

Virtual Prototyping of COTS Processor-Based Systems

Dr. Vijay K. Madiseti

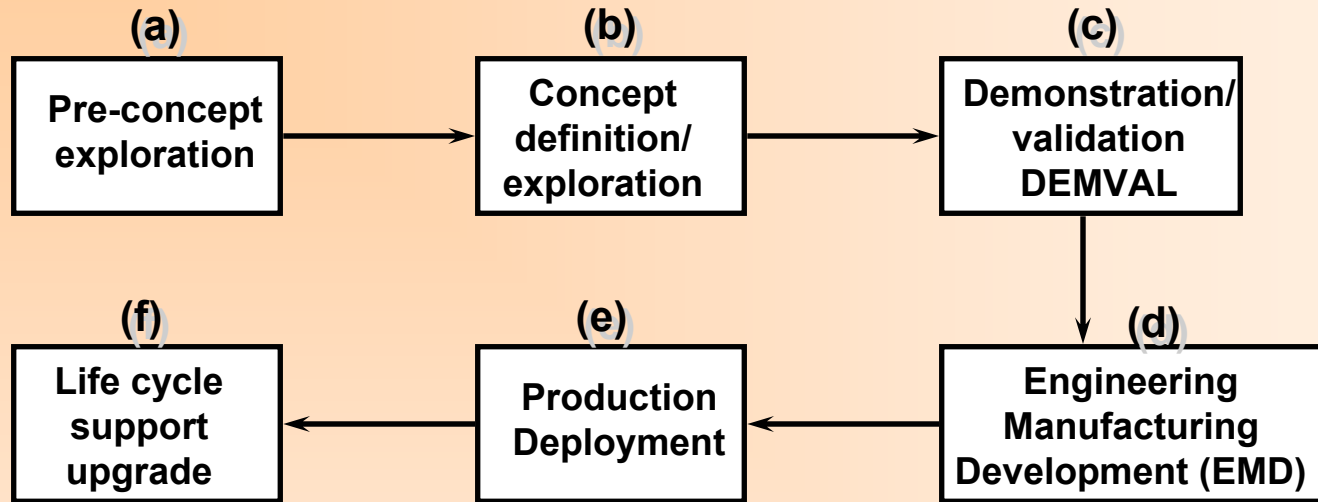
VP Technologies, Inc.

Vkm@vptinc.com

(770) 578 0448 Cell

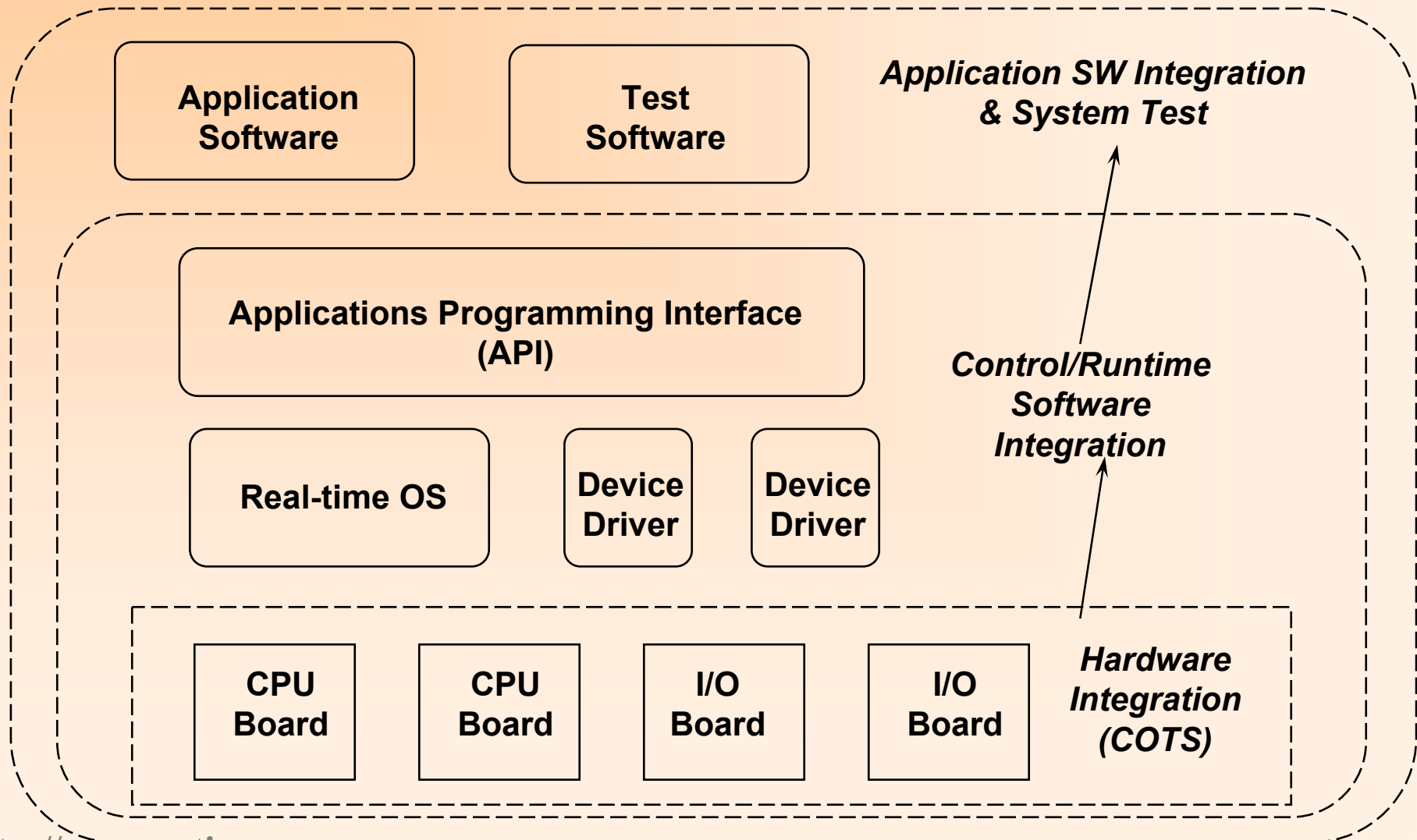
<http://www.vptinc.com>

Typical System Prototype Development Phases



- Currently, there are “independent” activities with little design or personnel continuity
- Communication is primarily through paper which results in
 - High cost
 - Low design efficiency
 - Difficult support
 - Loss of information through traversal between design stages a, b, c, d, e, and f

Embedded-System HW/SW Design, Integration, and Test



Need for Virtual Prototyping

- At earlier development phases, design requirements and specifications primarily captured in paper form
- A modeling methodology is required to capture the design requirements and specifications in an executable format
 - Provides for early design verification through simulation
 - Helps remove requirements and specifications errors earlier in the design process

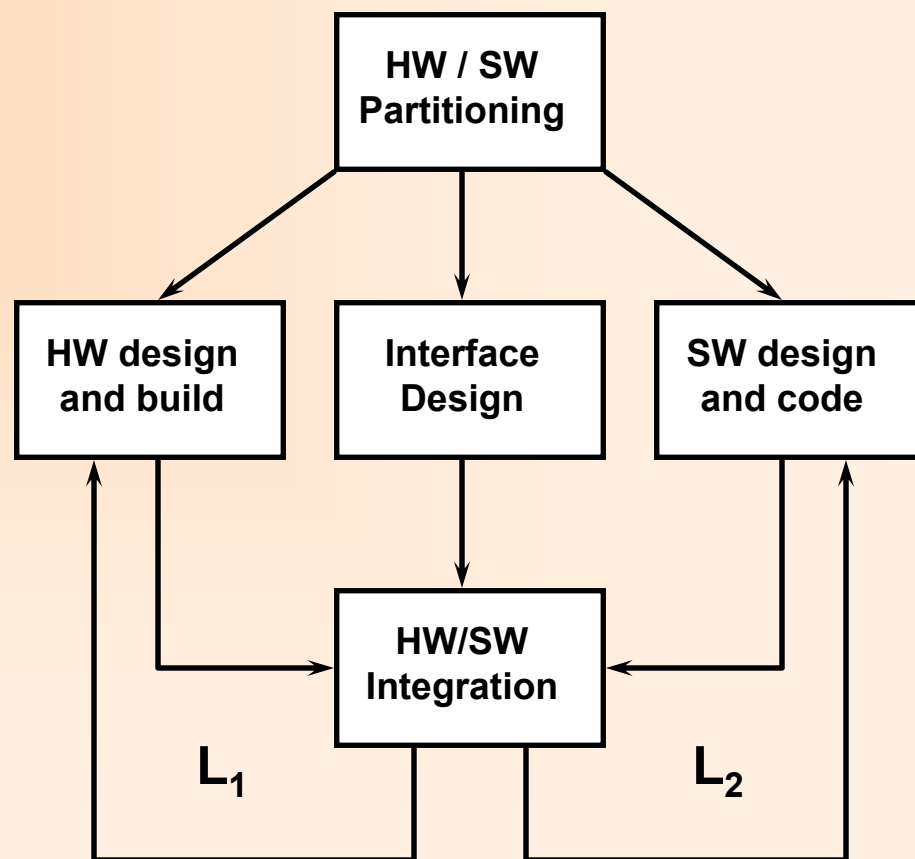
**The Virtual Prototype
provides this representation**

Need for Virtual Prototyping (cont.)

- SW design loop L_2 depends on HW design loop L_1
- L_1 can be slow and costly due to HW fabrication and test

- Virtual Prototyping couples the HW and SW design processes by eliminating hardware in the loop
- Reduces the L_1 time and initiates the L_2 design loop earlier in the design process

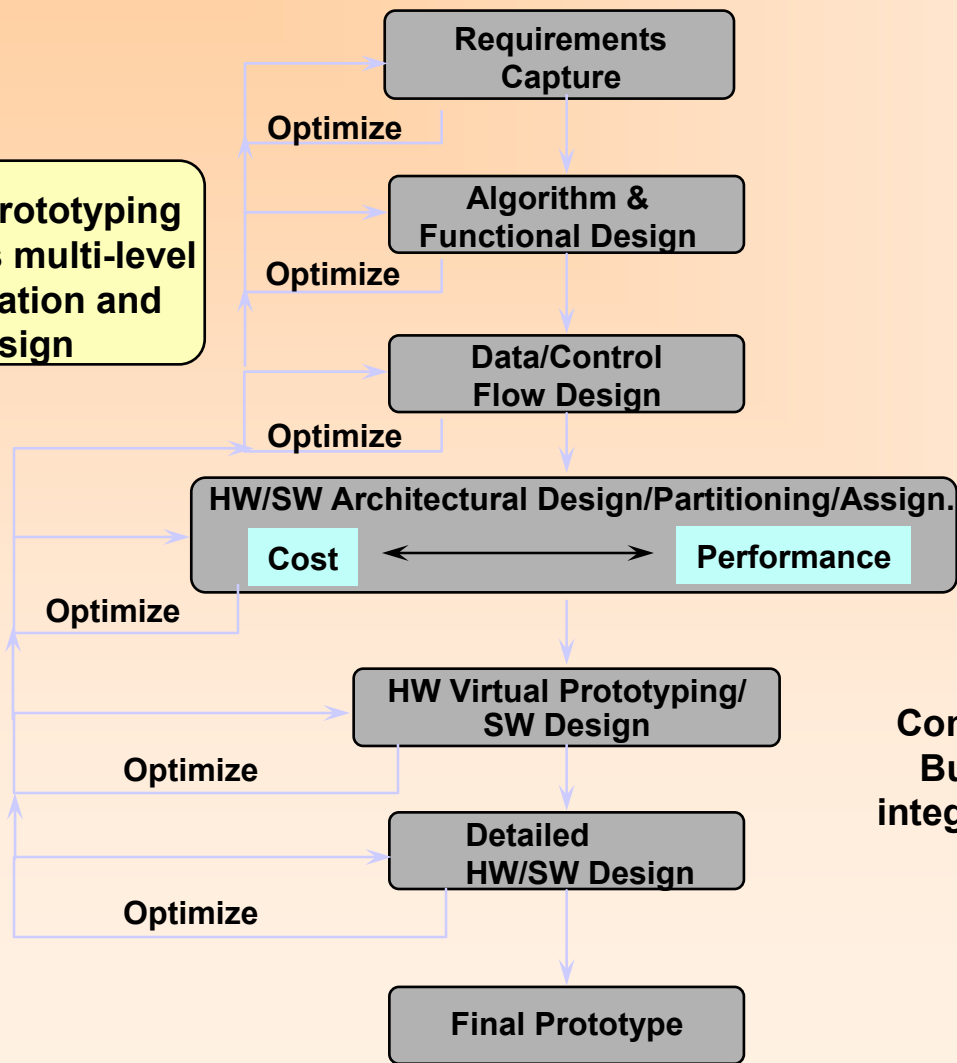
Dependence between
HW and SW development



Need for Virtual Prototyping (cont.)

Design can be optimized at multiple levels of abstraction to meet customer requirements

Virtual Prototyping facilitates multi-level optimization and design



Example

ATR

Edge Detection & pattern match on 16 bit data

i860's - mesh based multiprocessor

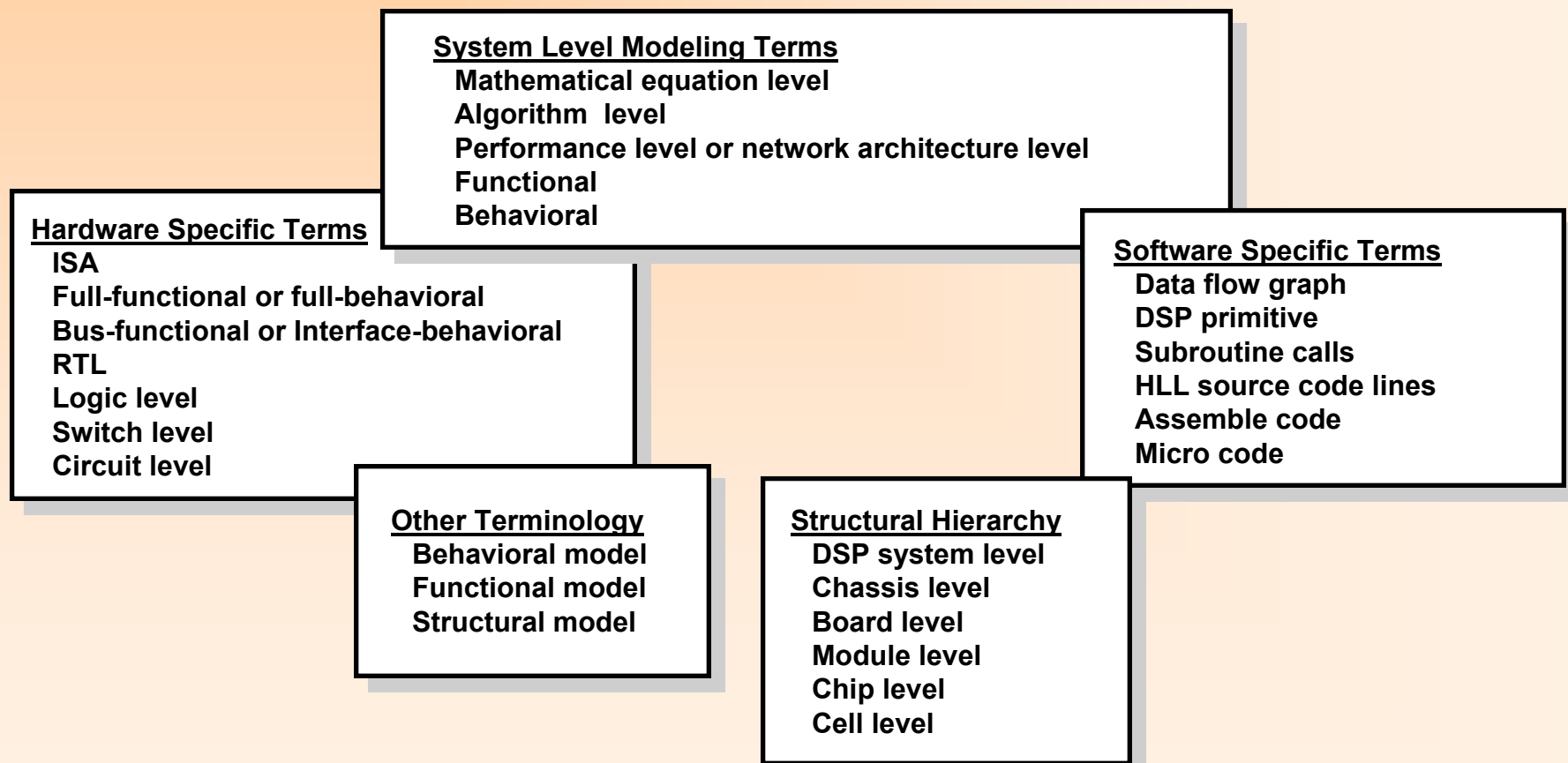
Control SW, application SW, Test SW, Bus protocols, HW boards, HW/SW integration, real-time operating systems I/O device drivers, etc.

