Micro-Code Class

John Providenza

1. **Overview** A micro-code instruction consists of four fields:
   - Data Bus
   - Alu Control
   - Sequencer Control
   - Hardware Control

By convention, the fields are ordered:

HWCTL Alu Seq Data-Bus

The function of each is:

- *HW CTL* Control miscellaneous hardware devices such as counters, registers, etc. This field changes the most among the graphics controllers (Omega 400, Lambda, Omega 500). These bits are used to control pixel writing, pixel address counters (px and py), scratch pad ram, etc. A typical hardware control field may look like: *LPCNTEN MEMWR YDWN.*
- *Alu* This field controls the 2901 Alu's. There is a direct correlation between the opcodes for this field and the 2901 opcodes as described in the AMD data book. A typical ALU field is: *Dec B,R5,R3 Ldb.A*
- *Sequencer* This field controls the 2909/2911 sequencer chips that implement our program counter (PC). This field specifies branches, subroutine branch, return, four-way branch, etc. A sequencer instruction may be: *Br R,NLPC*
- *Data Bus* This field controls the data bus by specifying the source and destination devices that are driving/receiving the data bus. If the pipeline register is used to specify immediate data, this field supplies the data. A data bus instruction is like: *Dbus WDATLD,Aluy*

Incidently, the assemblers are: *Omac* for the Omega 400, *Lmac* for the Lambda, and *Ormac* for the Omega 500. Do a *man umac* for a rough synopsis of the flags, etc.

2. Data Bus This field controls the data bus (DB). If the field is missing from the instruction, the DB is idle with no source or destination. The DB data is always latched into the destination at the end of the current instruction.

The valid Omega 400 destinations are:

BOX,CENTER; CI S C C L L. Omega 400 Destinations Mnemonic Function _ cgenldLoad Ascii character to cgen cmapld Write color to color map crcrd Actually, a SOURCE. Read signature crtcld Load CRT controller register. crtcrd SOURCE. Read Crt controller register. ledld Load diagnostic LED's lpcntld Load loop counter nodestNo dest, usually used for Alu input odhld Really, IORESET odlld Write data to IO interface patrld Load pattern register. pxld Load PX counter (pixel X address) pyld Load PY counter (pixel Y address) rmskld Load read mask spadrld Load scratch pad ram address spdatld Write scratch pad ram data at current address spwrup Set power up flip-flop uartld Write data to gin interface uartrd SOURCE. Read data from gin interface wdatldLoad drawing color wmskld Load write mask xstld Load X starting address (pan) ystld Load Y

The Omega 400 data bus sources are:

starting address (pan) zoomld Load zoom factor BOX,CENTER; CI S C C L L. Omega 400 Bus Sources Mnemonic Function _ aluy 2901 Alu drives the bus extsrc Nop so that a "dest" can drive the bus imrdh Read the odd pixel from cache imrdl Read even pixel from cache inh Unused inl Read data from IO

The data bus field has 3 keywords associated with it: *Plreg*, *Dbus*, and *Const*. Some examples of these are:

interface spram Read data from scratch pad ram CENTER; L L. Dbus Wdatld,Aluy Write alu data to the color reg. Dbus PXLD,Plreg,5 Write a 5 to the PX register. Dbus PYLD,5 Write a 5 to the PY register. Const 33h Put a

The *Plreg* construct is a hang over from a previous assembler that was, if you can believe it, slower and less friendly than our current assembler.

3. Sequencer Control The sequencer implements the program counter for our bit slice engines. We use AMD 2909 and 2911 slices to create a 12 bit sequencer. Thus, we can directly address 4096 words of control store (instructions). The Omega 400 and the Lambda use the same sequencer instructions. The Omega 500 has been significantly enhanced to support looping and a few other bazzar features.

The sequencer also implements the subroutine return stack. Unfortunately, the 2909/2911 chips allow only 4 levels of subroutine call. Of course, there aren't any interrupts to worry about.

The legimate Omega 400 sequencer commands are:

Br D,CC1

Branch if CC1 is true. The branch address comes from the data bus, and is usually specified with a *Const Label* type of data bus control field. It is possible to branch to a location stored in scratch pad ram or the Alu. Some typical branch and data bus control instructions may look like:

33h on the bus (maybe for a jump). CENTER; L L. Br D,Z Const POLYF ; Branch

Br D

This is a degenerate case of the above. It is a branch always.

Br R,CC1

This is a conditional branch to the location that the R register points to. The R reg is in the sequencer chips, and, for the Lambda and 400, is loaded via the hardware control bits. A typical instruction stream may be:

to POLYF Br D,Pl Dbus Nodest,Aluy ; branch to address from the alu
CENTER; L L. LDR Const POLYO ; load R with address of POLYO Br R,Mi

Note that by using the R register, the data bus is left open for use during the branch.

