <ea>
  - 32-bit product is deposited in Dn
- **SUB, SUBA, SUBQ, SUBI, SUBX**
  - **NEG** – forms the 2’s complement of an operand
  - **NEGX** – Negate with Extend, used for multi-prec. arith.
- **EXT** – Sign Extend
  - EXT.W Dn copies bit 7 to bits 8-15
  - EXT.L Dn copies bit 15 to bits 16-31

BCD Arithmetic Operations

- Only 3 instructions support BCD
  - ABCD Di,Dj or ABCD –(Ai),-(Aj)
  - Add BCD with extend – adds two packed BCD digits together with X bit from the CCR
  - SBCD – similar
    - [destination]=-(destination)-[source]-[X]
  - NBCD <ea>, subtracts the specified operand from zero together with X bit and forms the 10’s complement of the operand if X =0, or 9’s complement if X =1
- **Involve X** because they are intended to be used in operations on a string of BCD digits

Logical Operations

- Standard AND, OR, EOR, and NOT
- Immediate operand versions: ANDI, ORI, EORI
- **AND** a bit with 0 – mask
- **OR** a bit with 1 – set
- **EOR** a bit with 1 – toggle

- Logical operations affect the CCR in the same way as MOVE instructions

Shift Operations

- **Logical Shift**
  - **LSL** – Logical Shift Left
  - **LSR** – Logical Shift Right

Shift Operations, cont’d

- **Arithmetic Shift**
  - **ASL** – Arithmetic Shift Left
  - **ASR** – Arithmetic Shift Right

Shift Operations, cont’d

- **Rotate**
  - **ROL** – Rotate Left
  - **ROR** – Rotate Right
Shift Operations, cont’d

- Rotate Through Extend
  - ROXL – Rotate Left Through Extend
  - ROXR – Rotate Right Through Extend

Effect of the Shift Instructions

<table>
<thead>
<tr>
<th>Initial Value</th>
<th>After First Shift</th>
<th>CCR</th>
<th>After Second Shift</th>
<th>CCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>11010111</td>
<td>1100101010</td>
<td></td>
<td>1100101100</td>
<td>11001</td>
</tr>
<tr>
<td>11011110</td>
<td>0011111111</td>
<td></td>
<td>000000111111</td>
<td>10001</td>
</tr>
<tr>
<td>11010101</td>
<td>1100101100</td>
<td></td>
<td>1100101100</td>
<td>11001</td>
</tr>
<tr>
<td>11011110</td>
<td>1111111000</td>
<td></td>
<td>1111111000</td>
<td>11001</td>
</tr>
<tr>
<td>11010101</td>
<td>010000111111</td>
<td></td>
<td>000000011111</td>
<td>10001</td>
</tr>
<tr>
<td>11011110</td>
<td>001111111111</td>
<td></td>
<td>000000011111</td>
<td>10001</td>
</tr>
<tr>
<td>11010101</td>
<td>1100101000</td>
<td></td>
<td>1100101000</td>
<td>11001</td>
</tr>
<tr>
<td>11011110</td>
<td>011111100000</td>
<td></td>
<td>000000011111</td>
<td>10001</td>
</tr>
<tr>
<td>11010101</td>
<td>1111111000</td>
<td></td>
<td>1111111000</td>
<td>11001</td>
</tr>
<tr>
<td>11011110</td>
<td>001111111111</td>
<td></td>
<td>000000011111</td>
<td>10001</td>
</tr>
</tbody>
</table>

Forms of Shift Operations

- **Mode 1**
  - ASL Dx,Dy
  - Shift dy by dx bits

- **Mode 2**
  - ASL #<data>,Dy
  - Shift Dy by #data bits

- **Mode 3**
  - ASL <ea>
  - Shift the contents at the effective address by one place

All three modes apply to all eight shift instructions

Bit Manipulation Operations

- Act on a single bit of an operand:
  1. The complement of the selected bit is moved to the Z bit (Z set if specified bit is zero)
  2. The bit is either unchanged, set, cleared, or toggled
- NVCX bits are not affected
- May be applied to a bit within byte or longword
- BTST – Bit Test only
- BSET – Bit Test and Set (specified bit set)
- BCLR – Bit Test and Clear (specified bit cleared)
- BCHG – Bit Test and Change (specified bit toggled)

Program Control Operations

- Examine bits in CCR and choose between two courses of action
- CCR bits are either:
  - Updated after certain instruction have been executed, or
  - Explicitly updated (bit test, compare, or test instructions)
- **Compare instructions**: CMP, CMPA, CMPI, CMPM
  - Subtract the contents of one register (or mem. location) from another register (or mem. location)
  - Update NVC bits of the CCR
  - x bit of the CCR is unaffected
  - The result of subtraction is ignored
Program Control Operations, cont’d

- **CMP**: CMP <ea1>,<ea2>
  
  ![Equation](\text{CMP \ <ea1>,<ea2>})

- **CMPI**: CMP #<data>,<ea1>
  
  compares with a literal

- **CMPA**: CMP <ea1>,An
  
  used for addresses, operates only on word and longword operands

- **CMPPM**: CMP (Ai)+,(Aj)+
  
  compares memory with memory, one of few that works only with operands located in memory