Virtual Prototyping Should Provide the Following

- “Represent” the prototype during various stages in system development process
- “Represent” the prototype at multiple levels of abstraction in top-down design
- Allow optimization of design at multiple levels or different stages
- “Document” the design for efficient upgrades and support
- Be cost effective (time and dollars)

**One must understand the attributes of the system being designed to be able to “represent” it accurately
“Represent” = Modeling**

Resolution of Common Modeling Terms

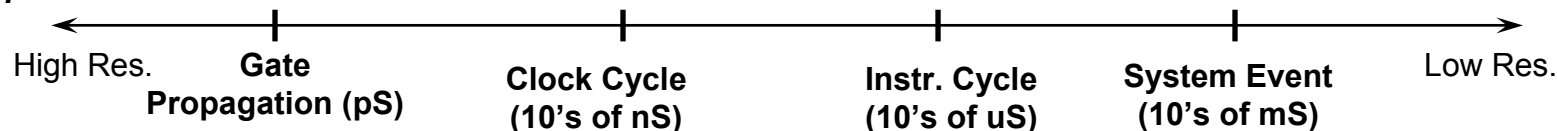


VHDL Modeling Taxonomy (Madisetti 1995, Hein/Madisetti 1998)

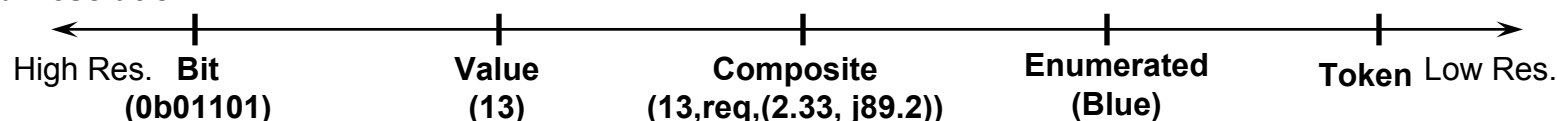
Independently Describe: 1) Resolution of INTERNAL (kernel) details
2) Representation of EXTERNAL (Interface) details

In terms of:

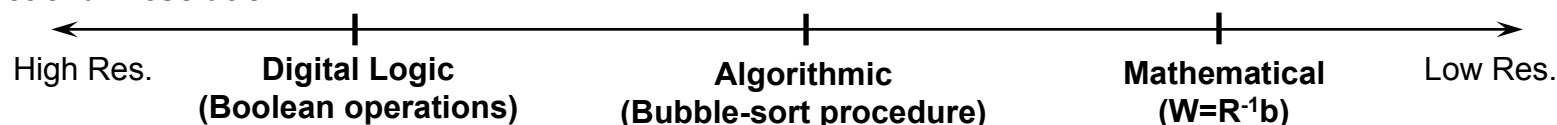
Temporal Resolution



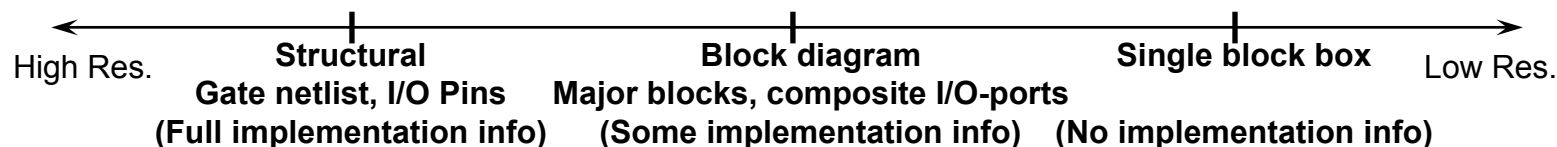
Data Resolution



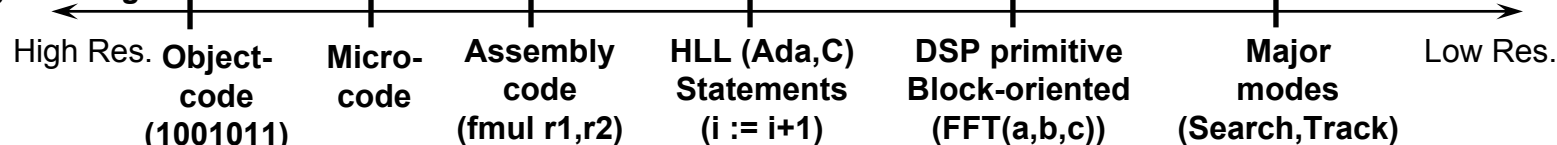
Functional Resolution



Structural Resolution

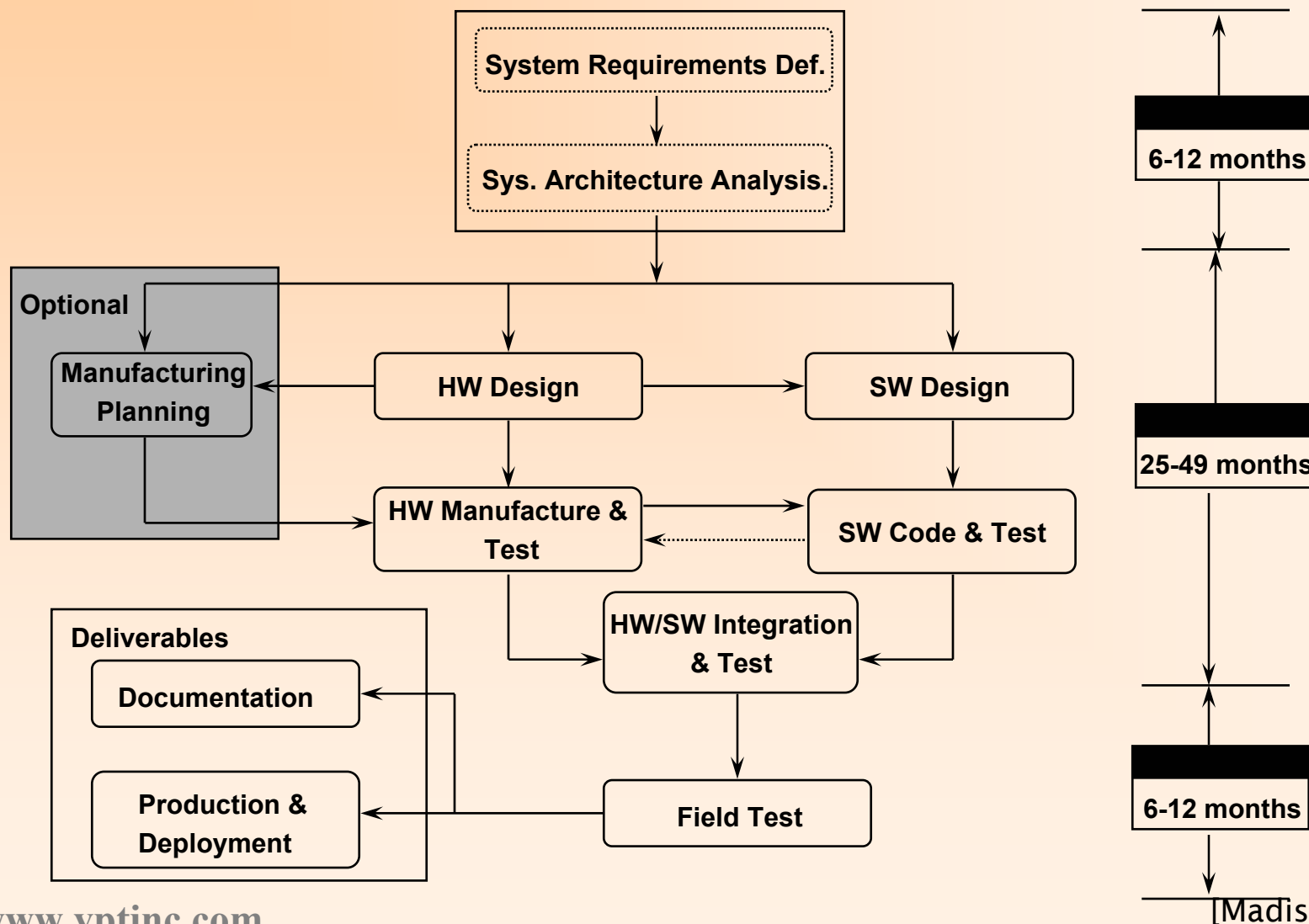


Programming Level

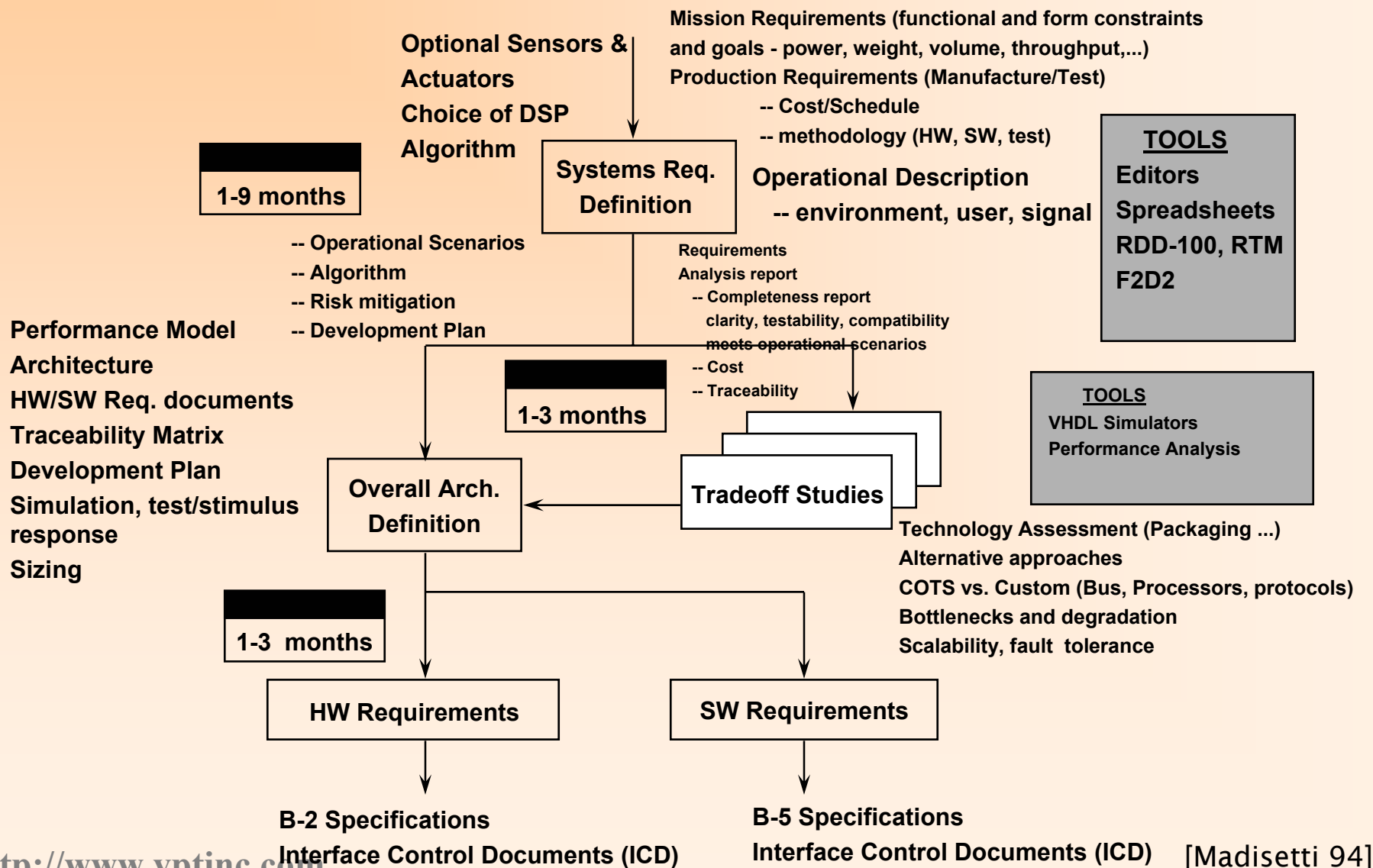


(Note: Low resolution of details = High level of abstraction
High resolution of details = Low level of abstraction)