Br R

This is the uncondition branch with R.

Bsr D,CC1

Branch to subroutine whose address is on the data bus. The next instruction's address is pushed on the stack, and be branch to the subroutine to resume execution.

Bsr D

This is the unconditional branch to subroutine via D.

Bsr R,CC1

Branch to subroutine whose address is in R.

Bsr R

This is the unconditional branch to subroutine via R.

Rts CC1

Return from subroutine if CC1 is true, else continue.

Rts

Unconditional return from subroutine.

Fwybr D,CC2,CC1

The fourway branch instruction is the most unique instruction in the Omega/Lambda display conrollers. It allows the program to test 2 condition codes simultaneously and branch to one of 4 locations. One restriction is that the target code must be *aligned* on a four byte boundary. A typical sequence of commands is:

Dbus Wdatld,Aluy ; Branch to POLYO while using data bus CENTER; L L L.
Inc A,R0 Fwybr D,LPC2,Z Const XYZ ; fourway branch to XYZ
Align 4; Force proper alignment XYZ: inst0 ; CC2 = false, CC1 = false inst1 ; CC2 = false, CC1 = true inst2 ; CC2 = true,

Fwybr R,CC2,CC1

Another fourway branch. We use the R reg for the destination address.

The legal CC1 condition codes on the Omega 400 are:

CC1 = false inst3 ; CC2 = true, CC1 = true BOX,CENTER; CI S L C L L. Omega 400 CC1's Mnemonic Test _ bmi byte minus bpl byte plus ca alu carry (no borrow) db0 DB0 = 1 idrdy input data ready iof 30/60 hz flag lpc loop counter carry lrc left alu register carry mi minus nca no alu carry (ie, borrow) ndb0 DB0 = 0 nidrdyinput data not ready niof not 60/30 Hz nlpc no loop counter carry nlrc no alu register left carry nodac output data not accepted by host nov no overflow nqrc no Q register right carry nrrc no alu register right carry nvbl not vertical blank nz not zero odac output data accepted ov alu overflow pl alu plus qrc alu Q register right carry rrc alu register right carry vbl vertical blank

The valid Omega 400 CC2 codes are:

z alu zero BOX,CENTER; CI S L C L L. Omega 400 CC2's Mnemonic Test _ ca2 Alu carry lpc2 loop counter carry nca2 no alu carry nlpc2 no loop counter carry nrrc2 no alu register right carry nz2 alu not zero rrc2 alu re-

Note that some of the condition codes are misnomers. For example, we don't check loop counter carry, we actually use the loop counter minus. Thus the 12 bit loop counter actually only can count 2047 events.

Micro-Code Class 2

Aug 31, 1983

John Providenza

1. ALU Control The Arithmetic Logic Unit (ALU) is the "brains" of the bit slice processor. Our ALU is based on the AMD 2901 4 bit processor slice. We use three slices to create our 12 bit ALU With a 12 bit processor, we can represent numbers in the range of -2048 to +2047, or, as unsigned integers, 0 to 4095. The 2901 contains 16 registers, an accumulator, and a logic unit.

The ALU control field can be broken into four parts:

- *Opcode* This field controls the action of the logic unit. The user specifies opcodes such as *Add*, *Exor*, or *Subr*.
- *Source Select* This field selects the two input data sources for the ALU.
- *Register Select* This field specifies the two addresses for the A and B registers.
- *Destination Control* This field selects where the output data from the Alu is to be put.

These fields are used to select what data the alu will operate on (via source select and register select), what the alu will do (via the opcode), and what is done with the data (via destination control and the register select). A typical instruction that uses all the fields is:

By convention, the result from of the Alu operation is called *F* and the chip output (if enabled as a data bus source) is called *Y*.

1.1. Source Select The Alu requires two sources of data, fondly known as the *R* and *S* busses. A two letter mnemonic is used to select the Alu sources for all of the native 2901 instructions (see derived instructions). The valid 2 letter source mnemonics are:

- *AB*
  Register data is used for the Alu inputs, the A reg data goes on the R inputs to the Alu, the B reg data to the S inputs.
- *AQ*
  Register and Accumulator data is used for the Alu inputs, the A reg data goes on the R inputs to the Alu, the Q reg data to the S inputs.
- *DA*
  Data Bus and Register data is used for the Alu inputs, the Data Bus goes on the R inputs to the Alu, the A reg data to the S inputs.
- *DQ*
  Data Bus and Accumulator data is used for the Alu inputs, the Data Bus goes on the R inputs to the Alu, the Q reg data to the S inputs.
- *DZ*
  Data Bus and Zero data is used for the Alu inputs, the Data Bus goes on the R inputs to the Alu, the S inputs are set to 0.
- *ZA*
  Zero and Register data is used for the Alu inputs, 0's are placed on the R inputs to the Alu, the A reg data to the S inputs.
- *ZB*
  Zero and Register data is used for the Alu inputs, 0's are placed on the R inputs to the Alu, the B reg data to the S inputs.