- **TST**: TST <ea1>
  
  zero is subtracted from specified operand; N and Z are set accordingly, V and C are cleared, X is unchanged

- Except CMPPM, all take byte, word, or longword operands

Program Control Operations, cont’d

- **BRANCH CONDITIONALLY**
  
  Bcc <label>
  
  ![Equation](\text{Bcc \ <label>})

  - cc stands for one of 14 logical conditions (Table 2.4)
  - Automatically calculated displacement can be d8 or d16
  - Displacement is 2’s complement signed number
  - 8-bit displacement can be forced by adding .S extension
  - ZNCV bits are used to decide

Program Control Operations, cont’d

- **BRANCH UNCONDITIONALLY**
  
  BRA <label> or JMP (An)

  ![Equation](\text{BRA \ <label> \ or \ JMP \ (An)})

  - JMP d16(An)
  - JMP d8(An,Xi)
  - JMP Absolute_address
  - JMP d16(PC)
  - JMP d8(PC,Xi)

- **TEST CONDITION, DECREMENT, and BRANCH**

  DBcc Dn,<label> (16 bit displacement only)

  ![Equation](\text{DBcc \ Dn,<label> \ (16 \ bit \ displacement \ only)})

  - If test is TRUE, branch is NOT taken.
  - If test is NOT TRUE, Dn is decremented by 1.
  - If Dn is now equal to –1 next instruction is executed
  - If not, branch to <label is taken>

Subroutines

- **BRANCH TO SUBROUTINE**

  BSR <label> = \[A7\] ← \[A7\] - 4

  ![Equation](\text{BSR \ <label> \ = \[A7\] ← \[A7\] - 4})

  
  M(\[A7\]) ← [PC]

  ![Equation](\text{M(\[A7\]) ← [PC]})

  
  [PC] ← [PC] + d8

  ![Equation](\text{[PC] ← [PC] + d8})

- **RETURN FROM SUBROUTINE**

  RTS = \[PC\] ← M([A7+1])

  ![Equation](\text{RTS \ = \[PC\] ← M([A7+1])})

  
  [A7] ← [A7] + 4

Subroutines, cont’d

- **BRANCH TO SUBROUTINE**

  000FFA 41F900040000 LEA TABLE, A0

  001000 61000206 NextChr BSR GetChar

  001004 40C0 MOV.L B0, (A0)

  001006 0200000D CMP.B #$0D,D0

  00100A 66F4 BNE NextChr

  001102 61000104 BSR GetChr

  001106 02000051 CMP.B #’Q’,D0

  00110A 67000EF4 BEQ QUIT

  001208 1239000080000GetChr MOVE.B ACIA,D0

Subroutines, cont’d

- **BRANCH TO SUBROUTINE**

  BSR d8 = \[PC\] ← M([A7+1])

  ![Equation](\text{BSR \ d8 \ = \[PC\] ← M([A7+1])})

  
  [A7] ← [A7] + 4

  ![Equation](\text{[A7] ← [A7] + 4})

  current PC value

  ![Equation](\text{current \ PC \ value})

  d8 = \$00001028 - (\$00001000 + 2) = \$00000206

  ![Equation](\text{d8 = \$00001028 - (\$00001000 + 2) = \$00000206})

Nested Subroutines
Nested Subroutines, cont’d

Returning directly to a higher-level subroutine
Sub2
BEQ Exit
RTS
Exit LEA 4(A7),A7

RTR (Return and restore condition codes)
• Save the condition code register on the stack:
  MOVE CCR,-(A7)
• Use RTR instead of RTS

Miscellaneous Instructions

• Scc: Set byte conditionally
  Scc <ea> (cc same as in DBcc)
  If the condition is TRUE, all the bits of the byte specified by <ea> are SET, if the condition is FALSE, bits are CLEARED

• NOP: No Operation

• RTS: Return from Subroutine

• STOP:
  STOP n
  Stop and load n into Status Register; n is 16-bit number; Privileged instruction

• CHK, RESET, RTE, TAS, TRAPV - later

Example: Linked List

Adding an element to the end of a linked list
• HEAD points to the first element, NEW contains the address of the new item to be inserted
• Longwords

Example: Linked List, cont’d

Initial linked list:

Example: Linked List , cont’d

Linked list after inserting an element at the end:
### Example: Linked List, Memory Map

<table>
<thead>
<tr>
<th>Memory map of linked list before inserting an element</th>
<th>Memory map of linked list after inserting an element</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001000 0001000</td>
<td>00003000 00001000</td>
</tr>
<tr>
<td>00001000 0001000</td>
<td>00003000 00001000</td>
</tr>
<tr>
<td>00001000 0001000</td>
<td>00003000 00001000</td>
</tr>
<tr>
<td>00001000 0001000</td>
<td>00003000 00001000</td>
</tr>
<tr>
<td>00001000 0001000</td>
<td>00003000 00001000</td>
</tr>
<tr>
<td>00001000 0001000</td>
<td>00003000 00001000</td>
</tr>
<tr>
<td>00001000 0001000</td>
<td>00003000 00001000</td>
</tr>
</tbody>
</table>

Note: The shaded memory elements represent data values

---

Alex Milenkovich