MICROCODE CLASS SCHEDULE

Monday, April 25:  200M (DMR - Overview, 2.56, DMR - Intro, N, O)
2:00  Hardware overview - Dan Davies
3:00  Booting - Roy Ogus
4:00  Microcode overview - Ev Neely

Tuesday, April 26:  200M (DMR - A, C, D, F, DMR - 2.1-2.34, 2.51, 2.54)
1:30  Registers, constants, shifting, rotating - Chuck Fay
3:30  Branches, dispatches - Ev Neely

Thursday, April 28:  200M (DMR - F, G, H, I, J, DMR - 2.35-2.37, 2.52, 2.53, 2.55, 3)
9:00  Memory and mapping - Amy Fasnacht
10:30  Mesa stack, LH, etc. - Jim Sandman

Friday, April 29:  (DMR - P)
9:30  Emulator details - Jim Sandman 100G
1:30  Burdock demo - Amy Fasnacht 200M
      Problem specification

Friday, May 5:  (DMR - O, DMR - 2.3.8)
10:00  Review of problem, timing constraints, LOOPHOLES - Amy Fasnacht
SU (Stack and U) registers

- also called just U registers
- slower source and destination than R registers
  - can’t be source operand to ALU arithmetic operations unless only the low byte or nibble is significant
  - can’t be destination for ALU arithmetic operations unless only the low nibble is significant
  - *can* be operand to ALU logical operations
  - *can* be destination for ALU logical operations
- 256 regs x 16 bits
- split into 16 "blocks" or "banks" of 16 registers each, due to overloading of rA microinstruction field
- block 0 contains the Mesa evaluation stack and can be addressed by stackP, a special 4-bit stack pointer register
**Q register**

- special purpose register for double length shifting with ALU output F bus; intended primarily for multiplication and division routines
- writable from the ALU output F bus
  - but not when A-bypass is used
  - not simultaneously with a R register write
  - not shifted
- written back to itself shifted left or right one bit as part of a double length shift
- 1 reg x 16 bits
- reserved for use by emulator in Mesa machines
- can be used by emulator as a general temporary
- frequently used to update RH registers
- can be used as input to ALU
**RH registers — examples**

- **Naming**
  
  RegDef[rhT, RH, 1]; \{Note: T = 1\}
  
  RegDef[rhTT, RH, 2]; \{Note: TT = 2\}
  
  RegDef[RHD1, RH, 7]; \{Note: RD1 = 7\}
  
  RegDef[RHD2, RH, 8]; \{Note: RD2 = 8\}

- **Usage examples**
  
  Q ← rhTT + 1, LOOPHOLE[byteTiming] \(,c1;\)
  
  rhTT ← Q LRot0 \(,c2;\)
  
  RD1 ← RD1 + RD2, RHD1 ← RD2 LRot8 \(,c3;\)
  
  RHD2 ← TOS LRot0 \(,c1;\)
  
  rhT ← 0 \(,c2;\)
RH registers

- 8-bit "extensions" of R registers
- 16 regs x 8 bits
- always addressed by $rB$ microinstruction field; hence RH registers can only be written in the corresponding R register or the Q register
- typically combined with corresponding R register to form 24-bit virtual or 20-bit real addresses for memory references
- also can be used for flags, subroutine linkage, or just 8 bits of storage
- readable via the X bus (high byte of X bus is set to zero)
- writable via the X bus also (high byte of X bus is ignored)
R registers — examples

- Naming via RegDef[] macro (built into Mass)

  RegDef[TOS, R, 0];  \{Top of stack\}
  RegDef[PC, R, 5];  \{Program Counter\}
  RegDef[RD1, R, 7];  \{Disk reg 1\}
  RegDef[RD2, R, 8];  \{Disk reg 2\}

- Usage examples

  RD2 ← 0AA                   ,c1;
  RD1 ← (RD2 LRot8) or RD2    \{RD1 ← 0AAAA\} ,c2;
  RD2 ← −AE                   \{RD2 ← 0FF2\} ,c3;

  RD1 ← RD1 + RD2 + 1         \{Cin = 1\}    ,c1;
  RD1 ← RD1 + RD2             \{Cin = 0\}    ,c2;
  RD1 ← RD1 − RD2             \{Cin = 1\}    ,c3;
  RD1 ← RD1 − RD2 − 1         \{Cin = 0\}    ,c1;
R registers