- *ZQ*
  Zero and Accumulator data is used for the Alu inputs, 0's are placed on the R inputs to the Alu, the Q reg data to the S inputs.

1.2. Alu Opcode The Alu Opcode specifies the operation that the Alu will perform. The R and S busses supply data to the Alu (see source select), and the Alu output may either be saved in a register, put on the data bus, or ignored (see destination control).

CENTER; l. L. Addi AB,R0,R1 Ldb.F; r0 + r1 + 1 -> r1 BOX,CENTER; CI S L C L L. Alu Opcode Table Mnemonic Function _add R + S addi R + S + 1 and R & S exnor ~(R ^ S) exor R ^ S notrs (~R) & S or R | S sub R - S subd R -

There are several more opcodes that will be described under ***derived opcodes*** since they are derived from the standard 2901 commands.

1.3. Register Select The 2901 contains 16 registers that are dual ported into the A and B ports. The register select field controls which of the A and B registers are used. Note that the A register is used only for read out, while the B register may be read, write, or both. Up to 2 registers may be specified. If two registers are specified, the 1st is the A reg, the 2nd is the B reg. If 1 register is specified, it selects the same A and B registers.

Register names are either R0 thru R15, or defined symbol that evaluates to a number in the range of 0 thru 15. Thus, all of the following are identical:

S - 1 subr S - R subrd S - R - 1 CENTER; L L L. P1X Equ 15; equate P1X to a 15
Temp Equ 0 ; equate Temp to 0 Add AB,P1X,R0 ; add r15 and r0
Add AB,R15,R0 ; add r15 and r0 Add AB,R15,Temp ; add r15

An *equate* or *defl* statement is frequently used to define a register name for clarity. Some common register equates are: *P1X*, *P1Y*, *P2X* and *P2Y*.

Some more examples:

and r0 CENTER; L L. Subr AB,R0,R1 ; r1 - r0 Exor AB,R1,R3 ; r1 ^ r3

1.4. Destination Control This is the hardest of the Alu fields to understand. It controls the storage of the Alu result (F), as well as the data source for the 2901 output pins (Y). The valid destination keywords are:

Subd AB,R2,r15 ; r2 - r15 -1 CENTER,BOX; CI S C C L L. Destination Keywords Mnemonic Function _ldb.a F -> B, A -> Y ldb.f F -> B, Y ldq F -> Q, Y nold F -> Y ldb.dn F/2 -> B, F -> Y ldb.up 2F -> B, F -> Y ldbq.dn F/2 -> B,

If an Up or Down shift was requested, an additional keyword is required to specify the type of shift:

- ***arshft*** Arithmetic shift. For upshift, shift in a 0, for downshift, sign extend.
- ***rotate*** End Around Rotate. We don't tolerate any thru the carry nonsense.
- ***shift0*** Shift in a 0 for a up or down shift.
- ***shift1*** Shift in a 1 for a up or down shift.

Some typical Alu instructions may be:

Q/2 -> Q, F -> Y ldbq.up 2F -> B, 2Q -> Q, F -> Y CENTER; L L L. P1X Equ 0Ch P1Y Equ 13

```
      Add AB,R0,P1X Ldb.F       ; r0 + r12 -> r12        Add DA,P1Y,R5 Ldb.F
Const 9      ; r13 + 9 -> r5        Add AB,R0 Ldb.F     ; r0 * 2 -> R0        Add
AB,R0 Ldb.Up Shift0    ; r0 * 4 -> R0        Add ZA,R0,R1 Ldb.Up Shift0        ; r0
```

1.5. Derived Instructions Some interesting combinations of Alu sources and opcodes exist that the assembler supports as separate instructions. These *derived* opcodes are frequently used. Because many of them use the *Zero* Alu source as an inherent source, only A, B, D, or Q need to be specified in the source specification field. The derived opcodes are:

- *Inc* Add 1 to the source.
- *Dec* Subtract 1 from the source.
- *Pass* Pass the source thru unchanged.
- *Not* Complement the data.
- *Neg* A 2's complement.

