Emulation Implementation Notes

August 22, 2015

Part I

Source Code Layout

1.1 dps8_sys

This module handles the abstract "entire system." The bulk of the code is initialization and simh command processing hooks.

1.1.1 dps8_cable

Hardware device cabling emulation

1.2 dps8_cpu

The CPU emulator

1.2.1 dsp8_iefp

CPU Memory access

1.2.2 dps8_addrmods

CPU Address Modifcation

1.2.3 dps8_append

CPU Append Unit

1.2.4 dps8_bar

CPU BAR address computation

1.2.5 dps8_ins

CPU instructions

1.2.6 dps8_eis

CPU EIS instructions

1.2.7 dps8_math

CPU math support routines

1.2.8 dps8_opcodetable

CPU opcode table; drives address preparation and mode checking

1.2.9 dps8_faults

CPU fault handling code

1.2.10 dps8_decimal

CPU Decimal math support routines

1.3 dps8_scu

The SCU emulator

1.4 dps8_iom

The IOM emulator

1.4.1 dps8_console

Operator console device

1.4.2 dps8_disk

Disk device

1.4.3 dps8_mt

Tape device

1.4.4 dps8_fnp, fnp_ipc

FNP device

1.4.5 dps8_lp

Printer device

1.4.6 dps8_crdrdr

Card reader device

1.5 dps8_utils

Common utility routines

1.5.1 shm

Host shared memory management

1.6 Other

dps8_utils Common support code
dps8_clk Unused simh clock hooks
dps8_loader Segment loader for unit tests
dps8_stddev Vestigial
dps8_fxe Faux Multics Execution module

Part II

CPU operation

2.1 simh

When simh is running the CPU it calls sim_instr(), which is the CPU emulator entry point.

sim_instr() loops, executing emulated instructions until some emulation halting condition is met, or some simh component signals for a pause due to an external event.

2.2 CPU emulation organization

sim_instr first establishes a setjmp context; this is used primarily by RCU mechanism to restart instruction processing after restoring a saved system state due to a fault or an interrupt.

The cpu emulation is written as a state machine. The longjmp parameter is used to setup the desired state; at initial entry, setjmp returns a zero, and the appropriate setup is done.

```
#define JMP\_ENTRY       0

    int val = setjmp(jmpMain);

    switch (val)
      {
        case JMP\_ENTRY:
        case JMP\_REENTRY:
            reason = 0;
            break;
        case JMP\_NEXT:
            goto nextInstruction;
        case JMP\_STOP:
            reason = STOP\_HALT;
            goto leave;
        case JMP\_SYNC\_FAULT\_RETURN:
            goto syncFaultReturn;
        case JMP\_REFETCH:
            cpu . wasXfer = false;
            setCpuCycle (FETCH\_cycle);
            break;
        case JMP\_RESTART:
            setCpuCycle (EXEC\_cycle);
            break;
        default:
          sim\_printf ("longjmp value of %d unhandled\n", val);
          goto leave;
      }
```

The cpu emulation then enters a "do ... while (reason == 0)" loop, which cycles the CPU through its states.

The top of the loop checks the simh components for events that need to be handled (simh_hooks()), polls various subsystems for service requests (emulator console commands to be processed, incoming FNP messages and operator console input).

```
reason = 0;

// Process deferred events and breakpoints
reason = simh\_hooks ();
if (reason)
  {
    //sim\_printf ("reason: %d\n", reason);
    break;
  }

static uint queueSubsample = 0;
if (queueSubsample ++ > 10240) // ~ 100Hz
  {
    queueSubsample = 0;
    scpProcessEvent ();
    fnpProcessEvent ();
    consoleProcess ();
  }
if (check\_attn\_key ())
  console\_attn (NULL);
```

It then checks for Timer Register runout, setting the group 7 fault flag if needed. (Group 7 faults are distinguished as not resulting from instruction execution, but external events; and has such are handled synchronously between instruction execution steps, rather then interrupting mid-instruction.

```
bool overrun;
UNUSED word27 rTR = getTR (& overrun);
if (overrun)
  {
    ackTR ();
    if (switches . tro\_enable)
      setG7fault (FAULT\_TRO, 0);
  }
```