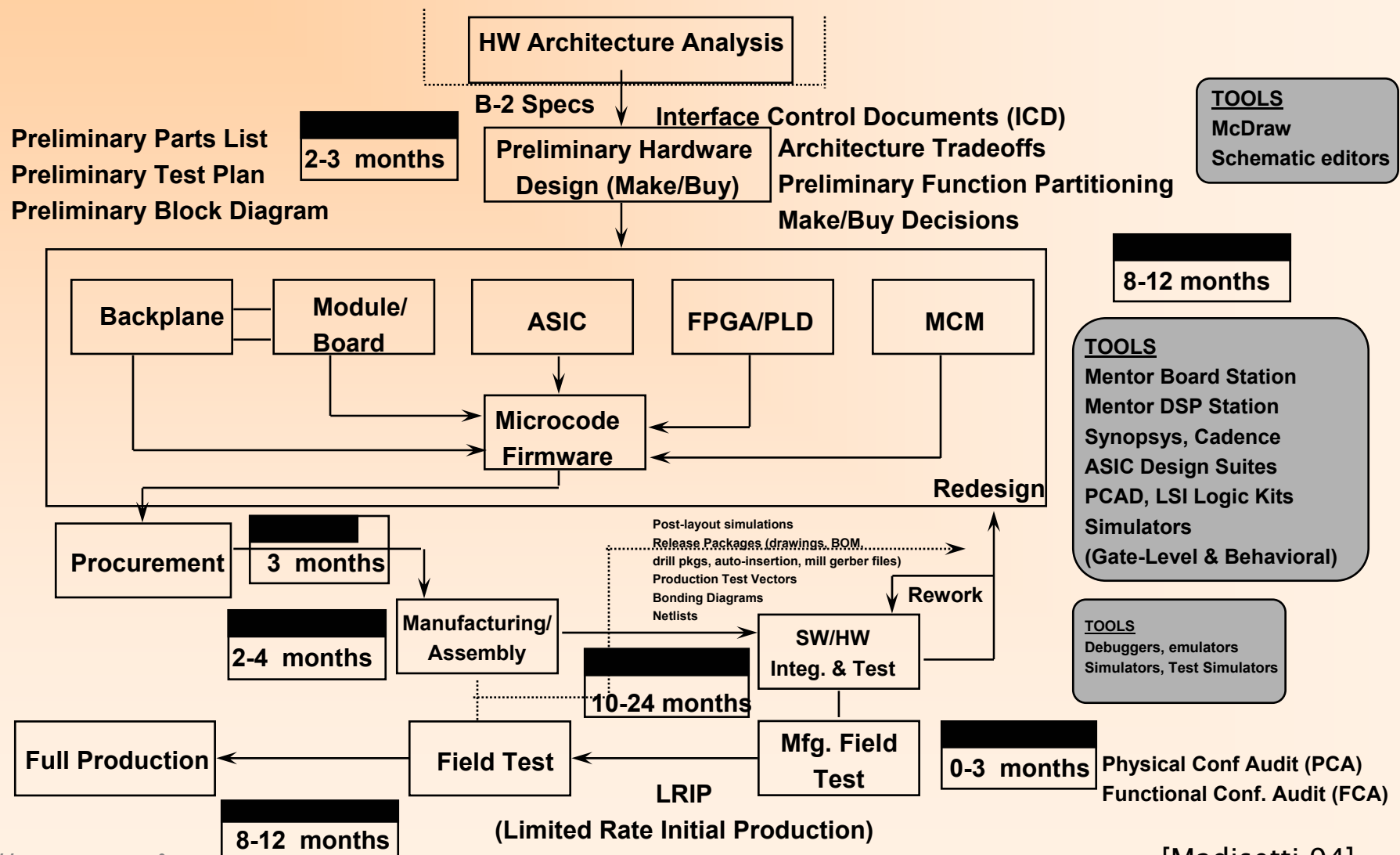
Traditional Design Process



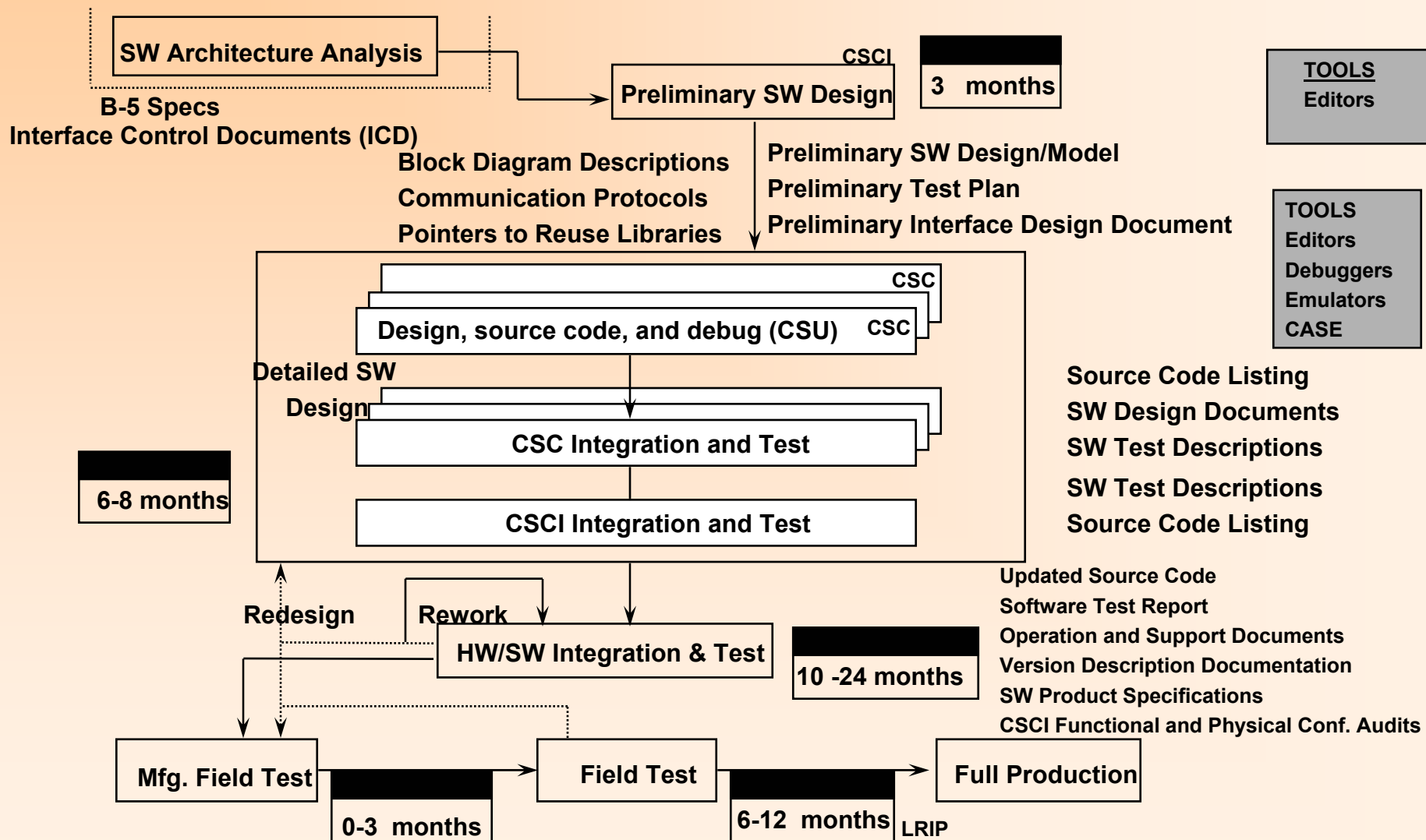
Architectural Analysis



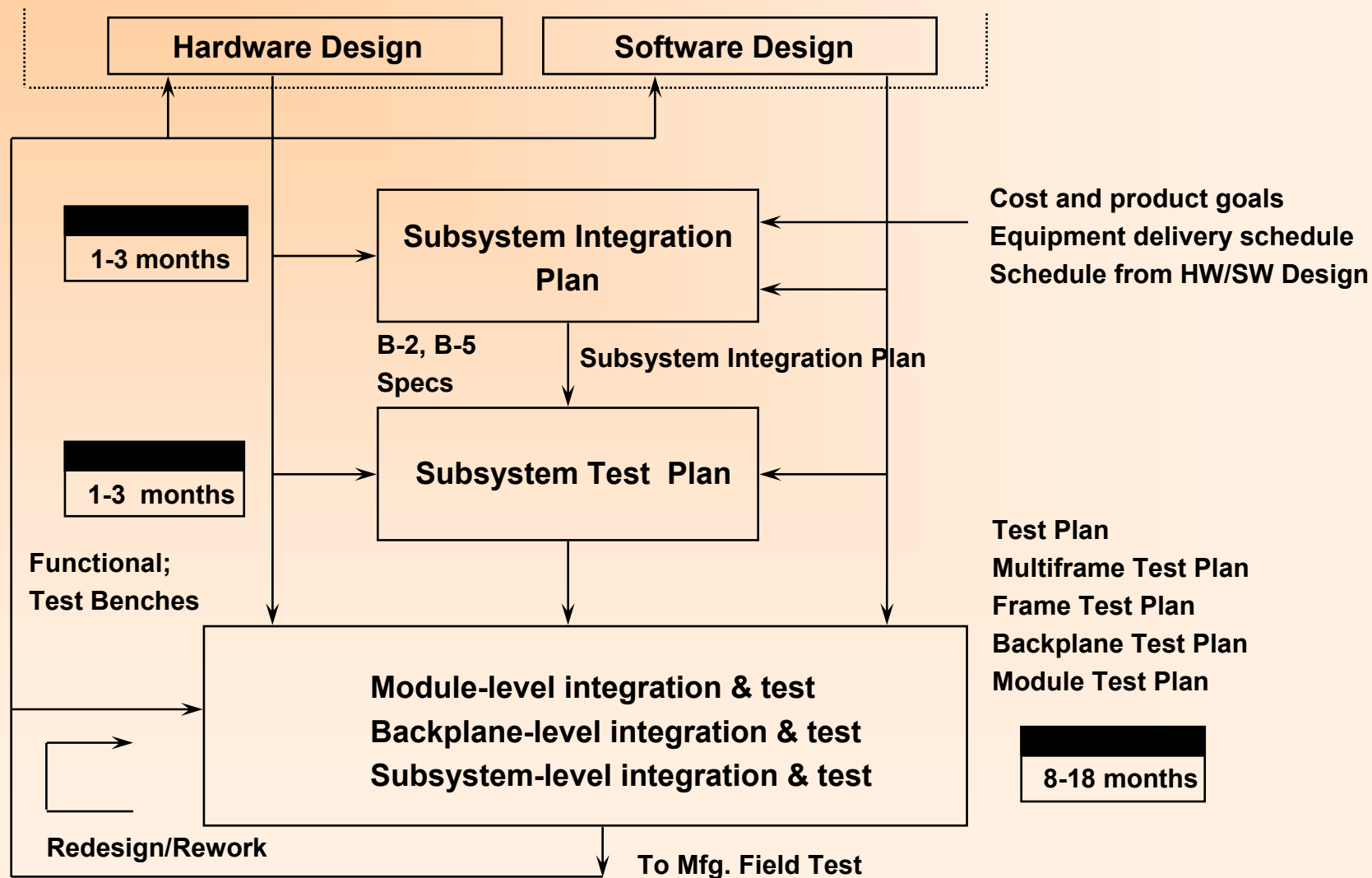
HW Design Process



SW Design Process

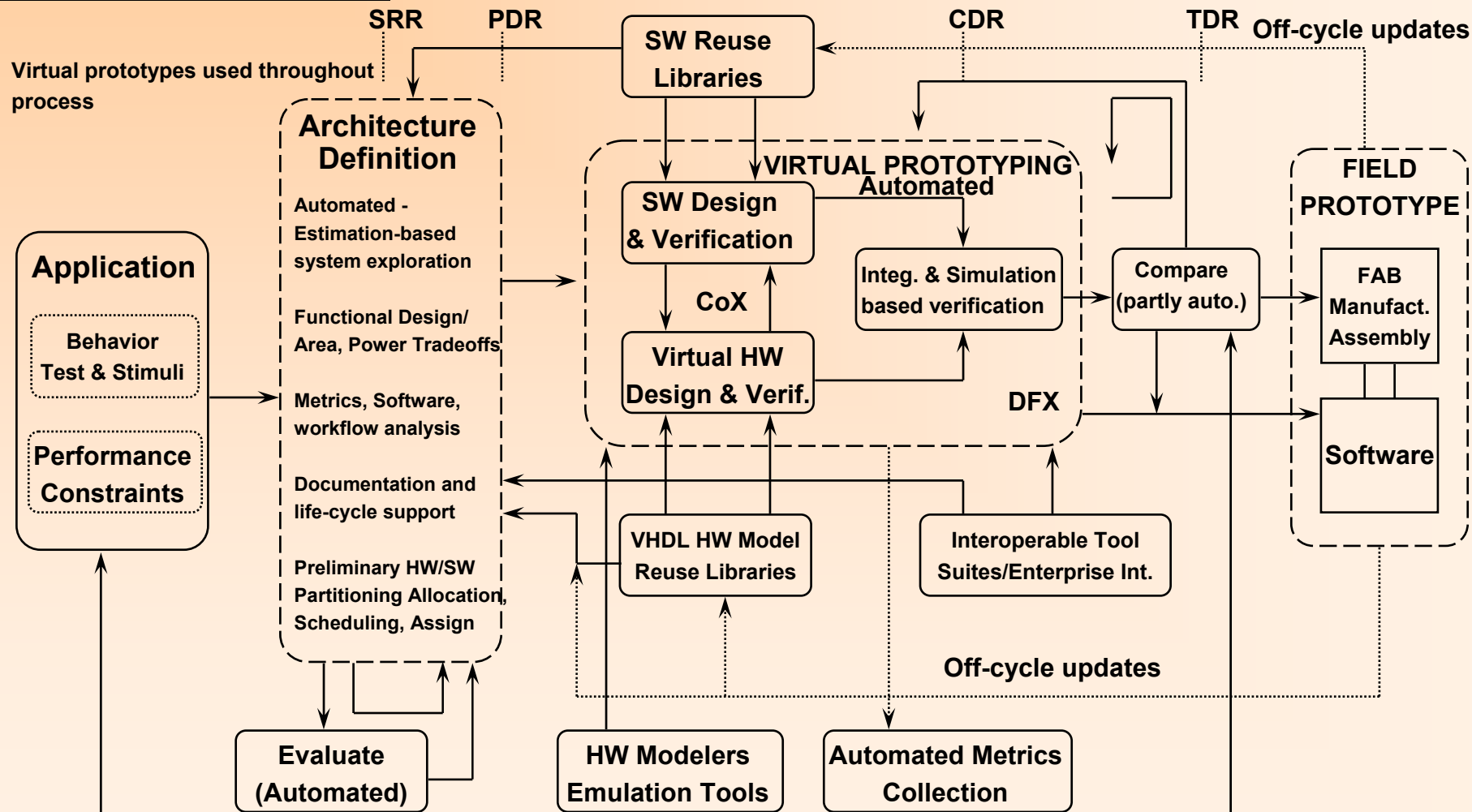


Integration Plan

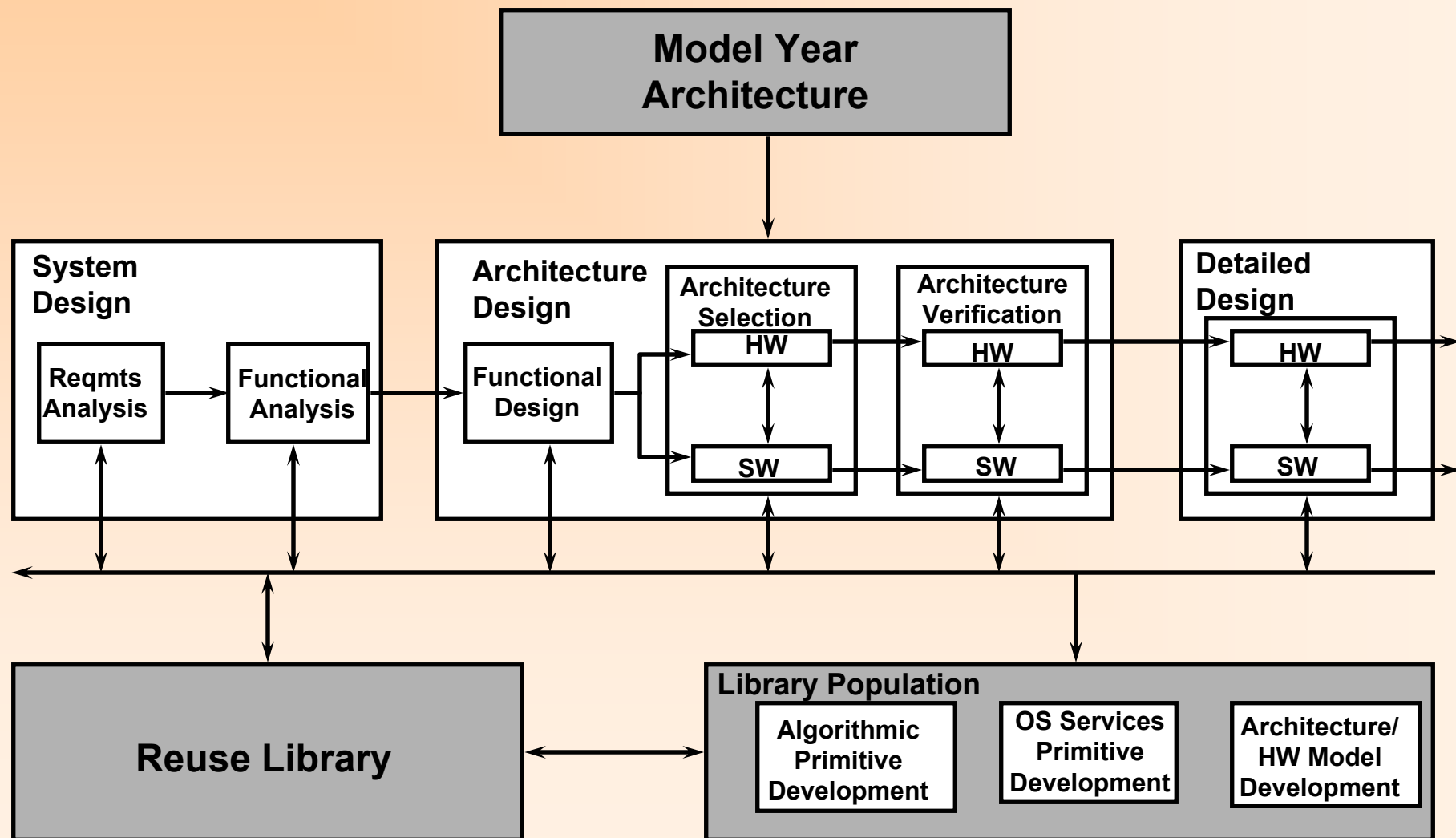


Top-down Design Process: Proposed Approach

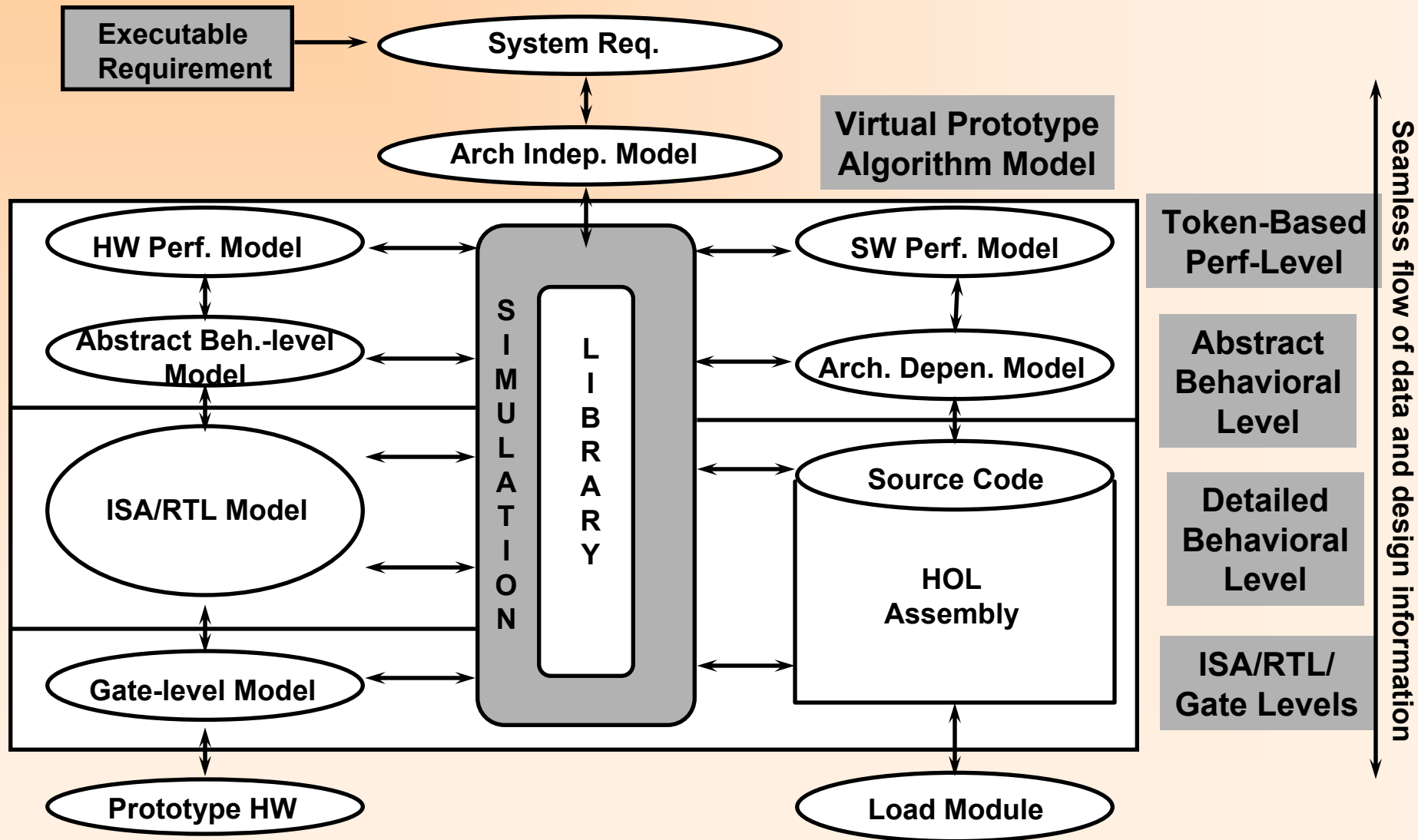
Top-down Design Flow - Mature



Stages in the Virtual Prototyping Design Process



Virtual Prototyping Design Process and Model Abstractions



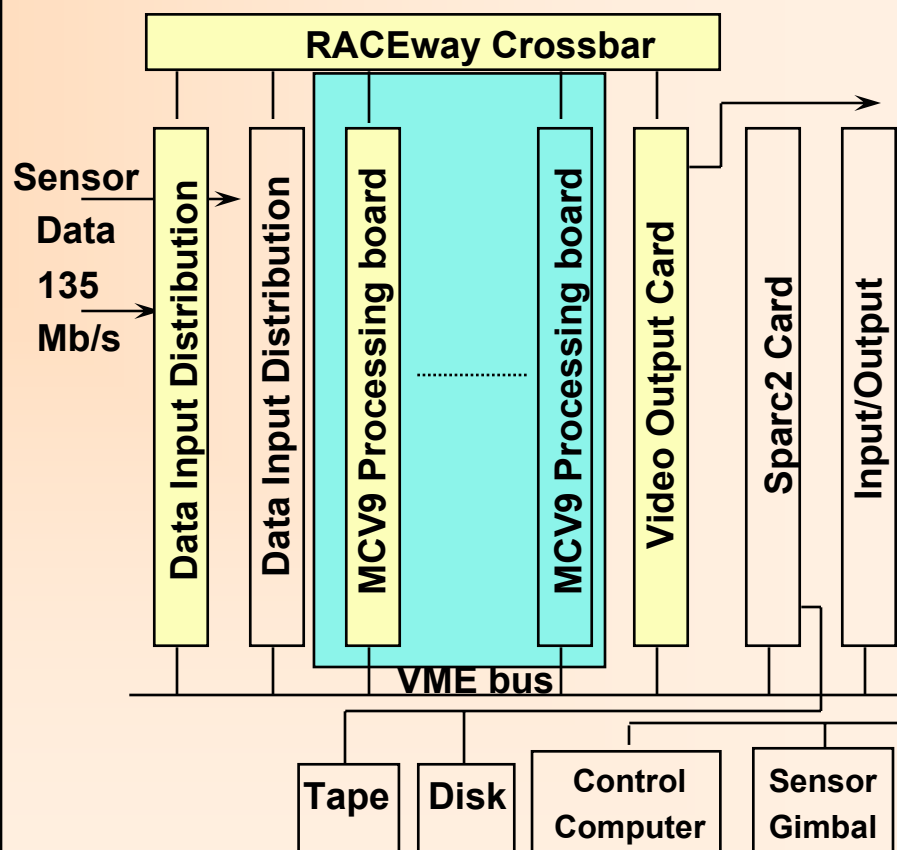
Case Study


Technology Foundations of Virtual Prototyping

Case Study: Design IRST System Prototype

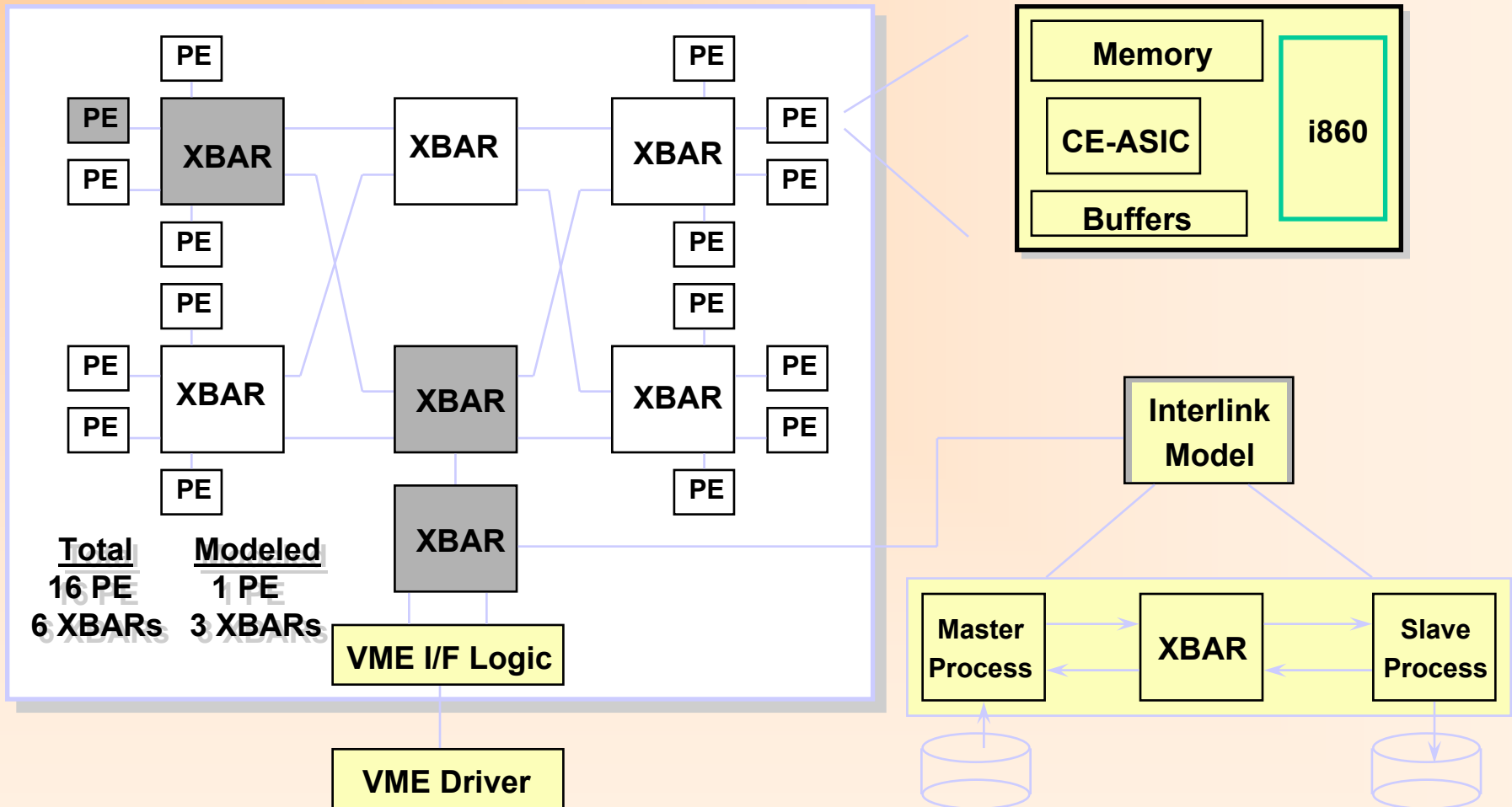
• Hardware Elements

- Data Input Distribution card
- RS-170 Daughter card
- Sensor Interface card
- Video output card
- MCV9 processing boards
- VME interface
- RACEway XBAR network
- 190 processors in total system and only a fraction of them were modeled at this level (1-16 processor tests)