- high speed
- 16 regs x 16 bits in two-port register file
- any 2 of the 16 are readable per microinstruction via dual output ports A and B
- addressed by $rA$ and $rB$ microinstruction fields
- $rB$-designated register may be written in same microinstruction from the ALU output F bus
- may be shifted/rotated one bit left or right before writing
- fixed allocation among micro-tasks, defined in Dandelion.df:
  0–6: Mesa emulator (TOS, L, G, PC, and 3 temps)
  7–8: Disk task
  9–B: Display/Raven/MagTape
  C–D: Ethernet
  E: IOP
  F: IOP/Kernel
Registers

R registers
- high speed (located inside 2901 chips)
- 16 regs x 16 bits

RH registers
- 8-bit extensions to the R registers for memory addressing
- 16 regs x 8 bits

Q register
- high speed (located inside 2901 chips)
- special purpose register for double length shifts
- Quotient register for division; also used for multiplication
- 1 reg x 16 bits

SU (Stack and U) registers
- lower speed than R registers
- 256 regs x 16 bits

L (Link) registers
- used for subroutine linkage
- 8 regs x 4 bits
XEROX

Dandelion CP Microcode Class

Registers, Constants, and Shifting

Chuck Fay
April 26, 1983
Palo Alto, California
SU (Stack and U) registers – continued

• fixed allocation among micro-tasks (with a few exceptions), defined in Dandelion.df:
  - block 0: Mesa emulator stack
  - blocks 1–6: Mesa emulator
  - blocks 7–8: Disk
  - blocks 9–B: Display/Raven/MagTape/Emulator
  - blocks C–D: Ethernet/LSEP
  - block E: Ethernet/LSEP/Emulator/IOP
  - block F: Kernel/Emulator/IOP
• EnSU microinstruction field enables SU access; Cin field determines read or write
  - Cin = 0 implies read
  - Cin = 1 implies write
• Hence, some arithmetic operations are disallowed when reading and writing SU registers.
  Cin must be 0 for the SU read in the following two examples, yet the arithmetic implies Cin = 1
  - Xbus ← SU, B ← R + R + 1 {illegal}
  - Xbus ← SU, B ← R – R {illegal}
  Cin must be 1 for the SU write in the following two examples, yet the arithmetic implies Cin = 0
  - SU ← A, B ← R + R {illegal}
  - SU ← A, B ← R – R – 1 {illegal}
SU (Stack and U) registers — addressing

• Three ways to generate U register addresses:

<table>
<thead>
<tr>
<th>0</th>
<th>3</th>
<th>4</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>rA</td>
<td>fZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>stackP</td>
<td></td>
</tr>
<tr>
<td>rA</td>
<td></td>
<td>Y[12-15]</td>
<td></td>
</tr>
</tbody>
</table>

• Only one ALU source combination allows inputting both a U register and an R register. Because that combination is D,A (not D,B), and because of overloading of rA field, each R register can be combined in the ALU with only one block of 16 U registers.

• Note that stackPointer addressing only applies to the first block of 16 U registers, i.e., the Mesa stack.

• Alternate U register addressing (AltUaddr) provides a means of reading a block of main memory into a block of U registers in a small loop.
SU (Stack and U) registers — examples

- Naming
  \[\text{RegDef[RD1, \, R, \, 7]; \quad \{\text{Disk reg 1}\}}\]
  \[\text{RegDef[RD2, \, R, \, 8]; \quad \{\text{Disk reg 2}\}}\]
  \[\text{RegDef[UDisk1, \, U, \, 70]; \quad \{\text{Disk}\}}\]
  \[\text{RegDef[UDisk2, \, U, \, 80]; \quad \{\text{Disk}\}}\]

- Usage examples
  \[\text{RD1} \leftarrow \text{UDisk1} \quad ,c1;\]
  \[\text{UDisk1} \leftarrow \text{RD2} \quad ,c2;\]
  \[\text{RHD1} \leftarrow \text{UDisk2} \quad ,c3;\]
  \[\text{RD1} \leftarrow \text{UDisk1 xor RD2} \quad ,c1;\]
  \[\text{RD2} \leftarrow \text{UDisk1 and Q} \quad ,c2;\]
  \[\text{UDisk1} \leftarrow \text{RD1 or RD2} \quad ,c3;\]