Next, it then checks for lockup (the operating system has not enabled interrupts for more then 32 ms.), and faults if needed.

```
lufCounter ++;
// Assume CPU clock ~ 1Mhz. lockup time is 32 ms
```

```
if (lufCounter > 32000)
  {
    lufCounter = 0;
    doFault (FAULT\_LUF, 0, "instruction cycle lockup");
  }
```

Lastly, it checks the CPU state and branches to the appropriate code.

```
switch (cpu . cycle)
  {
    case INTERRUPT\_cycle:
      ....
  }
```

The states are:

FETCH_cycle Fetch the next instruction

EXEC_cycle Execute an instruction

INTERRUPT_cycle Fetch an Interrupt instruction pair

INTERRUPT_EXEC_cycle Execute the even instruction of an interrupt pair

INTERRUPT_EXEC2_cycle Execute the odd instruction of an interrupt pair

FAULT_cycle Fetch an Fault instruction pair

FAULT_EXEC_cycle Execute the even instruction of a fault pair

FAULT_EXEC2_cycle Execute the odd instruction of a fault pair

The normal instruction flow is alternating FETCH and EXEC cycles.

2.3 FETCH_cycle

The fetch cycle first checks for pending interrupts and group 7 faults, according to complex eligibility rules (AL39, pg 327, "Interrupt Sampling.")

```
if ((! cpu . wasInhibited) &&
    (PPR . IC % 2) == 0 &&
    (! cpu . wasXfer) &&
    (! (cu . xde | cu . xdo | cu . rpt | cu . rd)))
  {
    cpu . interrupt\_flag = sample\_interrupts ();
    cpu . g7\_flag = bG7Pending ();
  }
```

```
// The cpu . wasInhibited accumulates across the even and
// odd intruction. If the IC is even, reset it for
// the next pair.

if ((PPR . IC % 2) == 0)
  cpu . wasInhibited = false;
```

If a eligible interrupt is pending, the CPU state is switched to INTERRUPT_cycle.

```
if (cpu . interrupt\_flag)
  {
    setCpuCycle (INTERRUPT\_cycle);
    break;
  }
```

Likewise, if a Group 7 faults is pending, cause a fault.

```
if (cpu . g7\_flag)
  {
    cpu . g7\_flag = false;
    doG7Fault ();
  }
```

There is now code to process the XEC and XED instructions; the idea here is that the processing of the XEC and XED instructions loads the target instructions into the Control Unit IWB and IODD words, and that the fetch cycle is a no-op, as the instructions have already been fetched.

If not the XEC or XED the case, the instruction is fetched into the CU IWB word.

```
else
  {
    processorCycle = INSTRUCTION\_FETCH;
    clr\_went\_appending ();
    fetchInstruction (PPR . IC);
  }
```

Now that the instruction is in the IWB, switch to EXEC state.

```
setCpuCycle (EXEC\_cycle);
break;
```

2.4 EXEC_cycle

The execute cycle starts right off with:

```
t\_stat ret = executeInstruction ();
```

A return value of greater than 0 indicates that the CPU needs to halt:

```
if (ret > 0)
  {
    reason = ret;
    break;
  }
```

A return value of CONT_TRA indicates that a transfer instruction was executed, which requires different handling. A transfer may be of interest to the handling of transfer into append or BAR modes, so the 'wasXfer' flag is set; and the code that increments the IC is skipped. The CPU state is set to EXEC_cycle, and it all repeats.

```
if (ret == CONT\_TRA)
  {
    cpu . wasXfer = true;
    setCpuCycle (FETCH\_cycle);
    break;   // don't bump PPR.IC, instruction already did it
  }
cpu . wasXfer = false;
```

Next, the IC is incremented to point to the next instruction; any EIS operands that the instruction had are also skipped over. The CPU state is set to FETCH_cycle, and it all repeats.

```
PPR.IC ++;
if (ci->info->ndes > 0)
  PPR.IC += ci->info->ndes;

cpu . wasXfer = false;
setCpuCycle (FETCH\_cycle);
break;
```

2.5 Instruction execution: 'executeInstruction'

Instruction execution is managed by 'executeInstruction()'.

When 'executeInstruction' starts, the instruction has been loaded into the CU IWB (Control Unit Instruction Working Buffer).

First, the instruction is decoded, extracting the operation code, address field, tag field and the A and I bits into the 'currentInstruction' structure; and setting the 'info' member to point to the operation's entry in the opcode table.

```
decodeInstruction(cu . IWB, ci);
```

The 'info' member contains a wide variety of details about the instruction, including:

PREPARE_CA Instruction will need the operand's computed address.