 Elements of Hardware VP
created and simulated

MCV9 Subsystem Architecture



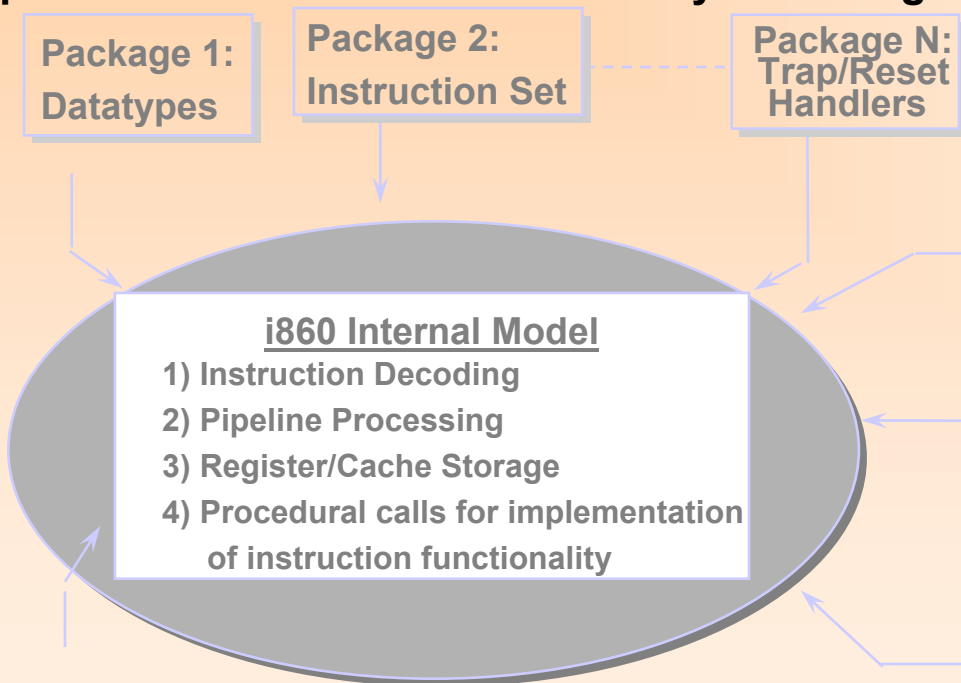
Need for an i860 Emulation

- The subsystem (MCV9) uses i860 and a number of interconnect components
- The next step in the virtual prototyping process is to emulate the i860 processor in VHDL, along with emulations of all other components on the MCV9 board

Case Study: i860XP RISC Emulation

i860 Detailed-behavioral Model

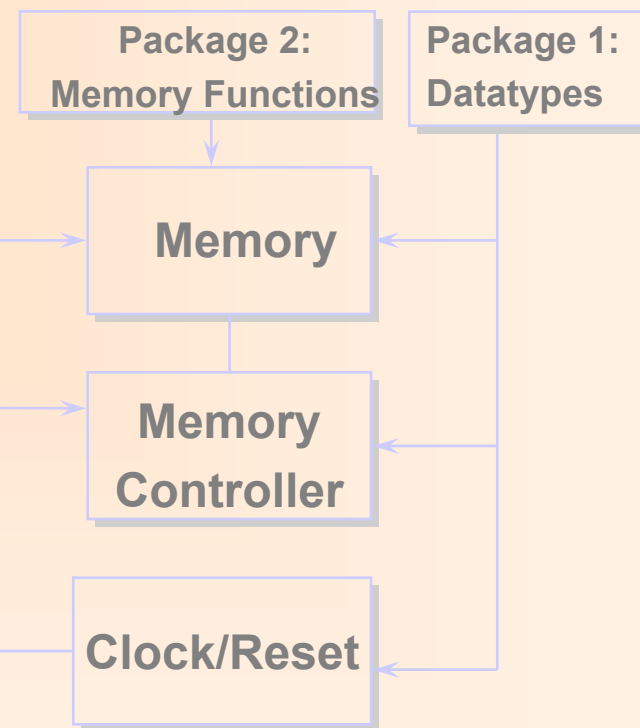
Encapsulation of Common Functionality in Packages



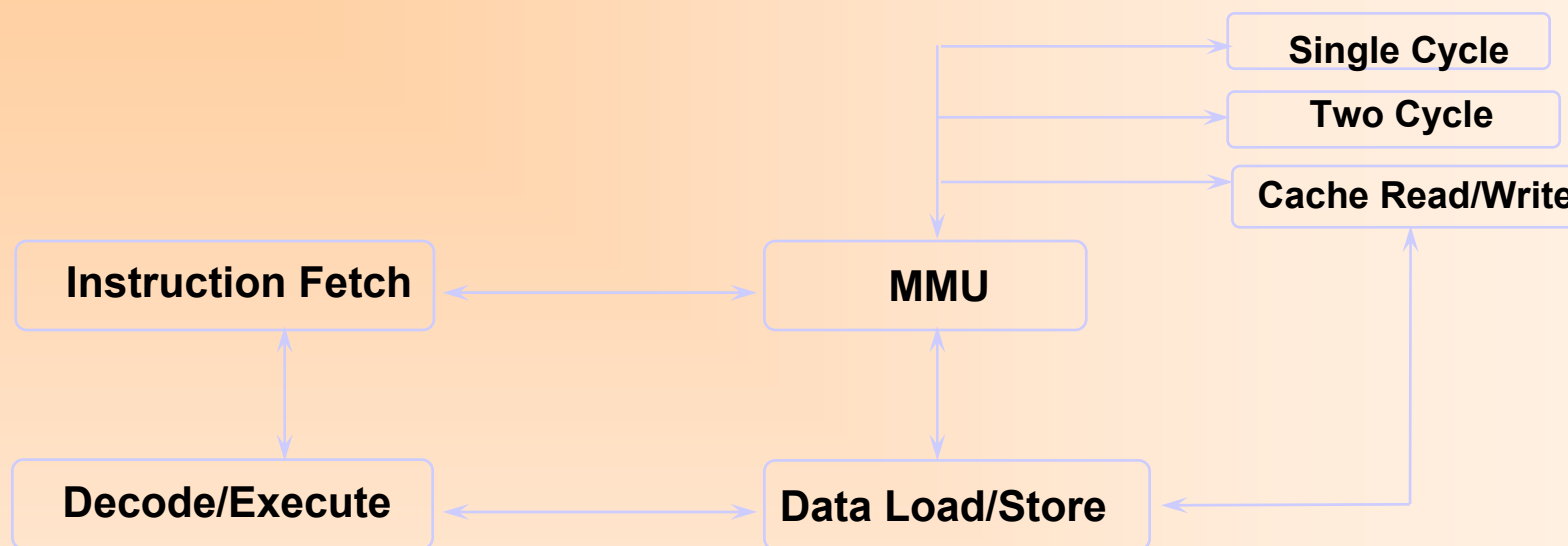
i860 Interface Wrapper

- 1) Interface Timing as specified in users manual

i860 testbench



Case Study i860: Functionality Breakdown



- Seven Processes
 - Needed to simulate concurrency of units
 - Approx. 200 instructions/sec
- Initially single process (ISS)
 - Interface behavior not needed
 - Approx. 2000 instructions/sec

Process Functionality

- Minimize use of internal signals
- Bit vectors bundled into integers or arrays of integers when passed between processes
- Decode/execute process has majority of functionality
 - Artifact of single- process model
- The table on the right represents the breakdown of functionality among the processes

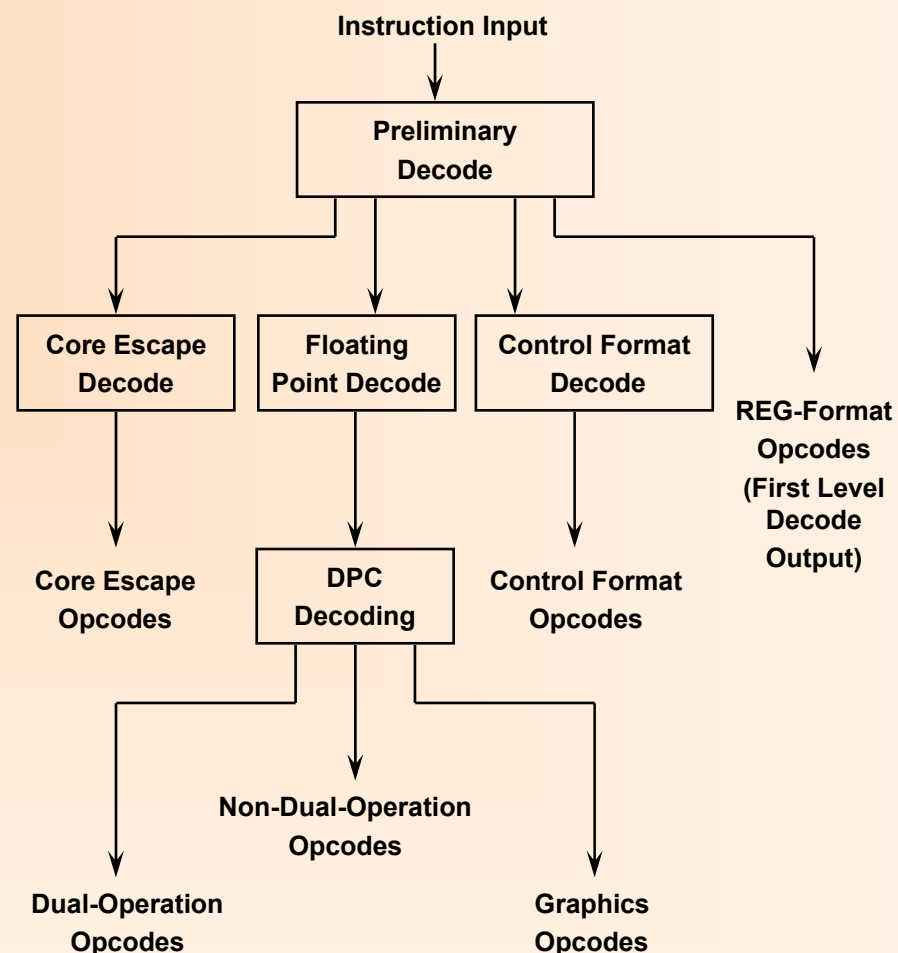
Process Name	Functionality
Instruction Fetch	Instruction Cache, Instruction Buffers Reset Functionality
MMU	Translation Caches, Address Translation Algorithm, Reset Functionality
Data Load/Store	Data Cache, Write Buffers, Pipelined Float Load Buffers, Reset Functionality
Single Cycle	Interface handling for single-cycle reads and writes, Reset Functionality
Two Cycle	Interface handling for burst reads and writes of 2 cycles, Reset Functionality
Cache Fill/Write	Interface handling for cache fills and write backs, Reset Functionality
Decode/Execute	Core Execution Unit, Floating Point Execution Unit, Graphics Unit, FP Pipeline, Register Files, Pipelined Float load buffers, Special Purpose Registers, Instruction Interaction checks, Dependency checking, Trigger mechanism for Data load/stores, Reset Functionality

Internal Model

- Fetch/decode/execute/write back functionality
- Internal storage
 - Caches
 - Registers
 - Pipeline stages
- Mostly contained inside decode/execute process except
 - Instruction cache contained in fetch process
 - Data cache contained in load/store process
 - Address translation caches in MMU process
- Instruction execution implemented with procedural calls
- Trap/reset/exception/interrupt handling
- Reuse through package encapsulation of common functionality
 - IEEE Standard 754 for floating point math

Instruction Set Decoding/Execution

- “Case” statements used to decode opcode
 - REG-Format => 1 case statement
 - Core Escape and CTRL-Format => 2 case statements
 - FP => 3 case statements
- Additional decoding done after opcode determined
 - Operand locations
 - Branch Offset
 - etc.
- Execution performed by use of procedural calls



Floating Point Decoding

- There are three levels of decoding for floating point instructions
- Preliminary decode uses OP_FLOAT
- Secondary decode uses the 2 bits from 6 down to 5 of the instruction
- Ternary decoding uses the lower 4 bits (DPC) to determine where to fetch the operands
- Operands from KR, floating point register file and previous multiply result

```
when OP_FLOAT =>
```

```
OP_FE := CURRENT_INSTR(6 downto 0);  
OP_FE_DPC1:= OP_FE(6 downto 5);  
OP_FE_DPC2:= OP_FE(4);  
DPC := OP_FE(3 downto 0);  
D := CURRENT_INSTR(9);  
S := CURRENT_INSTR(8);  
R := CURRENT_INSTR(7);  
FSRC1 := BITS_TO_NATURAL(CURRENT_INSTR(15 downto 11));  
FSRC2 := BITS_TO_NATURAL(CURRENT_INSTR(25 downto 21));
```

```
case OP_FE_DPC1 is
```

```
when OP_FE_1 =>
```

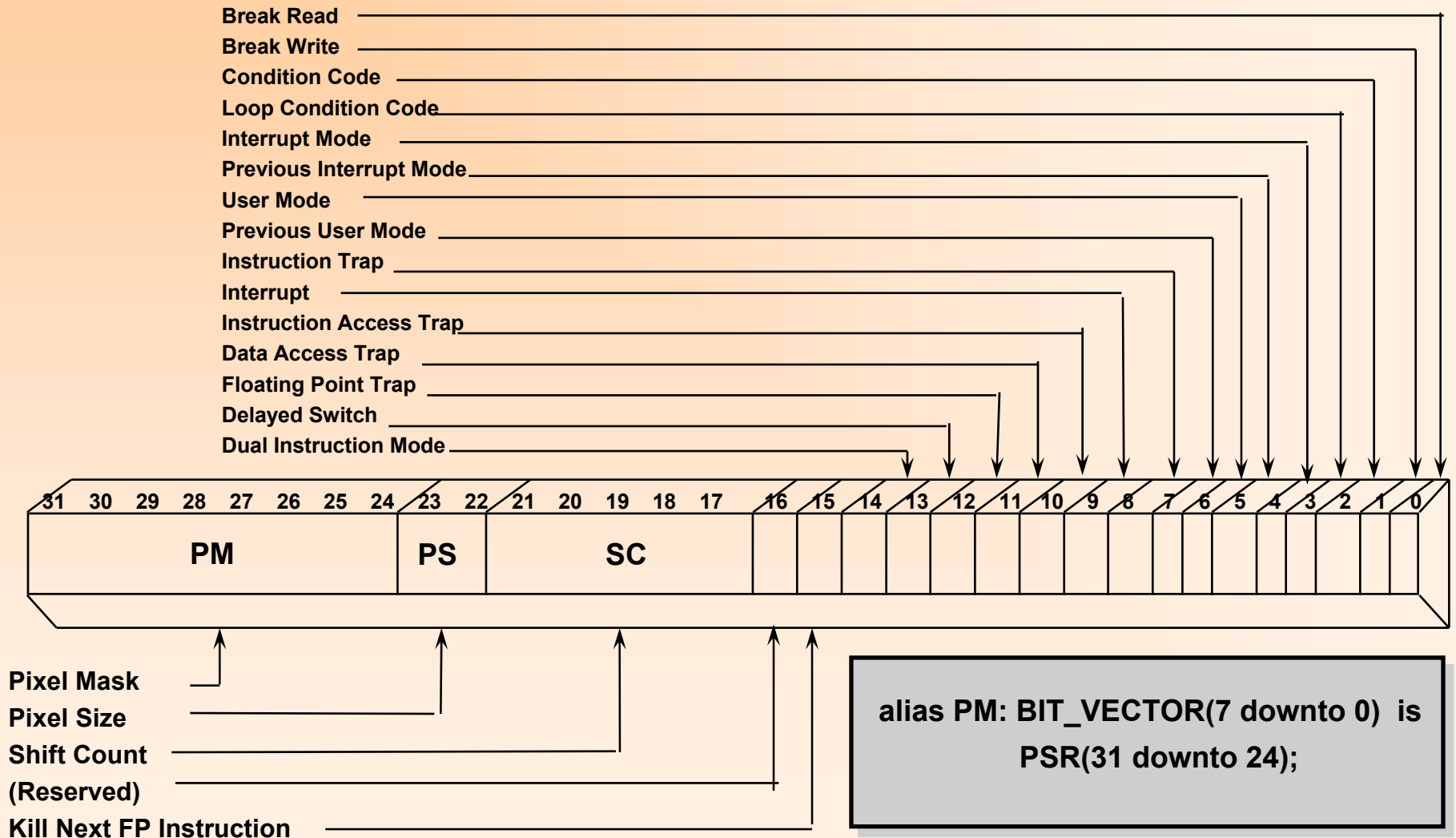
```
if (P = '1') then
```

```
case DPC is
```

```
when OP_FE_R2P1 =>
```

```
PFMY_OP1(0):= KR(0);  
PFMY_OP1(1):= KR(1);  
PFMY_OP2(0):= FL_REG(FSRC2);  
PFMY_OP2(1):= FL_REG(FSRC2+1);  
PFAD_OP1(0):= FL_REG(FSRC1);  
PFAD_OP1(1):= FL_REG(FSRC1+1);  
PFAD_OP2(0):= PFMY_REAL_DOUBLE_RES(0);  
PFAD_OP2(1):= PFMY_REAL_DOUBLE_RES(1);
```

Register Model



Processor Status Register (PSR) Breakdown

- The alias construct allows the processor status register to be accessed by its bit field names
- More readable and understandable
- Assignment to the aliased value implies assignment to the register field