  \{\text{Examples using A-bypass}\}
  \[\text{UDisk2} \leftarrow \text{RD2, RD1} \leftarrow \text{RHD1} \quad ,c1;\]
  \[\text{UDisk2} \leftarrow \text{RD2, RD2} \leftarrow \text{RHD2} \quad ,c2;\]
  \[\text{UDisk2} \leftarrow \text{RD2, RD1} \leftarrow \text{RD1 + RD2 + 1} \quad ,c3;\]
Constants

- Set[] macro (built into Mass)
  
  Set[CSBSIZE, 0F]; {hexadecimal is default}
  Set[CSBSIZE, 0F’x]; {also hex}
  Set[CSBSIZE, 17’b]; {octal}
  Set[CSBSIZE, 15’d]; {decimal}

- a nibble or byte constant can be embedded in a microinstruction using the \textit{fZ} or \textit{fY}, \textit{fZ} fields respectively

- frequently used 16-bit constants can be preloaded into dedicated U registers

- other 16-bit constants can be generated on the fly by various tricks (see the \textit{DMR})
Shifting & Rotating

- Single-bit shifting is available in the ALU, either left or right, in 16 or 32-bit units
Shifting & Rotating — 1-bit shifting

- Shift end \((SE←)\) is used to specify what is shifted into the end of the 16-bit or 32-bit register(s)
- \(SE←\) is equivalent to \(Cin←\), which has more than one meaning — **Watch out**!
- Thus \(Cin\) is used to distinguish SU reads from writes, so
  - SU read implies \(SE←0\),
  - SU write implies \(SE←1\).
- \(Cin\) is also used by the ALU for carry in, so remember that
  - \(R + R\) implies \(SE←0\),
  - \(R + R + 1\) implies \(SE←1\),
  - \(R - R - 1\) implies \(SE←0\),
  - \(R - R\) implies \(SE←1\).
Shifting & Rotating — 4-bit rotates

- 4-bit rotating is available between the Y bus and X bus
  - anything on the Y bus can be rotated 0, 4, 8, or 12 bits
  - either output of ALU or register to be input to ALU can be rotated
- When LRotn is used with A-bypass, the ALU can be used for other purposes. Remember that $rB$-designated register must always be written when A-bypass is specified.
Shifting & Rotating — examples

- Single-bit shifts

RD1 ← LShift1 RD1, c1;
RD1 ← RShift1 (RD1 and 0FF), c2;
RD1 ← LShift1 (RD1 + 1), c3;

RD2 ← LShift1 0FF, SE ← 1  \{RD2 ← 01FF\}, c1;
RD1 ← RShift1 (RD1 xor ~RD1) \{07FFF\}, c2;
RD2 ← DALShift1 RD2, SE ← 1 \{Q.15 ← 0\}, c3;

\{U register complications\}
RD2 ← LShift1 UDisk1, \{SE ← 0\}, c1;
UDisk1 ← RD1, RD1 ← RD1 LShift1 \{SE ← 1\}, c2;
RD2 ← LShift1 RD2, Xbus ← UDisk2 \{SE ← 0\}, c3;

\{ALU complications\}
RD2 ← LShift1 (RD1 + RD2), \{SE ← 0\}, c1;
RD1 ← LShift1 (RD1 − RD2), \{SE ← 1\}, c2;
Microcoding is really different

- Control store
  - provides only instruction for execution
  - can't store into it (from the CP)
  - can't read from it in same sense of obtaining its contents as Data

- Memory
  - is "slow" to access
  - Belongs to somebody else (software)

- So, one is left with registers CR, U, RH

- Cycles
  Three microinstructions per clock:
  \[ C_1, \quad C_2, \quad C_3 \]
  \( C_1 \) exactly
  \( C_2 \) \( C_3 \)

Some functions belong to particular cycles
  (e.g., \( MAR \leq \), \( C_1 \) is only 3)

(?) Onion Microcoding
How much can you do in a single UI?