READ_OPERAND Instruction will read the operand.

WRITE_OPERAND Instruction will write the operand.

NO_RPT Not allowed in a repeat instruction.

PRIV_INS Privileged instruction.

The number of EIS operands.

Allowed tag values.

The flags in the 'info' structure are used to check instruction restrictions, such as privilege and allowed modifiers; violation causes a fault.

```
if ((ci -> info -> flags & PRIV\_INS) && ! is\_priv\_mode ())
    doFault (FAULT\_IPR, ill\_proc,
             "Attempted execution of privileged instruction.");
// No CI/SC/SCR allowed
if (ci->info->mods == NO\_CSS)
{
    if (\_nocss[ci->tag])
        doFault(FAULT\_IPR, ill\_mod, "Illegal CI/SC/SCR modification");
}
// No DU/DL/CI/SC/SCR allowed
else if (ci->info->mods == NO\_DDCSS)
{
    if (\_noddcss[ci->tag])
        doFault(FAULT\_IPR, ill\_mod, "Illegal DU/DL/CI/SC/SCR modification");
}
// No DL/CI/SC/SCR allowed
else if (ci->info->mods == NO\_DLCSS)
{
    if (\_nodlcss[ci->tag])
        doFault(FAULT\_IPR, ill\_mod, "Illegal DL/CI/SC/SCR modification");
}
// No DU/DL allowed
else if (ci->info->mods == NO\_DUDL)
{
    if (\_nodudl[ci->tag])
        doFault(FAULT\_IPR, ill\_mod, "Illegal DU/DL modification");
}
```

Next, an assortment of initializations occurs, setting various registers to the operand address field, and initializing the TPR register.

```
    TPR.CA = address;
    iefpFinalAddress = TPR . CA;
    rY = TPR.CA;

    TPR.TRR = PPR.PRR;
    TPR.TSR = PPR.PSR;
```

If the instruction has EIS operands, they are read into holding variables.

```
    if (info -> ndes > 0)
      {
        for(int n = 0; n < info -> ndes; n += 1)
          {
// XXX This is a bit of a hack; In general the code is good about
// setting up for bit29 or PR operations by setting up TPR, but
// assumes that the 'else' case can be ignored when it should set
// TPR to the canonical values. Here, in the case of a EIS instruction
// restart after page fault, the TPR is in an unknown state. Ultimately,
// this should not be an issue, as this folderol would be in the DU, and
// we would not be re-executing that code, but until then, set the TPR
// to the condition we know it should be in.
            TPR.TRR = PPR.PRR;
            TPR.TSR = PPR.PSR;
            Read (PPR . IC + 1 + n, & ci -> e . op [n], EIS\_OPERAND\_READ, 0);
          }
        // This must not happen on instruction restart
        if (! (cu . IR & I\_MIIF))
          {
            du . CHTALLY = 0;
            du . Z = 1;
          }
      }
```

If the instruction expects the address field to be converted to the Computed Address, do that.

```
        if (ci->info->flags & PREPARE\_CA)
          {
            doComputedAddressFormation ();
            iefpFinalAddress = TPR . CA;
          }
```

Otherwise if the instruction wants the operand value, do that. 'readOperands' will handle single, double, eight and sixteen word operands. The value is held in 'CY', 'Ypair', 'Yblock8' or 'Yblock16' holding registers, as appropriate. Read operands handles all aspects of indirection and address appending.

```
    else if (READOP (ci))
      {
        doComputedAddressFormation ();
        iefpFinalAddress = TPR . CA;
        readOperands ();
      }
```

Now that the operands have sorted, the instruction is executed.

```
t\_stat ret = doInstruction ();
```

If an transfer instruction has the A bit set, and accesses the Append Unit during the Computed Address formation, the processor is switch to Append mode.

```
if (info->ndes == 0 && a && (info->flags & TRANSFER\_INS))
  {
    if (get\_addr\_mode () == BAR\_mode)
      set\_addr\_mode(APPEND\_BAR\_mode);
    else
      set\_addr\_mode(APPEND\_mode);
  }
```

Finally, it the instruction writes the operand, do that.

```
if (WRITEOP (ci))
  {
    if (! READOP (ci))
      {
        doComputedAddressFormation ();
        iefpFinalAddress = TPR . CA;
      }
    writeOperands ();
  }
```