-- Breakdown for the processor status register PSR

```
alias BR      : BIT is PSR(0);
alias BW      : BIT is PSR(1);
alias CC      : BIT is PSR(2);
alias LCC     : BIT is PSR(3);
alias IM      : BIT is PSR(4);
alias PIM     : BIT is PSR(5);
alias UPSR    : BIT is PSR(6);
alias PU      : BIT is PSR(7);
alias IT      : BIT is PSR(8);
alias IN_PSR  : BIT is PSR(9);
alias IAT     : BIT is PSR(10);
alias DAT     : BIT is PSR(11);
alias FT      : BIT is PSR(12);
alias DS      : BIT is PSR(13);
alias DIM     : BIT is PSR(14);
alias KNF     : BIT is PSR(15);
alias SC      : BIT_VECTOR(4 downto 0) is
                  PSR(21 downto 17);
alias PS      : BIT_VECTOR(1 downto 0) is
                  PSR(23 downto 22);
alias PM      : BIT_VECTOR(7 downto 0) is
                  PSR(31 downto 24);
```

Pipeline Modeling

	ADDER PIPELINE VARIABLES			OUTPUT VARIABLE
Variable Name	PFAD_STAGE1_RES	PFAD_STAGE2_RES	PFAD_STAGE3_RES	PFAD_REAL_DOUBLE_RES
Instruction				
pfadd.ss f2, f7, f0	Res1 = f2 + f7	?	?	Undefined
pfadd.ss f3, f8, f0	Res2 = f3 + f8	Res1	?	Undefined
pfadd.ss f4, f9, f0	Res3 = f4+ f9	Res2	Res1	Undefined
shl r0,r0,r0	Res3	Res2	Res1	This instruction does not advance the pipeline
pfadd.ss f5, f10, f12	Res4 = f5+ f10	Res3	Res2	Res1 => f12

- Variables contain pipeline states
- Computation is done in first stage when operands are available
- Result is passed through all stages so output occurs on correct clock cycle
- Only pipeline add instruction advances pipeline above

Pipelined Adder Code Segment

```
if (PFAD_FLAG = TRUE) then
```

```
  if(SE = '0') then    -- No pipeline advance on source except.
```

```
    PFAD_SR_STAGE3    := PFAD_SR_STAGE2;  
    PFAD_STAGE3_RES   := PFAD_STAGE2_RES;  
    PFAD_SR_STAGE2    := PFAD_SR_STAGE1;  
    PFAD_STAGE2_RES   := PFAD_STAGE1_RES;  
    PFAD_SR_STAGE1    := PFAD_SR;
```

*Second/Third
Stages Data Passage*



```
  if (PFAD_SR(1) = '0') then -- Single precision source
```

```
    PFAD_REAL_SINGLE_OP1:= INT_TO_REAL_SINGLE(PFAD_OP1(0));  
    PFAD_REAL_SINGLE_OP2:= INT_TO_REAL_SINGLE(PFAD_OP2(0));
```

```
  if (PFAD_SR(0) = '0') then -- Single precision result
```

```
    --
```

```
    -- CASE 1:
```

```
    -- Single Precision sources and Single Precision result
```

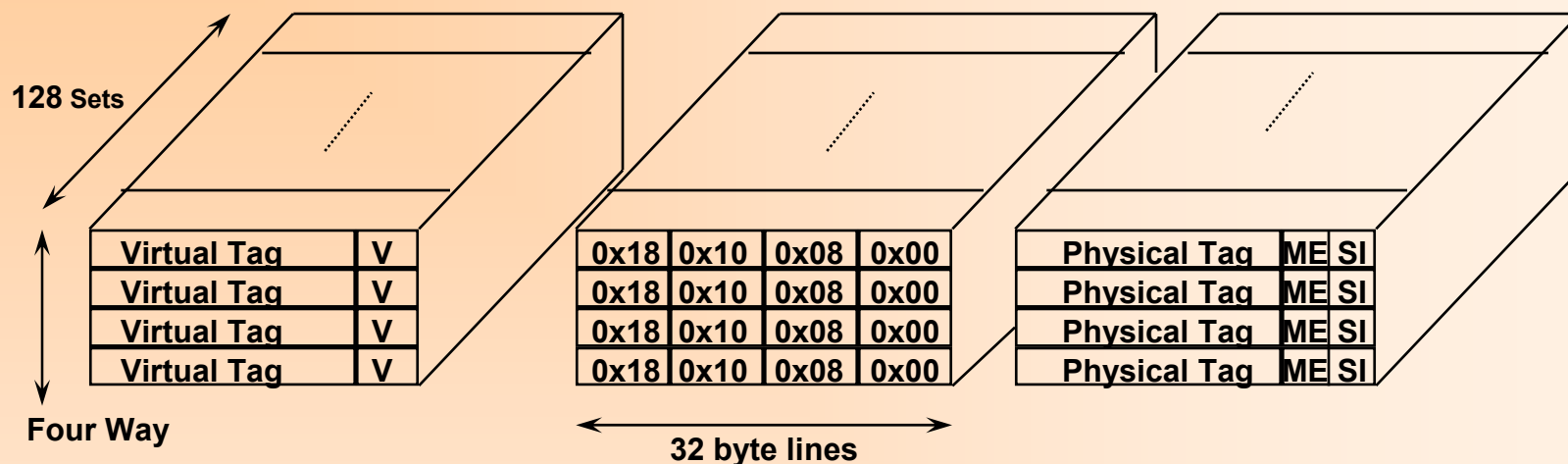
```
    --
```

```
    PFAD_REAL_SINGLE_RES :=PFAD_REAL_SINGLE_OP1 + PFAD_REAL_SINGLE_OP2;  
    PFAD_STAGE1_RES(0) := REAL_SINGLE_TO_INT(PFAD_REAL_SINGLE_RES);
```

```
  else
```

```
    |
```

Cache Modeling



- Record data types used
- Caches contained in three separate processes
 - Instruction Cache => Instr. Fetch process
 - Data Cache => Load/Store process
 - Address Translation Caches => MMU process

Data Cache VHDL Representation

- VHDL representation of the the data cache
- Record type
- Elements of data cache
 - Virtual tags
 - Validity bits
 - Lines of data
 - Physical tags
 - State information
 - MESI

```
subtype BIT_20 is BIT_VECTOR(19 downto 0);
type CACHE_VTAG_BLOCK is array(0 to 3) of BIT_20;
type CACHE_VTAG_SET is array(0 to 127) of
CACHE_VTAG_BLOCK;
type CACHE_BIT_BLOCK is array(0 to 3) of BIT;
type CACHE_V_SET is array(0 to 127) of
CACHE_BIT_BLOCK;
type INT_8D is array(0 to 7) of INTEGER;
type CACHE_LINE_BLOCK is array(0 to 7) of INT_8D;
type CACHE_LINE_SET is array(0 to 127) of
CACHE_LINE_BLOCK;
type CACHE_PTAG_BLOCK is array(0 to 3) of BIT_20;
type CACHE_PTAG_SET is array(0 to 127) of
CACHE_PTAG_BLOCK;
subtype BIT_2 is BIT_VECTOR(1 downto 0);
type CACHE_MESI_BLOCK is array(0 to 3) of BIT_2;
type CACHE_MESI_SET is array(0 to 127) of
CACHE_MESI_BLOCK;
```

```
type DATA_CACHE_TYPE is record
    VIRTUAL_TAG    : CACHE_VTAG_SET;
    V              : CACHE_V_SET;
    INSTR_LINE     : CACHE_LINE_SET;
    PHYSICAL_TAG   : CACHE_PTAG_SET;
    MESI           : CACHE_MESI_SET;
end record;
```

Instruction Cache Search

- Segment of code to search the set and index of the instruction cache for the next instruction
- First check the validity bits
- Second check the virtual tags
- If search is not successful, then start the memory management process

```
while SEARCH = FALSE loop
```

```
-- Search the validity bits and virtual tag if necessary
```

```
if (INST_CACHE.V(SET)(INDEX) = '1') then
```

```
-- Check the virtual tag
```

```
if (INST_CACHE.VIRTUAL_TAG(SET)(INDEX) =  
PC_BIT32(31 downto 12)) then
```

```
SEARCH:= TRUE;
```

```
DATA_VALID:= '1'; -- Instruction in cache
```

```
CACHE_LINE:= INST_CACHE.INSTR_LINE(SET)(INDEX);
```

```
FETCH_BUFF(0) <=
```

```
CACHE_LINE(BITS_TO_NATURAL(PC_BIT32(4 downto 2)));
```

```
end if;
```

```
end if;
```

```
INDEX:= INDEX + 1;
```

```
if (INDEX = 4 and SEARCH = FALSE) then
```

```
SEARCH:= TRUE;
```

```
DATA_VALID:= '0'; -- No instruction in cache
```

```
ADDRESS_INT_SIG <= BITS_TO_INT(PC_BIT32);
```

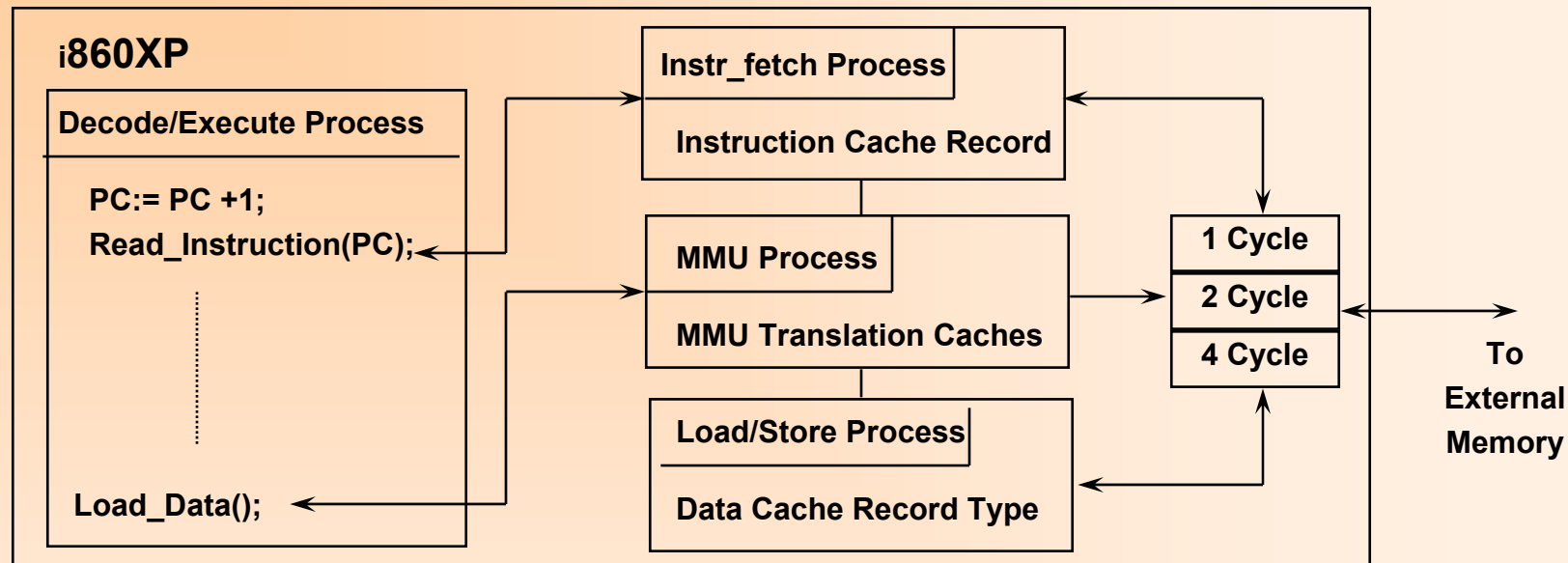
```
MMU_START <= '1', '0' after DEFAULT_TIME;
```

```
FETCH_INSTR_EXT <= '1', '0' after CLK_PERIOD;
```

```
end if;
```

```
end loop;
```

Instruction/Data Reading/Writing



Instruction read:

Decode/Execute => Instr_fetch => Read From Cache

Data Read/Write:

↳ => MMU Translation => Ext_Code_Read

Decode/Execute => MMU Translation => Load/Store => Read from Data Cache

↳ => Ext_Burst_read or Ext_Single_Read

Code Segment for External Instruction Fetch

```

EXT_CODE_READ:
process
  variable DONE          : BOOLEAN;
  variable BURST_FILL   : BOOLEAN:= FALSE;
  variable BRDY_COUNTER: INTEGER:= 0;
  variable i            : INTEGER:= 0;
begin
  EXTERNAL_CODE_READ <= '0';
  wait until EXTERNAL_CODE_READ'EVENT
  and EXTERNAL_CODE_READ = '1';
  wait until FALLING_EDGE(CLK);
  EXTERNAL_CODE_READ <= '1';
  DONE := FALSE;
  BRDY_COUNTER:= 0;
  BURST_FILL := FALSE;
  i:= 0;
  while (DONE = FALSE) loop
    wait on CLK;
    if (RISING_EDGE(CLK)) then
      if (BRDY_N = '0') then
        INSTR_BUFF(i) <= BITS_TO_INT(
          std2bit_vector(DATA(31 downto 0),'0'));
        INSTR_BUFF(i+1) <= BITS_TO_INT(
          std2bit_vector(DATA(63 downto 32),'0'));

```

```

i:= i+2;

```

```

if (KEN_N = '0' or (BURST_FILL=TRUE and
  BRDY_COUNTER /= 3)) then

```

```

  BURST_FILL := TRUE;

```

```

  if (BRDY_COUNTER = 0) then

```

```

    SAVE <= TRUE;

```

```

  else

```

```

    SAVE <= FALSE;

```

```

  end if;

```

```

  BRDY_COUNTER:= BRDY_COUNTER + 1;

```

```

  CACHE_FILL <= '1';

```

```

  INST_LOAD_IN_PROGRESS <= '1';

```

```

else

```

```

  DONE:= TRUE;

```

```

  EXTERNAL_CODE_READ <= '0';

```

```

  CACHE_FILL <= '0' after CLK_PERIOD;

```

```

  INST_LOAD_IN_PROGRESS <=
    '0' after (CLK_PERIOD + 3 ns);

```

```

end if;

```

```

  wait for 0 ns;

```

```

end if;

```

```

end loop;

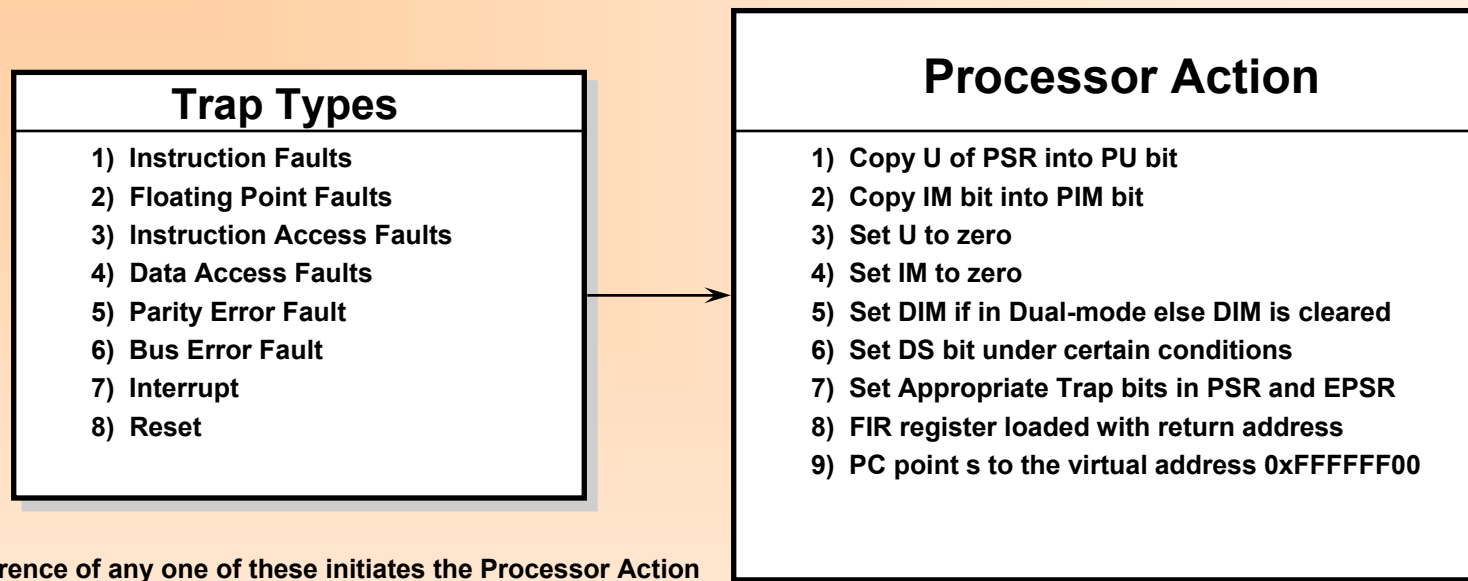
```

```

end process EXT_CODE_READ;

```


Trap/Reset/Exception/Interrupt Handling



- Trap handling is responsibility of programmer
- All trap types are checked on each clock cycle and during execution of FP operations
- All processing contained in a procedural call (TRAP_HANDLER_INIT)

Trap Handler Initiation

```
begin
```

```
-- If a trap occurs after a lock instruction
-- and before a load or store that unlocks the
-- instruction then set the IL bit in the EPSR register.
-- This can be determined by checking the BL bit of
-- the DIRBASE register which is bit 4.
```

```
if(DIRBASE(4) = '1') then
```

```
-- Set the IL bit of the EPSR to '1'
```

```
EPSR(13):= '1';
```

```
end if;
```

```
-- Copy U of the PSR to PU of the PSR
```

```
-- and set U to '0'. U is bit 6 and PU is bit 7
```

```
PSR(7):= PSR(6);
```

```
PSR(6):= '0'; -- Supervisor mode
```

```
-- Copy IM into PIM of PSR and set IM to '0'
```

```
-- IM is bit 4 and PIM is bit 5
```

```
PSR(5):= PSR(4);
```

```
PSR(4):= '0'; -- Interrupts disabled
```

```
if(DUAL_MODE = TRUE) then
```

```
-- Set DIM of PSR (bit 14)
```

```
PSR(14):= '1';
```

```
else -- Clear it
```

```
PSR(14):= '0';
```

```
end if;
```

```
if((FIRST_TIME = TRUE) or
(LAST_TIME = TRUE)) then
```

```
-- Set DS bit of PSR (bit 13)
```

```
PSR(13):= '1';
```

```
else
```

```
-- Clear DS bit of PSR
```

```
PSR(13):= '0';
```

```
end if;
```

```
-- If PEF (BIT 15 of EPSR) or
```

```
-- BEF (BIT 16 of EPSR) is set then
```

```
-- Place the bus address in the BEAR register
```

```
if(EPSR(15) = '1' or EPSR(16) = '1') then
```

```
BEAR(31 downto 3):= std2bit_vector(ADDRESS,'0');
```

```
end if;
```

```
-- Load the address of the trapped instruction
-- into the FIR register
```

```
INT_TO_BITS(TRAP_ADDR, FIR);
```

