Emulation Implementation Notes

August 20, 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_opcode

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;
```