2.6 Instruction execution: 'doInstruction'

First, initialize the EIS state registers.

```
if (i->info->ndes > 0)
{
    i->e.ins = i;
    i->e.addr[0].e = &i->e;
    i->e.addr[1].e = &i->e;
    i->e.addr[2].e = &i->e;

    i->e.addr[0].mat = OperandRead;   // no ARs involved yet
```

```
    i->e.addr[1].mat = OperandRead;   // no ARs involved yet
    i->e.addr[2].mat = OperandRead;   // no ARs involved yet
}
```

And switch based on the opcode extension bit; 'Basic' and 'EIS' are misleading here; the opcode extension bit has a minimal correlation to the EIS instruction opcode layout, but by separating the two cases, the code becomes a bit more readable, Both routines have the same preamble:

```
DCDstruct * i = & currentInstruction;
uint opcode  = i->opcode;  // get opcode

switch (opcode)
{
   ....
}
```

We will look at a few of the instructions so as to understand the general function of 'doInstruction.'

2.6.1 LCA Load Complement A

'readOperands()' has already retrieved the operand value, and left it in CY. The utility routine 'compl36' complements the value and sets the IR flags, and the assignment operation places the complemented value in the A register.

```
case 0335:  // lca
  rA = compl36 (CY, & cu . IR);
  break;
```

2.6.2 LREG Load Registers

LREG loads the A, Q, E, and index registers from a Y-block8. Again, 'readOperands()' has loaded the values into Yblock8.

```
case 0073:   ///< lreg
  rX[0] = GETHI(Yblock8[0]);
  rX[1] = GETLO(Yblock8[0]);
  rX[2] = GETHI(Yblock8[1]);
  rX[3] = GETLO(Yblock8[1]);
  rX[4] = GETHI(Yblock8[2]);
  rX[5] = GETLO(Yblock8[2]);
  rX[6] = GETHI(Yblock8[3]);
  rX[7] = GETLO(Yblock8[3]);
  rA = Yblock8[4];
  rQ = Yblock8[5];
  rE = (GETHI(Yblock8[6]) >> 10) & 0377;   // need checking
  break;
```

2.6.3 STA Store A

Since 'writeOperands()' will do the actual writing, all STA needs to do is copy A to CY.

```
case 0755:  // sta
  CY = rA;
  break;
```

2.6.4 STXn Store Index Register n

For many opcodes, the low bits of the opcode contains indexing information.

```
case 0740:  ///< stx0
case 0741:  ///< stx1
case 0742:  ///< stx2
case 0743:  ///< stx3
case 0744:  ///< stx4
case 0745:  ///< stx5
case 0746:  ///< stx6
case 0747:  ///< stx7
  {
    uint32 n = opcode & 07;  // get n
    SETHI(CY, rX[n]);
  }
break;
```

2.6.5 TRA Transfer

For the TRA instruction, the computed address is placed in the PPR, and the function return value is used to signal that a transfer is to occur.

```
case 0710:  ///< tra
  PPR.IC = TPR.CA;
  PPR.PSR = TPR.TSR;
  return CONT\_TRA;
```

2.7 RPT/RPD

TODO

2.8 XEC/XED

TODO

2.9 EIS

TODO

2.10 Computed Address Formation

TODO

2.11 Append Unit

TODO

2.12 Interrupts

Interrupts are handled at the start of the CPU fetch cycle:

```
if (cpu . interrupt\_flag)
  {
    setCpuCycle (INTERRUPT\_cycle);
    break;
  }
```

The DPS8M interrupt handling logic is to save the system state, place the processor in "temporary absolute mode", fetch a pair of instructions from the indicated location in memory and execute them. If one of the pair of instructions is a transfer instruction, the processor is set to absolute mode. If neither of the instructions transfers, the Control Unit is restored from the saved state and the processor is switched back to the fetch cycle.