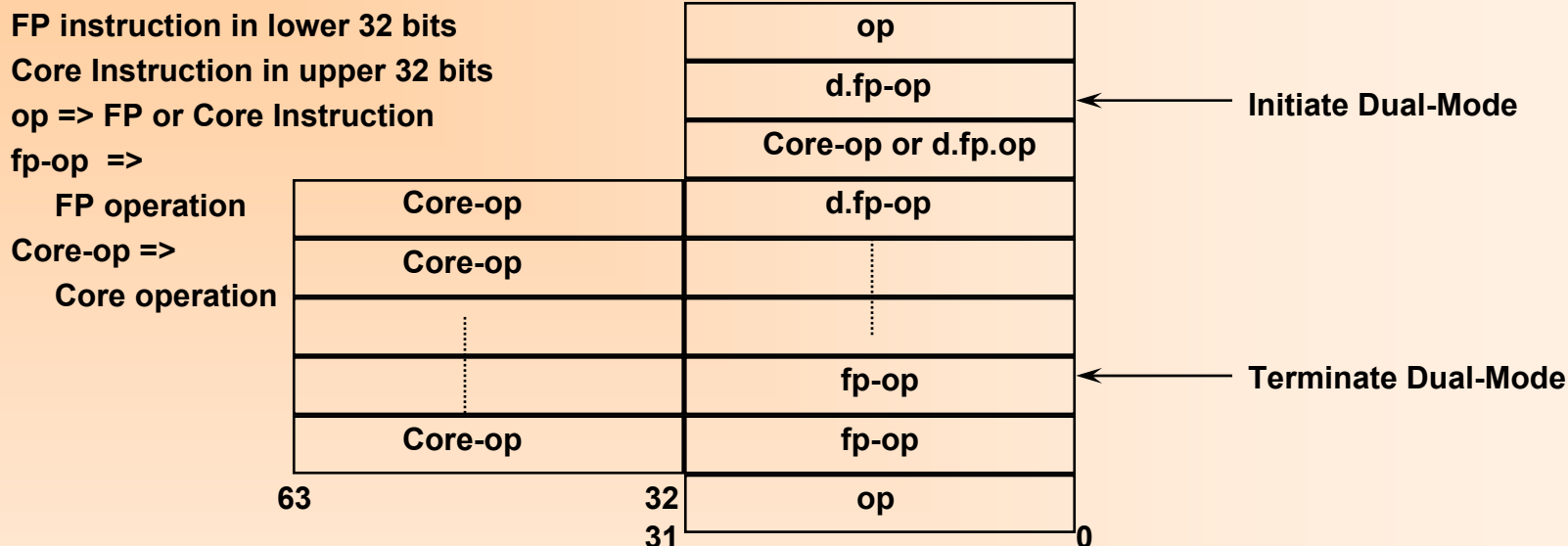
```
FIR(31 downto 0):= FIR(29 downto 0) & "00";
```

```
DUAL_MODE:= FALSE;
```

```
PC <= VIRTUAL_ADDR;
```

```
end TRAP_HANDLER_INIT;
```

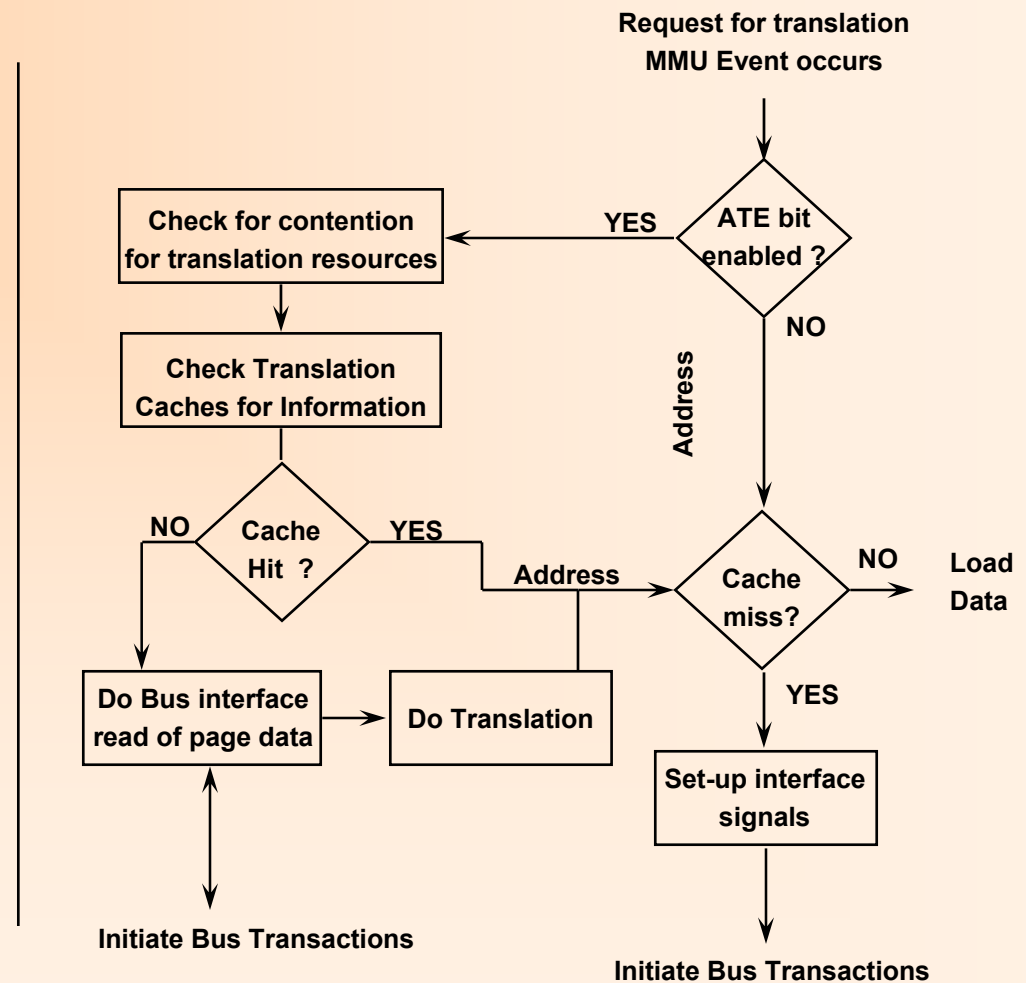
Dual Instruction Sequencing



- Variable set to TRUE when FP instruction has Dual-Mode bit set
- DUAL-MODE variable is set after the next execution
- When in dual-mode, instruction interactions are first checked before executing the FP and core instructions
- Terminating Dual-Mode uses two variables and is similar to initiating

Memory Management

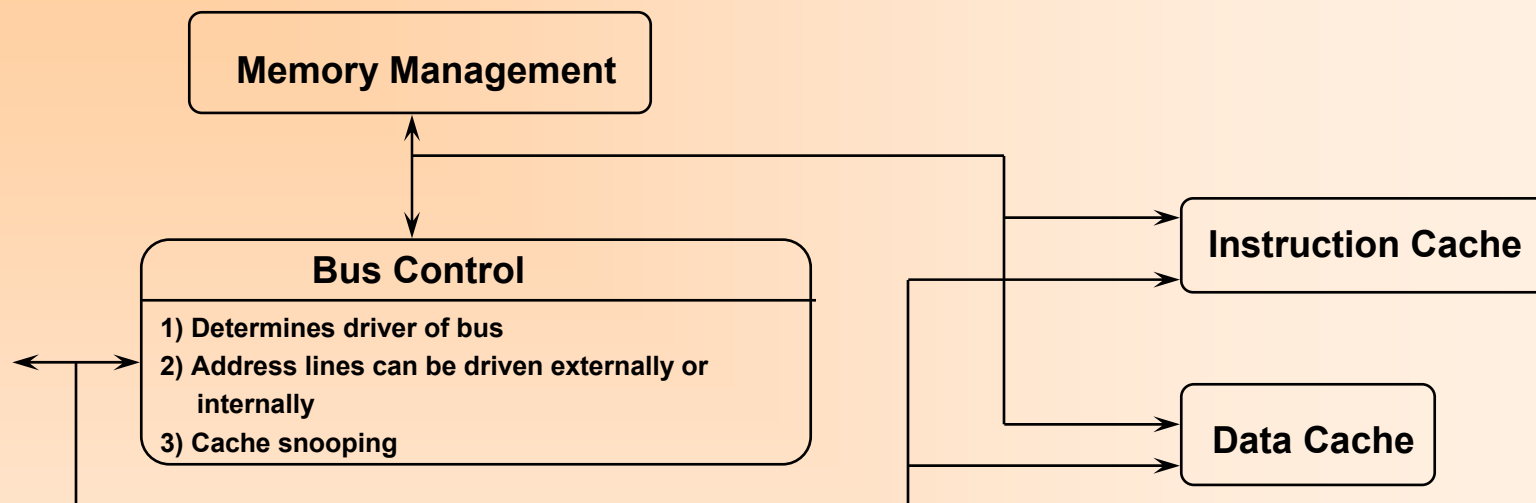
- Contains 2 translation caches (4k and 4M)
- Resolves data and instruction fetch conflicts
- Data accesses always translated, instruction accesses only on cache miss
- Separate process for MMU
- Three types of transactions
 - Code Read/Write
 - Data Read/Write
 - Page Directory Read



Bus Interface Modeling Scope

- Components interface to external world
- Perform handshaking protocols to external components
- Tasks of the interface model
 - Timing accuracy with respect to data manual
 - Set-up and hold checks
 - Pulse-width checks
 - Assertions on unknown values for control inputs
 - Clock, Reset
 - Timing performance values (min, typ, max) defined in separate package
 - Rise/fall times and capacitive effects can be included
- Clearly specified in DOD Deliverable Data Item Requirements (DI-EGDS-80811) or EIA-567A

Bus Control



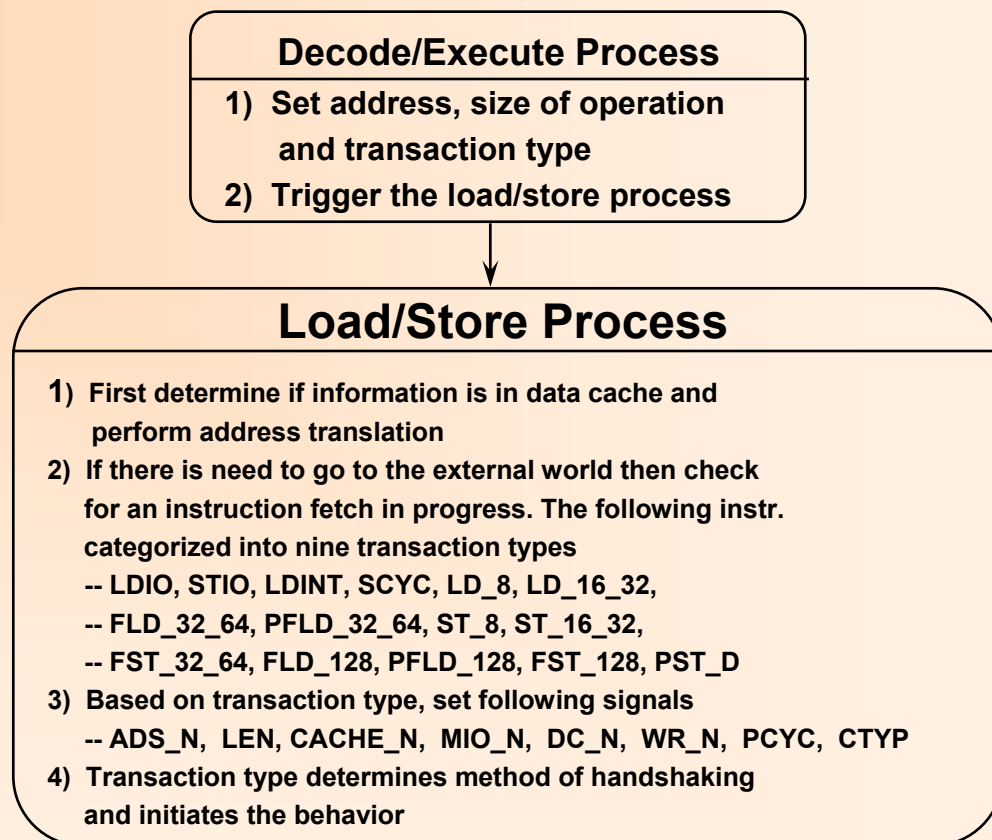
- Monitors signals from MMU and externally driven address and control lines
- Performs cache snooping checks
- Interfaces closely with bus interface model
- Checks for parity errors

Interface Types

- Address strobe initiated
- Bus arbitration activities
- Cache snooping protocol
- Miscellaneous
 - Clock, reset, lock, interrupt, etc.

Data Load/Store Initiated Activities

- Address-strobe-initiated interface protocol
- Decode/execute and load/store processes interact
- Load/store triggers one of 3 processes based on burst transfer mode
- Total of nine transaction types
- Load/store process contains data cache



Setup of External Signals for Data Transfer

```
case TRANSACTION_TYPE is
```

```
when 0 =>           -- LDIO instruction
    LEN    <= '0';
    CACHE_N <= '1';
    -- The next three signals identify this
    instruction
```

```
    MIO_N  <= '0';
    DC_N   <= '1';
    WR_N   <= '0';
```

```
when 1 =>           -- STIO instruction
    LEN    <= '0';
    CACHE_N <= '1';
    -- The next three signals identify this
    instruction
```

```
    MIO_N  <= '0';
    DC_N   <= '1';
    WR_N   <= '1';
```

```
case TRANSACTION_TYPE is
```

```
-- Store instruction
```

```
when 1| 5 =>
```

```
    case BYTE_NUM_SIG is
```

```
    when 0 =>
```

```
        case OP_SIZE_SIG is
```

```
        when 8 =>
```

```
            BE_N <= "11111110";
```

```
        when 16 =>
```

```
            BE_N <= "11111100";
```

```
        when 32 =>
```

```
            BE_N <= "11110000";
```

```
        when 64 =>
```

```
            BE_N <= "00000000";
```

```
        when others =>
```

```
            assert FALSE
```

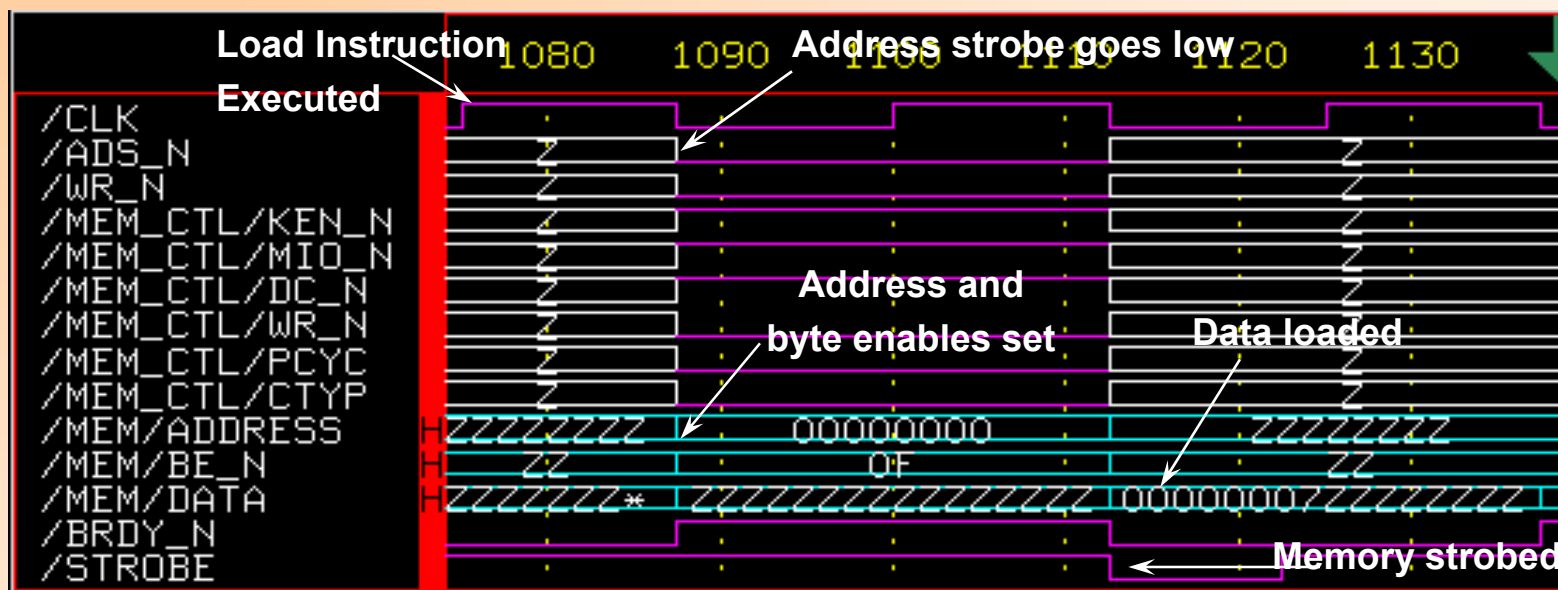
```
            report "Illegal Op size in store instruction 1 or 5"
            severity warning;
```

```
        end case;
```

ADS_N Strobe- Initiated Memory Write

```
INT_TO_BITS(DATA_ADDR_SIG,TEMP_32);
ADDRESS <= To_StdLogicVector(TEMP_32(31 downto 3));
-- PCYC and CTYP are only defined for memory reads/writes
ADS_N <= '0','1' after CLK_PERIOD;
-- Trigger the external read process
EXTERNAL_BUS_OP <= '1';
-- Now load the data onto the data bus
INT_TO_BITS(BUS_DATA_SIG_2INT(0),BUS_DATA_64(31 downto 0));
INT_TO_BITS(BUS_DATA_SIG_2INT(1),BUS_DATA_64(63 downto 32));
DATA <= To_StdLogicVector(BUS_DATA_64);
wait until RISING_EDGE(CLK);
wait until BRDY_N'EVENT and BRDY_N = '0';
EXTERNAL_BUS_OP <= '0';
DATA_STORE_IN_PROGRESS <= '0';
DATA <= (others => 'Z') after CLK_PERIOD/2;
```