Constraints:
- Multiple use of fields (e.g., RH < and MP < both use FX)
- Timing:
  - e.g., User Source and ALU with Takes 1000

Cycle:
- Functions restricted to a particular cycle (noops used)
- Synchronization (mpaths)

Allocation:
- Multiple constraints (e.g., UI reads two places at once)
Microcode Tools
1. Mass 
   microcode assembler
2. MakePLion MicroBoot 
   microcode packager
3. Burdock 
   microcode Debugger

Mass Syntax
Note: --- underlining ⇒ optional

- Micro instruction generally: statements
  label: clause, ... clause, \texttt{\textbackslash r};

  where \texttt{Cn} is \{C1, C2, C3, or \texttt{*}\}
  talk about \texttt{*} when talk about subroutine

Non u I. generating statements
  clause, ... clause;

Definitions file - no microcode generation
Mass requires: Define Register using Reg def
  rejects: \texttt{\textbackslash r}; \texttt{\textbackslash r}A - must use reg def
  everything in Hex, I'd assume 'b' Octal
Branched + Dispatches

Terms
- µI: micro Instruction
- µIS: plural
- CS: control store
- INIA: intermediate next instruction address (1264c)
  \[ \text{INIA} = \text{INIA} \left[ 0-11 \right] = \text{MI} \left[ 36-47 \right] \]
  " is a field of MI
- NIA - next instruction address
- Branch
- Dispatch
- TPC - task program counter
- TC - task condition

Allocate 6 µISs for P
Dispatching

How Normal (ie. Disp Br) Dispatching Works

$8 \times 3 \sim \text{dispatch spec}$

$\text{Blk Addr} \Rightarrow \text{one of}$

$8 \text{Blk Addr} \lor i \mod m^3$

$+ [i, m, \text{Blk Addr}]$

$\Rightarrow \text{INIA}$

$\Rightarrow \text{Disp w [Blk Addr]}$

$\Rightarrow \text{INIA (0-7)}$

$\Rightarrow \text{Disp} (\text{just return})$

$\Rightarrow \text{C (0-3)}$
How to make a Mesa.db with your new code:

1. Retrieve Template.mc and add your code.

2. Retrieve FetchfilesSierra.ccm from [Idun]<fasnacht>Class> and execute it in the executive. This will retrieve all of the microcode source files, Mass, MakeDlionMicroBoot, Burdock and the files it needs, and some command files.

3. Execute Mesa.ccm in the executive. This will assemble all of the source files included in Mesa.db and will take some time. Once you have done this, the intermediate symbols files that the assembler uses will be around on your disk (*.si) and the assembly will go much faster when you do it a second time.

4. Fetch *Germ.burduck into a window and edit it. Put the names of your registers before the semicolons so they will appear in the titles of the Burdock window.

5. Run BurdockDlion.bcd in the executive.

6. Deactivate the IOP window.

7. Enter "Germ" in the "File" field of the CP window and bug Run!

8. When Othello has booted on the debuggee machine, boot the debuggee machine to its tajo or copilot volume. (If Othello doesn't boot, something is wrong. Give me a call.)


10. On the debugger machine, bug Stop! Enter the label of the first instruction of your byte code in the type-in field, select it, and bug Break! Then bug Continue!

11. On the debuggee machine, bug GO! in the tool window. This should cause your byte code to run and you should hit the breakpoint set at your first instruction.

12. From here on, its general debugging. Continue along, looking at what's in your registers and setting frequent breaks.
SetTask[0];
RegDef[r1, R, 1];
RegDef[r2, R, 2];
RegDef[r6, R, 6];
RegDef[rh2, RH, 2];
RegDef[rh1, RH, 1];
RegDef[rh6, RH, 6];

{You can use only the above R registers and RH registers. You can use any U-register EXCEPT those
defined in [Idun]<APilot100>Microcode>Private>DLion>Dandelion.df with a * after them. Remember that U
register bank 0 is the stack.}
RegDef[u0, U, 0];
RegDef[u14, U, 14];
RegDef[u17, U, 17];
RegDef[u24, U, 24];
RegDef[u27, U, 27];
RegDef[u43, U, 43];
RegDef[u60, U, 60];
RegDef[u63, U, 63];

something + TOS {virtual source high},
something + SIK {virtual source low}, pop.
something + STK {virtual dest high}, pop.
something + SIK {virtual dest low}, pop.

......
......
......
......
......

TOS + STK {put next thing on stack into TOS}, pop. GOTO[IBDispOnly], c1;
PROC [destination: LONG POINTER TO ARRAY[0..16] OF WORD, source: LONG POINTER TO ARRAY[0..16]]

BEGIN
  FOR i: CARDINAL IN [0..15] DO
    destination[15-i] := source[i];
  ENDLOOP;
END;

1. Unload the stack (R, RH)
2. Map the source and dest virtual address
3. Add 16 to dest address
4. Read a word out of source, increment source
  - Store a word into dest, decrement dest
  - Check counter to see if done