The INTERRUPT_cycle handler saves the Control Unit state so the processor can return to the state that was extant at the time the interrupt was handled. The interrupt number being serviced is stored in the Control Unit, where the guest operating system can inspect it.

```
// In the INTERRUPT CYCLE, the processor safe-stores
// the Control Unit Data (see Section 3) into
// program-invisible holding registers in preparation
// for a Store Control Unit (scu) instruction, enters
// temporary absolute mode, and forces the current
// ring of execution C(PPR.PRR) to
// 0. It then issues an XEC system controller command
// to the system controller on the highest priority
// port for which there is a bit set in the interrupt
// present register.

uint intr\_pair\_addr = get\_highest\_intr ();
```

```
cu . FI\_ADDR = intr\_pair\_addr / 2;
cu\_safe\_store ();
```

Next, the processor is placed in "temporary absolute mode":

```
// Temporary absolute mode
set\_TEMPORARY\_ABSOLUTE\_mode ();

// Set to ring 0
PPR . PRR = 0;
TPR . TRR = 0;
```

The interrupt pair is fetched and scheduled for execution:

```
// get interrupt pair
core\_read2 (intr\_pair\_addr, instr\_buf, instr\_buf + 1, \_\_func\_\_);

cpu . interrupt\_flag = false;
setCpuCycle (INTERRUPT\_EXEC\_cycle);
break;
```

The INTERRUPT_EXEC_cycle handler recovers an instruction from holding and executes it.

```
case INTERRUPT\_EXEC\_cycle:
case INTERRUPT\_EXEC2\_cycle:
  {
    if (cpu . cycle == INTERRUPT\_EXEC\_cycle)
      cu . IWB = instr\_buf [0];
    else
      cu . IWB = instr\_buf [1];

    t\_stat ret = executeInstruction ();
```

If the instruction was a transfer instruction, set the processor to absolute mode and start normal fetch/execute processing.

```
if (ret == CONT\_TRA)
  {
    cpu . wasXfer = true;
    setCpuCycle (FETCH\_cycle);
    set\_addr\_mode (ABSOLUTE\_mode);
    break;
  }
```

Otherwise, if the instruction just executed was the first of the pair, schedule the execution of the second.

```
    if (cpu . cycle == INTERRUPT\_EXEC\_cycle)
      {
        setCpuCycle (INTERRUPT\_EXEC2\_cycle);
        break;
      }
```

The only possibility now is that both instructions have been executed and neither transferred, so restore the saved state and resume processing.

```
    clear\_TEMPORARY\_ABSOLUTE\_mode ();
    cu\_safe\_restore ();
    cpu . wasXfer = false;
    setCpuCycle (FETCH\_cycle);
    break;
```

2.13 Faults

Faults fall (mostly) into two categories, ones that are generated by instruction execution (page faults, overflow, etc.), and those that are independent of the instruction (timer run-out, lockup fault, etc.).

The Group 7 faults are handled similarly to interrupts; at the start of the CPU fetch cycle the Group 7 fault pending flag is queried.

```
    if (cpu . g7\_flag)
      {
        cpu . g7\_flag = false;
        doG7Fault ();
      }
```

'doG7Fault' is invoked rather then switching cycles directly.

```
void doG7Fault (void)
  {
     if (g7Faults & (1u << FAULT\_TRO))
       {
         g7Faults &= ~(1u << FAULT_TRO);
         doFault (FAULT\_TRO, 0, "Timer runout");
       }

     if (g7Faults & (1u << FAULT\_CON))
       {
         g7Faults &= ~(1u << FAULT\_CON);
         cu . CNCHN = g7SubFaults [FAULT\_CON] & MASK3;
         doFault (FAULT\_CON, g7SubFaults [FAULT\_CON], "Connect");
       }
```

```
    // Strictly speaking EXF isn't a G7 fault, put if we treat is as one,
    // we are allowing the current instruction to complete, simplifying
    // implementation
    if (g7Faults & (1u << FAULT\_EXF))
      {
        g7Faults &= ~(1u << FAULT\_EXF);
        doFault (FAULT\_EXF, 0, "Execute fault");
      }
    doFault (FAULT\_TRB, (\_fault\_subtype) g7Faults, "Dazed and confused in doG7Fault"
}
```

Both the interrupt and fault paths involve saving the Control Unit, but the fault case was abstracted into 'doFault', due to the large number of code paths leading there; while there is only the single entry into the INTERRUPT_cycle, and the Control Unit save is done in the INTERRUPT_cycle state handler.

TODO

2.13.1 doFault()

TODO

2.13.2 RCU

TODO

Part III

"As-built"

3.1 Variations between the hardware and the emulator

DPS8 memory is managed by the CPU code, not the SCU code.
The history registers are not implemented.
The RPL instruction is not implemented.
The MPCs have been abstracted into the device code.