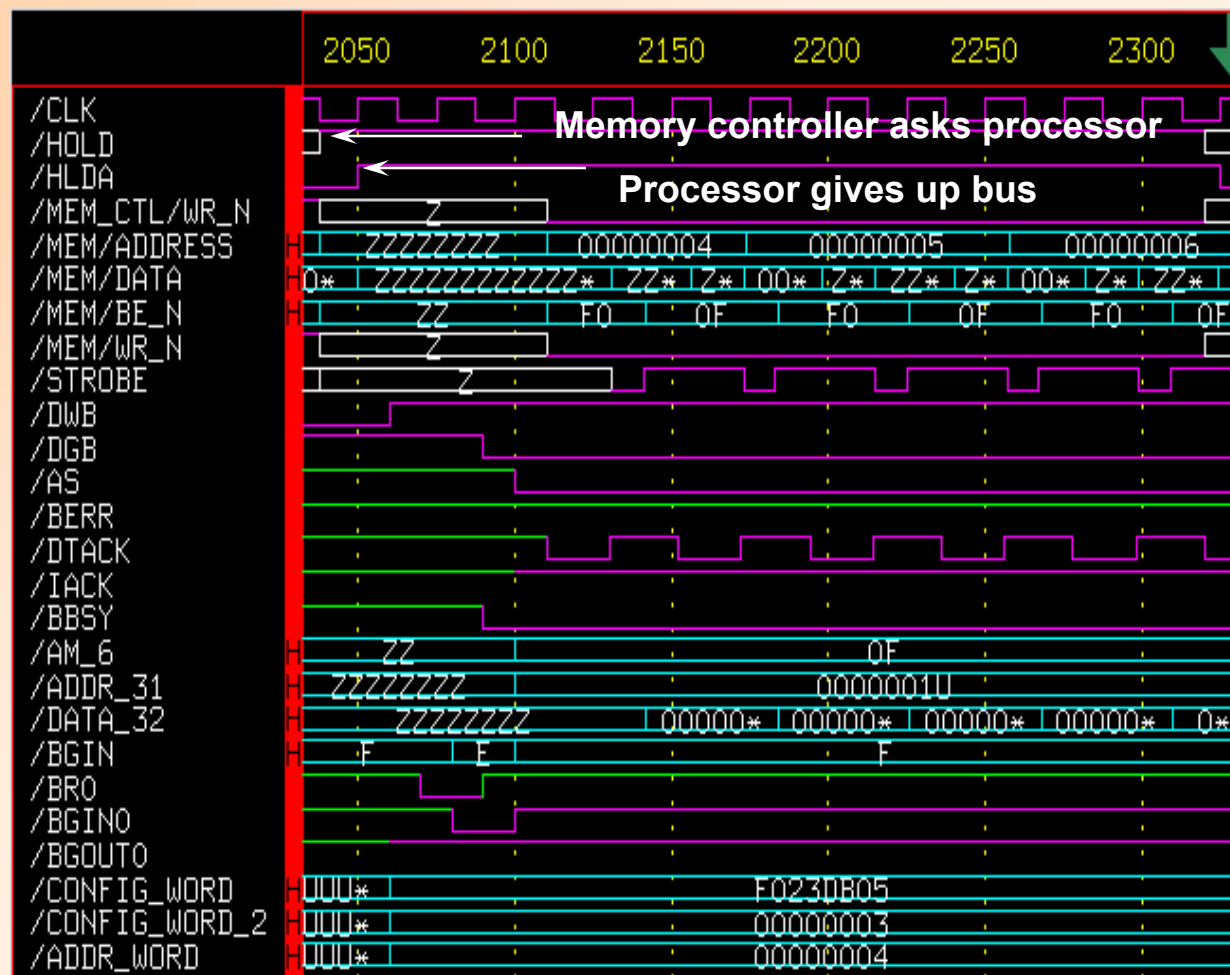
Timing Diagram for Load Operation



- Three clock cycles to perform the data store because of memory controller wait states
- Clock prior to ADS_N active initiates the store operation
- BRDY_N active low from the memory controller signifies data is ready
- ADDRESS and BE_N lines determine location in memory

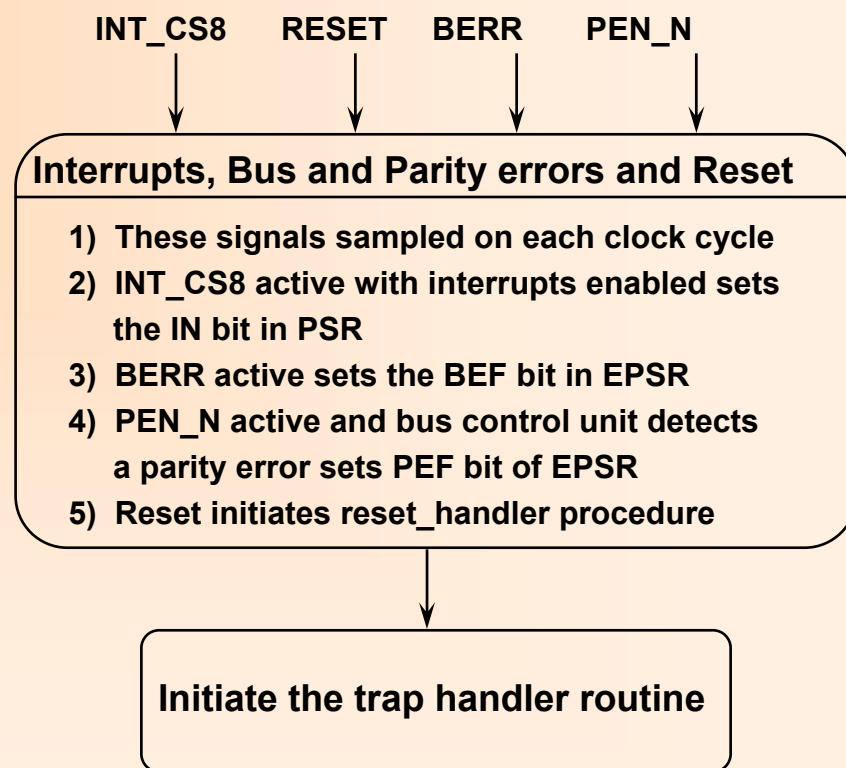
Bus Arbitration Timing

- Cache miss occurred at same time as controller- initiated write to VME. Conflict for address lines
- HOLD to processor prevented two drivers of bus
- Process acknowledges HOLD and waits for HOLD to go inactive again
- Controller performs its block of writes and de-asserts HOLD
- Processor continues with cache fill



Interrupts/Bus Errors/Parity Errors/Reset

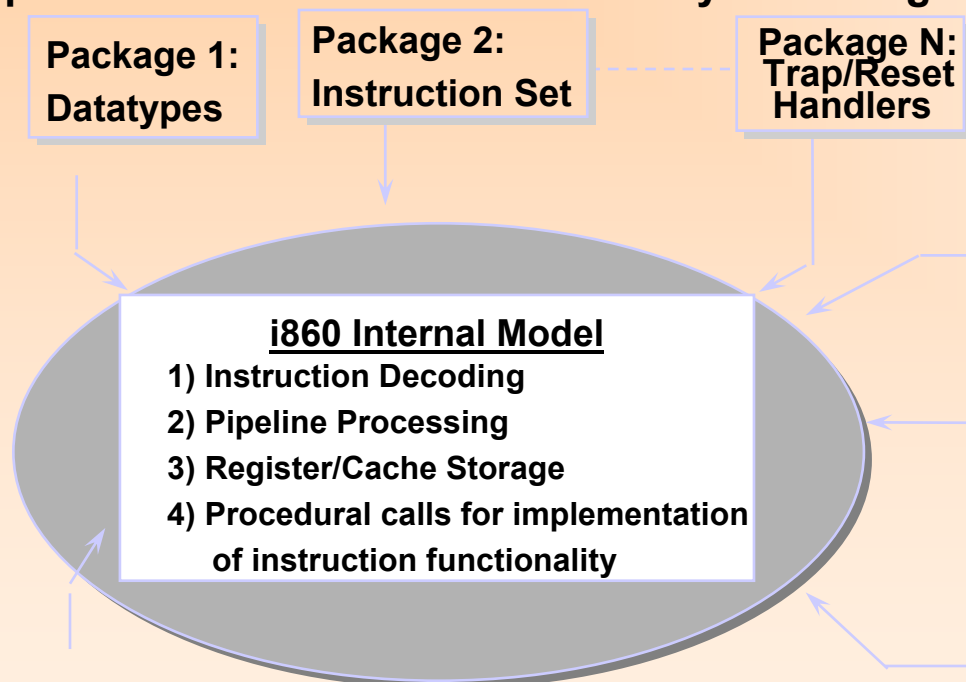
- Miscellaneous input signals
- Bus control unit checks for parity errors if PEN_N is active on bus read operation
- Interrupt signal sampled high with IM bit set in PSR causes trap invocation
- The trap handler initialization routine is a procedure
- All signals sampled on rising edge of clock
- Reset must be sampled high for 10 clock cycles.



Case Study: i860XP Emulation Test bench

i860 Detailed-behavioral Model

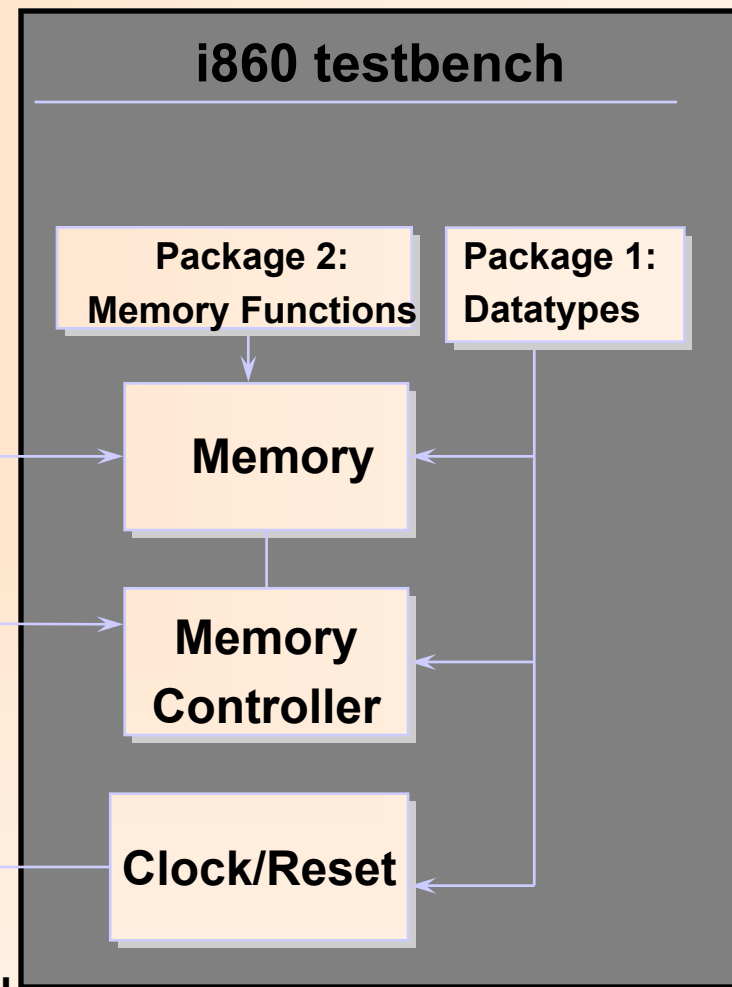
Encapsulation of Common Functionality in Packages



i860 Interface Wrapper

- 1) Interface Timing as specified in users manual

i860 testbench



Testbench Modeling

- Why ?
 - Necessary for verification of the component model
- What ?
 - Provides synchronization inputs for clock circuits
 - Initializes the component to a known state with reset
 - Stimulates the component to verify
 - Internal functionality
 - External timing
 - Observes responses resulting from stimuli
 - Use of files for internal state dumps
 - Storage elements to hold results of interface information
 - Results database for timing checks
 - Compares response values to expected values contained in a test bench entity-architecture
 - Provides for automated regression testing

Elements of the Testbench

- Clock/reset
 - Provides the initialization and synchronization
- Memory controller
 - Provides handshaking and special signals to the processor
- Memory
 - Stores results and provides test program
- Test program
 - Test code compiled from high-level language
 - Special instructions can also be used as input to the memory and memory controller for monitoring and dumping of specific ranges to a file for viewing

Clock/Reset Generator

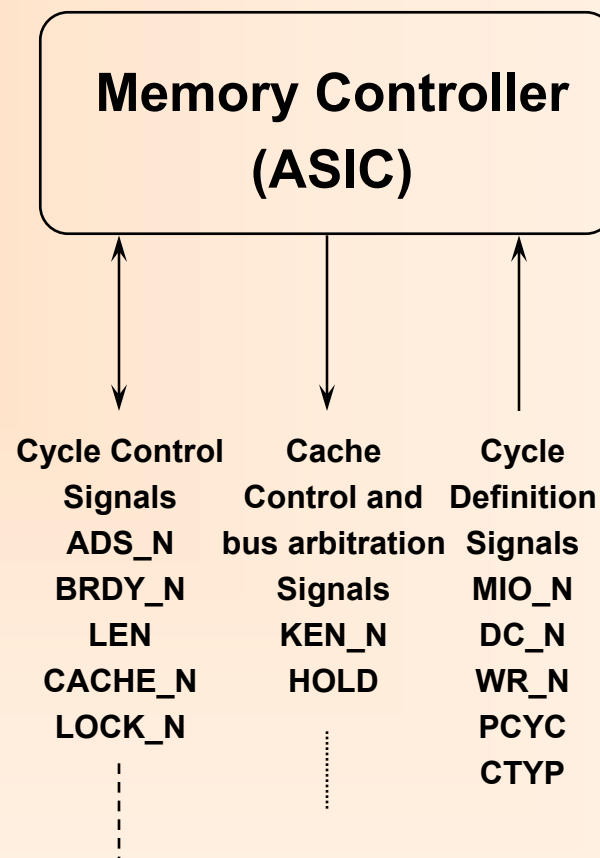
- Initialize with a reset
 - Held high for 10 clock periods using counter variable
- Clock period set using generic parameter, HALF_CLK_PERIOD
- Reset held high based on generic input

```
entity CLK_RESET_GEN is
  generic (HALF_CLK_PERIOD: TIME:= 12.5 ns;
           RESET_CYCLES   : INTEGER:= 10;
           ACTIVE_VALUE   : STD_LOGIC:= '1');
  port (CLK   : out STD_LOGIC:= '0';
        RESET: out STD_LOGIC);
end CLK_RESET_GEN;
architecture BEHAVIOR of CLK_RESET_GEN is
```

```
begin
  CLK_GENERATOR: process
    variable RESET_TOGGLE: BOOLEAN:= FALSE;
    variable COUNTER      : INTEGER:= 0;
  begin
    if RESET_TOGGLE = FALSE then
      RESET <= ACTIVE_VALUE;
    end if;
    CLK <= '1';
    wait for HALF_CLK_PERIOD;
    CLK <= '0';
    wait for HALF_CLK_PERIOD;
    COUNTER:= COUNTER + 1;
    if (COUNTER = INTEGER'HIGH) then
      COUNTER:= RESET_CYCLES;
    end if;
    if (COUNTER > RESET_CYCLES and
        RESET_TOGGLE = FALSE) then
      RESET <= not ACTIVE_VALUE;
      RESET_TOGGLE:= TRUE;
    end if;
  end process;
end BEHAVIOR;
```

Memory Controller

- Interfaces directly to the Proc.
 - Interprets processor output signals to determine transaction method
 - Drives processor input signals to define specific transfer types (KEN_N for cacheable addresses)
- Hierarchy of case statements used to decode address-strobe-initiated types
- Drives address lines on cache snooping and burst mode data and instruction transfers
- Interrupts processor
- Performs bus arbitration protocol



Address-Strobe-Initiated Bus Cycle Decoding

- Use of case statements for decoding the type of handshaking
- Use of memory, I/O transfer type cycle definition
- Allows for Transfers of 1, 2 or 4 words based on the type

```
if (ADS_N = '0') then
  CYCLE_DEFINITION:= std2bit_vector(MIO_N & DC_N & WR_N, '0');
  case CYCLE_DEFINITION is
    when "000" =>  -- Interrupt acknowledge
    when "001" =>  -- Special cycle
    when "010" =>  -- I/O Read

    .....

    when "110" =>  -- Memory Read
      MEMORY_TRANSFER_TYPE:=
        std2bit_vector(PCYC & CTYP & WR_N, "0");
      case MEMORY_TRANSFER_TYPE is
        when "000:" =>  -- Normal read

        .....

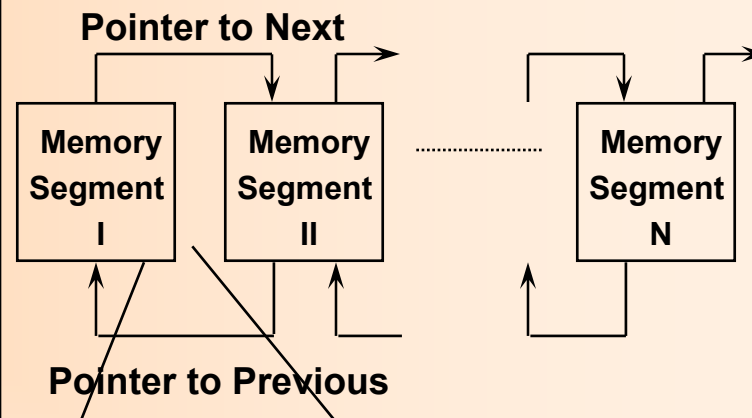
        when "111" =>  -- Page Table update
      end case;

      .....

    end case;
  end if;
```

Memory Model

- Each memory segment is dynamically allocated based on address need
- Consists of a doubly linked list of records
- Provide access time generic parameter
- Procedures are created to:
 - Create the initial memory segment after reset
 - Add memory segments
 - Search segments for data
- Record consists of:
 - Low and high address for segment
 - Data array
 - Next and Previous pointers to memory segments



```
type MEM_TYPE is array(0 to 4095) of integer;
type MEM_SEGMENT;
type MEM_SEG_PTR is access MEM_SEGMENT;
type MEM_SEGMENT is
  record
    low_addr  : INTEGER;
    high_addr : INTEGER;
    memory    : MEM_TYPE;
    prev_ptr  : MEM_SEG_PTR;
    next_ptr  : MEM_SEG_PTR;
  end record;
```

Creating and Deleting Memory Segments

- Functions are used to create and delete memory segments
 - On reset, the delete procedure is called to put all memory into its reset state.
- MEM_SEG_PTR is an access record type

```
function CREATE_MEM_SEG  
    return MEM_SEG_PTR is
```

```
begin
```

```
    return new
```

```
        MEM_SEGMENT'(0,0,(others =>0),null,null);
```

```
end CREATE_MEM_SEG;
```

```
procedure DELETE_MEM(  
    MEM_ARRAY : inout MEM_SEG_PTR) is  
    variable PREV_SEG: MEM_SEG_PTR;  
    variable NEXT_SEG: MEM_SEG_PTR;  
begin  
    -- Move to the end of memory  
    while MEM_ARRAY.NEXT_SEG_PTR /= null loop  
        NEXT_SEG:= MEM_ARRAY.NEXT_SEG_PTR;  
        MEM_ARRAY:= NEXT_SEG;  
    end loop;  
    -- Deallocate all segments starting at the end  
    while MEM_ARRAY.PREV_SEG_PTR /= null loop  
        PREV_SEG:= MEM_ARRAY.PREV_SEG_PTR;  
        deallocate(MEM_ARRAY);  
        MEM_ARRAY:= PREV_SEG;  
    end loop;  
end DELETE_MEM;
```