Some examples of the derived opcodes with a possible native instruction:

```
* 2 -> R1 CENTER; L L.  Inc A,R0,R1 Ldb.F    ; R0 + 1 -> R1 Addi ZA,R0,R1 Ldb.F

Dec D,R3 Ldb.F Const 5      ; 5 - 1 -> R3 Subd DZ,R3 Ldb.F Const 5

Dec B,R7,R4 Ldb.A   ; R4 - 1 -> R4, R7 -> Output Subrd ZA,R7,R4 Ldb.A

Neg A,R0,R1 Ldb.F   ; -R0 -> R1 Sub ZA,R0,R1 Ldb.F

Not B,R4        ; ~R4 -> Output Exnor ZB,R4
```

One additional derived instruction exists for forcing a 0 out of the Alu. It is appropriately called *Clr* and does not require an alu source.

```
Pass D,R5 Ldb.F Const 123h        ; 123h -> R5 Add DZ,R5 Ldb.F' Const 123h
CENTER; L L.  Clr R5 Ldb.F  ; 0 -> R5 and Output And ZA,R5 Ldb.F
```

1.6. Useful Tricks Some clever tricks allow useful constants to be loaded into a register without using the data bus.

```
Clr R5,R6 Ldb.A      ; 0 -> R6, R5-> Output And ZA,R5,R6 Ldb.A CENTER; L L.  Clr
R0 Ldb.Dn Shift1; 800h -> R0 Clr R0 Ldb.Up Shift1; 1 -> R0 Subd AB,R0 Ldb.F    ;

0FFFh -> R0 Subd AB,R0 Ldb.Dn Shift0   ; 7FFh -> R0
```

Micro-Code Class 5

Oct 5, 1983

John Providenza

2. Hardware Control Bits The Hardware Control Bits (HW Bits) are used to control various I/O devices/registers/counters... in the display controllers.

bpt

This bit is used for hardware debug. If the debug jumper on page 13 of the schematics is inserted, the Omega 400 will halt before each instruction. The single step switch may be used to step instruction by instruction. Since this bit is only asserted at the begining of the opcode fetch, it can be viewed as a FETCH signal.

cgencnt
: Step to the next row in the character rom. Used in drawing text.

cgenshft
: Step the character generator to the next pixel in the current row. Used in drawing text.

crcstop
: Same functionality as **bpt.**

crcstrt
: This asserts the currently spare bit in the hardware field.

ldr
: Load the sequencer R register. See the section on the 2909/2911 sequencer.

lpcnten
: Count the loop-counter.

memwr
: Write the pixel that is currently addresses by Px and Py. The color is in the write data register, and the write mask is used to protect bits in the pixel.

rdbckld
: Load 16 pixels into the ***readback shift register***. This is a 16 pixel cache that is **very** useful for many of our algorithms.

rdbcksl
: Shift the data in the readback shift register one pixel left. This gets you ready to read the next pixel on the right.

rdbcksr
: Shift the data in the readback shift register one pixel right. This gets you ready to read the next pixel on the left.

wall
: Write all 16 pixels in the current word addressed by Px and Py. The data comes from the write data register, and the write mask still works. The pixels written are at (Px & 03F0h, Py) thru (Px | 0Fh, Py). This instruction gives us our FLASH FILL capability.

xdwn
: Count the Px address down by 1.

xup Count the Px address up by 1.

ydwn
: Count the Py address down by 1.

yup
: Count the Py address up by 1.

Note that some of these Opcodes interact with each other. The following interact:

Thus a load or left shift of the readback shift register also causes the character generator to shift one.

Micro-Code Class 5

Oct 5, 1983

John Providenza

1. Hardware Control Bits The Hardware Control Bits (HW Bits) are used to control various I/O devices/registers/counters... in the display controllers.

bpt
: This bit is used for hardware debug. If the debug jumper on page 13 of the schematics is inserted, the Omega 400 will halt before each instruction. The single step switch may be used to step instruction by instruction. Since this bit is only asserted at the begining of the opcode fetch, it can be viewed as a FETCH signal.

cgencnt
: Step to the next row in the character rom. Used in drawing text.

cgenshft
: Step the character generator to the next pixel in the current row. Used in drawing text.

crcstop
: Same functionality as **bpt.**

crcstrt
: This asserts the currently spare bit in the hardware field.

ldr
: Load the sequencer R register. See the section on the 2909/2911 sequencer.

lpcnten
: Count the loop-counter.

memwr
: Write the pixel that is currently addresses by Px and Py. The color is in the write data register, and the write mask is used to protect bits in the pixel.

rdbckld
: Load 16 pixels into the *readback shift register*. This is a 16 pixel cache that is **very** useful for many of our algorithms.

rdbcksl
: Shift the data in the readback shift register one pixel left. This gets you ready to read the next pixel on the right.

rdbcksr
: Shift the data in the readback shift register one pixel right. This gets you ready to read the next pixel on the left.

wall
: Write all 16 pixels in the current word addressed by Px and Py. The data comes from the write data register, and the write mask still works. The pixels written are at (Px & 03F0h, Py) thru (Px | 0Fh, Py). This instruction gives us our FLASH FILL capability.

xdwn
: Count the Px address down by 1.

xup Count the Px address up by 1.

ydwn
: Count the Py address down by 1.

yup
Count the Py address up by 1.

Note that some of these Opcodes interact with each other. The following interact:

Thus a load or left shift of the readback shift register also causes the character generator to shift one.