Adding a New Memory Segment to Existing Memory

- This procedure adds a new memory segment to existing memory
- Pass filename and current memory array
- Return new memory array
- Attach the new segment to the end of the current memory array
- Allocate using the new operator

```
procedure ADD_NEW_MEM_SEG(  
    FILENAME      : MEM_FILENAME;  
    MEM_ARRAY     : inout MEM_SEG_PTR) is  
  
    variable PREV_SEG : MEM_SEG_PTR;  
    variable NEXT_SEG : MEM_SEG_PTR;  
  
begin  
    while (MEM_ARRAY.NEXT_SEG_PTR /= null) loop  
        NEXT_SEG:= MEM_ARRAY.NEXT_SEG_PTR;  
        MEM_ARRAY:= NEXT_SEG;  
    end loop;  
    PREV_SEG:= MEM_ARRAY;  
    NEXT_SEG:= new  
        MEM_SEGMENT'(0,0,(others =>0),null,null);  
    -- Replace the null value pointer with the new  
    -- allocated value  
    MEM_ARRAY.NEXT_SEG_PTR:= NEXT_SEG;  
    MEM_ARRAY:= NEXT_SEG;  
    MEM_ARRAY.PREV_SEG_PTR:= PREV_SEG;  
    MEM_ARRAY.NEXT_SEG_PTR:= null;  
    LOAD_MEM(FILENAME, MEM_ARRAY);  
  
end ADD_NEW_MEM_SEG;
```

Loading a Memory Segment

- This procedure loads the new memory segment with instructions or data
- The *Filename* variable contains the file to use for loading
- The file has the format
 - Low address of segment
 - High address of segment
 - Data to be read if any

```
procedure LOAD_MEM(  
    FILENAME      : in MEM_FILENAME;  
    MEM_ARRAY : inout MEM_SEG_PTR) is  
    file DATA_FILE : TEXT is in FILENAME;  
    variable IN_LINE : LINE;  
    variable NO_DATA : INTEGER;  
begin  
    NO_DATA:= 0;  
    -- First, read the address range of this data file  
    readline(DATA_FILE,IN_LINE);  
    read(IN_LINE, MEM_ARRAY.LOW_ADDR);  
    readline(DATA_FILE,IN_LINE);  
    read(IN_LINE, MEM_ARRAY.HIGH_ADDR);  
    while not(endfile(DATA_FILE)) loop  
        -- read the data from a file into memory  
        readline(DATA_FILE,IN_LINE);  
        read(IN_LINE, MEM_ARRAY.MEMORY(NO_DATA));  
        NO_DATA:= NO_DATA + 1;  
    end loop;  
end LOAD_MEM;
```

Test Strategies

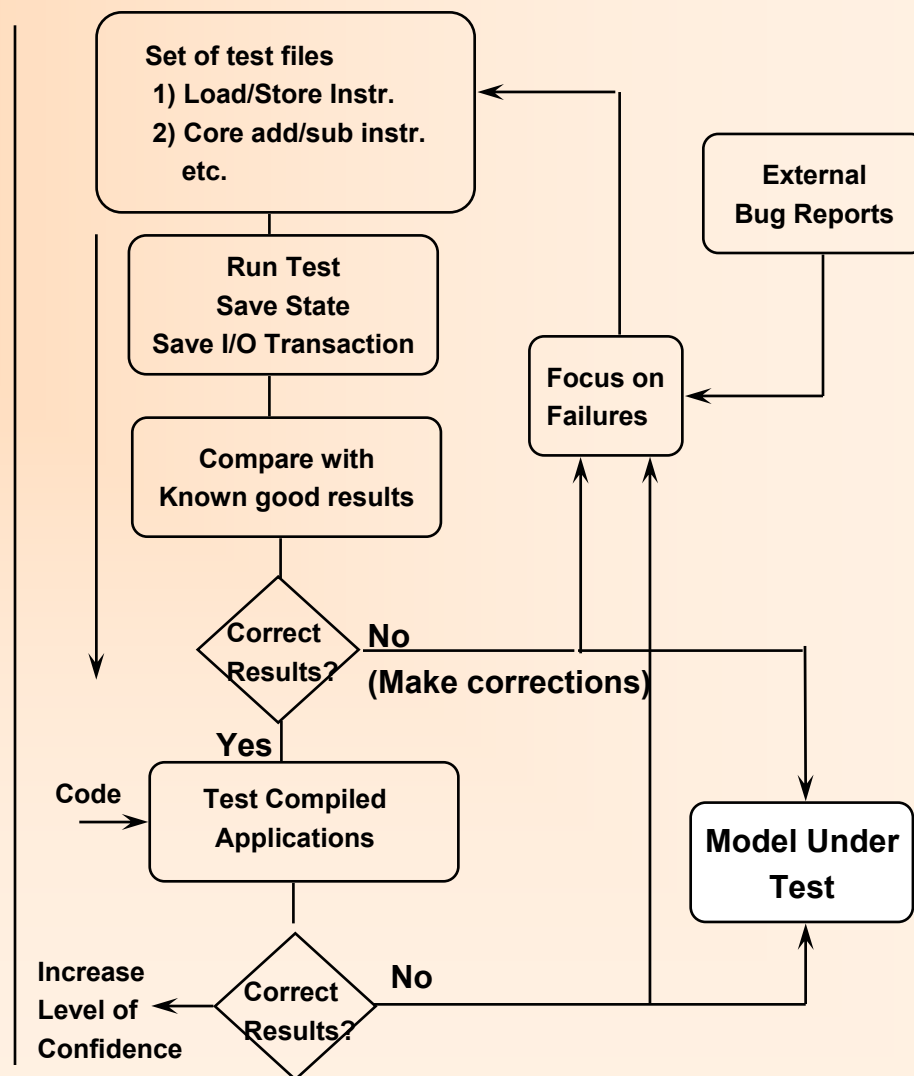
- Build test matrix to obtain acceptable coverage of functionality
- Provide at least one test for each instruction type and interface behavior
- Generate test applications from compiled C code
- Use script files to automate testing
- Perform regression tests for model changes
 - Should be script based to run quickly
- Store results for later comparisons
 - Internal state dumped to files for later observation
 - Results database of timing information

Testing the i860 Emulation

- Verify performance and interface operability
 - Does it meet specification in the manual?
 - Does everything talk to each other correctly?
- Build a test matrix
 - Is internal functionality correct?
 - Does external interface timing match data sheets?
- Test matrix design
 - Use files of short programs to test each instruction
 - Use compiled code to test applications with a large number of instruction interactions

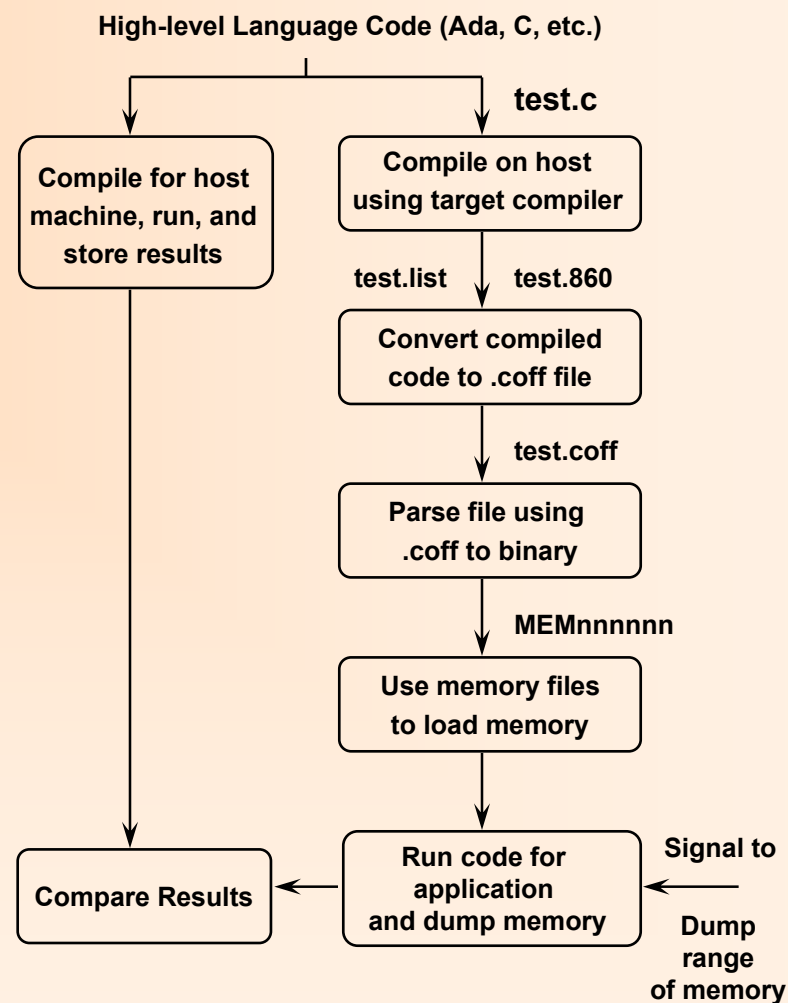
Test Matrix and Regression Test Strategy

- Use suite of test files
- Run tests from a script and store results
- Compare results with known good data
- If passed initial tests, run application code examples and compare with known results
- Code rework has two sources
 - External bug reports
 - Regression test failures
- Inner loop testing done before outer loop

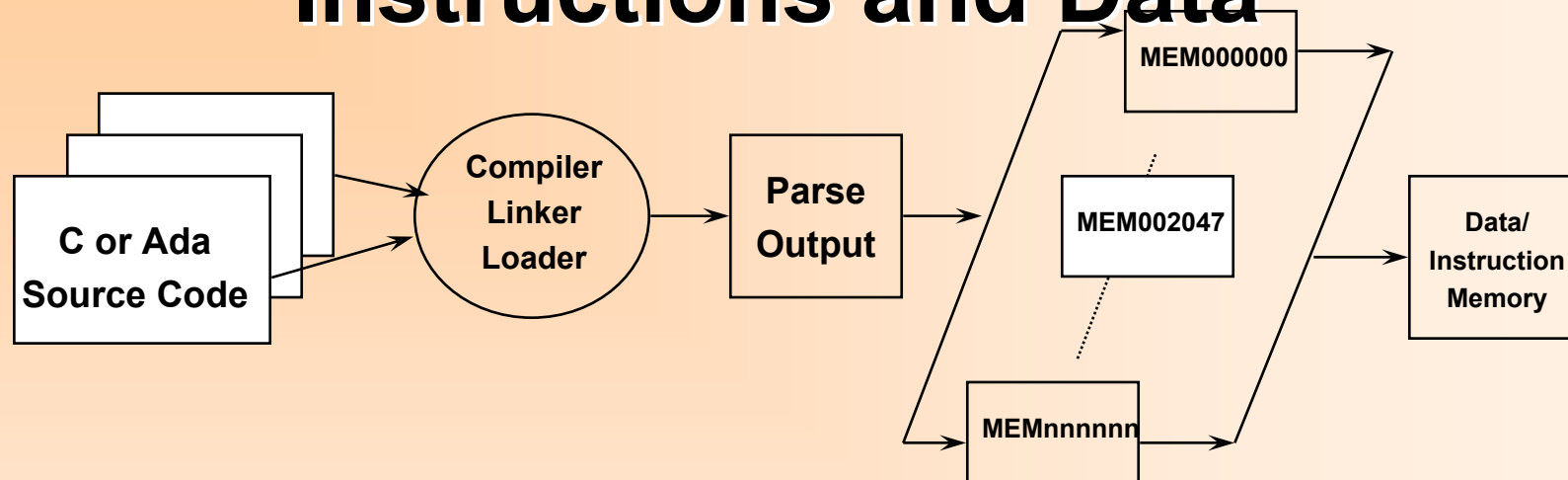


Building Test Application Code

- See flow chart of procedure used to build test application code
 - High-level language compiled and run on host machine
 - Compiled on host for the target
 - Code converted to memory files for the i860 using target tools and parsing routine for the common object file format file (.coff)
- Run code on model and compare results to host data



Loading Memory with Instructions and Data



- Use high-level source code (C, Ada, etc.)
- Compile with target compiler
- Parse output to place information in memory files for the memory model (binary format goes into memory)
- Read by memory and store in dynamically allocated list of arrays. Set by address lines.

Simulation Results

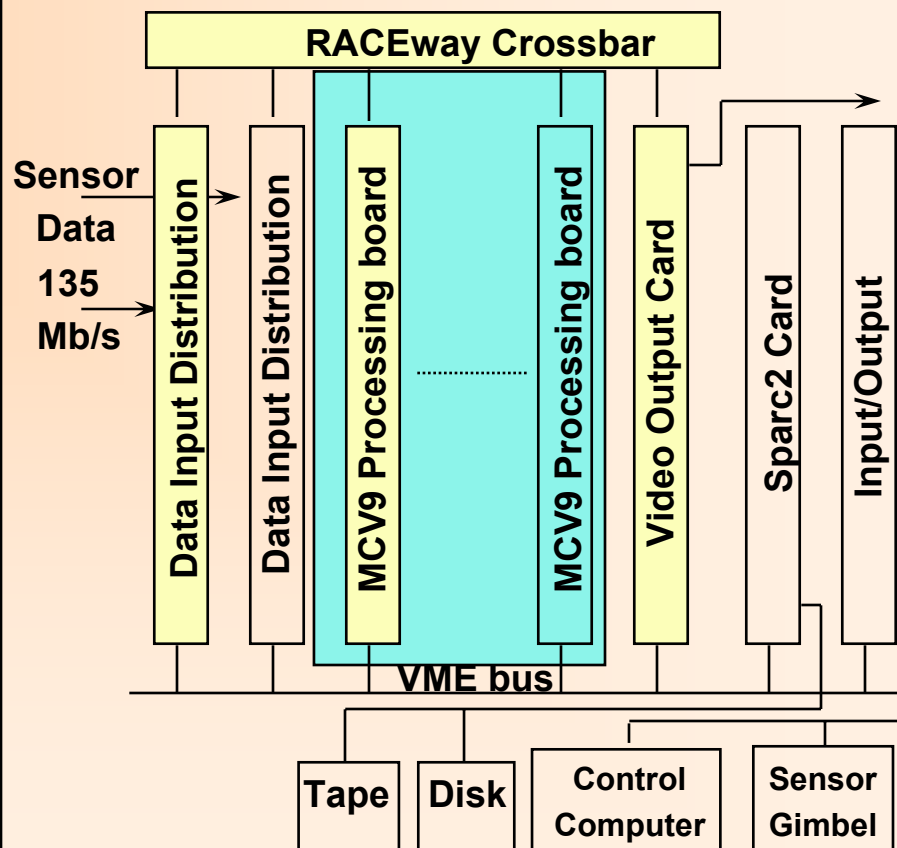
- Three applications tested
- Avg. instructions/sec executed => 220
- Hardware was Sun Sparc10 with 128 Mbytes memory
- Core instructions faster to simulate than load/stores

Appl	Wall Clock Run Time	Internal Simulation Time	Instr/sec.
Fahr2Cel	5.1 sec.	35 us	216
FIR	35 sec.	280 us	211
Sobe	70 min.	31.8 ms	234

Case Study: IRST System Virtual Prototype (Published in IEEE *Micro*, Fall 1995)

• Hardware Elements

- Data Input Distribution card
- RS-170 Daughter card
- Sensor Interface card
- Video output card
- MCV9 processing boards
- VME interface
- RACEway XBAR network
- 190 processors in total system and only a fraction of them were modeled at this level (1-16 processor tests)



Case Study MCV9: Subsystem-level Test

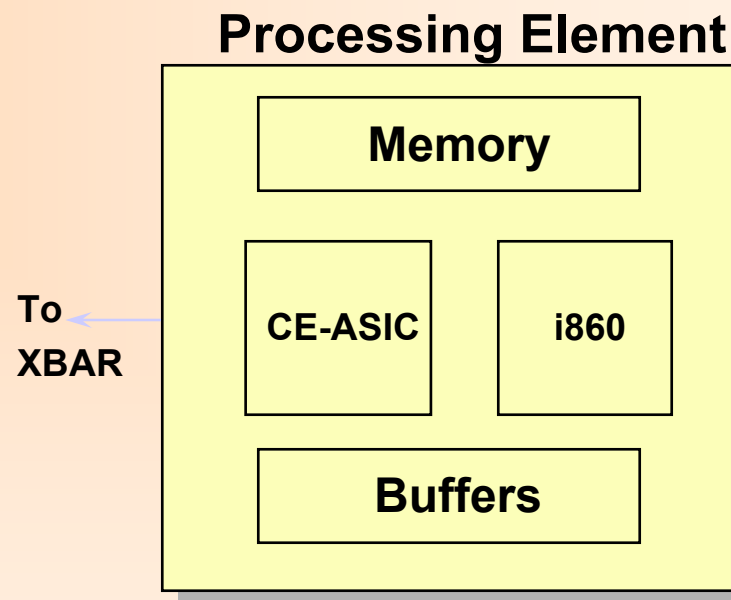
- Objectives of Subsystem Testing
 - Does each component model behave correctly stand-alone?
 - Is the handshaking protocol between components or buses functioning?
 - Are the data rates between components within the specified limits?
 - Does the prototype provide a high level of confidence for HW prototyping?

Case Study MCV9: Test Process

- Phase I: Initial Integration (Processing Element)
 - i860 \Leftrightarrow CE-ASIC \Leftrightarrow Memory
- Phase II:
 - PE \Leftrightarrow XBAR
- Phase III:
 - Multiple XBARs
 - Communications with boundaries
 - VME
 - RACEway
 - Added VME Driver and Interlink at this point
 - Multiple Processing Elements

Case Study MCV9: Phase I Integration Objectives

- Tests at this stage
 - Reset
 - Did anything reset to an unexpected state?
 - CE-ASIC registers
 - Can the i860 set key registers in the CE-ASIC?
 - Reading/Writing to CE-ASIC
 - Is the handshaking logic working between the i860 and the CE-ASIC?
 - Memory
 - Can we read and write data to the memory?



**I860 could be replaced by
G4 or future G5....**

Case Study MCV9: Phase II/III

Integration Objectives

- Interface a single XBAR to the PE, then multiple XBARs
- Check Communications at boundaries
- Two major interfaces
 - VME
 - VME bus model added to test bench
 - RACEway
 - Interlink model added to test bench
- Full MCV9 16 processor instantiation
 - Data written between processors

Case Study MCV9: Phase II/III Integration Tests

- Tests at this stage
 - i860 <=> XBARs <=> RACEway <=> Interlink
 - Verify writing data to RACEway from the i860
 - i860 <=> XBARs <=> VME <=> VME Driver
 - Verify writing data to the VME interface
 - i860 <=> XBARs <=> i860
 - Verify multiprocessor communication
- VME used for passing control information
- RACEway used for passing video data

Case Study MCV9: Simulation Results

Test Case	Internal Simulation Time	Instr/sec.
MCV9 to Interlink	1200 ns	7.5
MCV9 to VME Driver	3000 ns	5.5

- Application code requires significant simulation time
- Diagnostic and test code may be less compute intensive
- Test and Diagnostic can represent the majority of code

Case Study IRST: System Tests

- Software Reset
 - Verify all boards could be reset via software
 - Specific registers should be configured with correct default values
- VME Register Test
 - SW written to read and write all registers on data distribution and video output cards via VME
 - Known pattern placed in memory if tests are passed

Case Study IRST: System Tests (cont.)

- Floating Point RAM Test
 - Same as the previous RAM test, but different portion of RAM dedicated for FP
- Interrupt Tests (to check correct behavior of HW and SW responses)
 - FIFO Overflow
 - Data Overflow
 - Beginning of Frame
 - End of Frame

Case Study IRST: System Tests (cont.)

- RAM test
 - Read/Write portions of RAM on the video card via VME
 - 128 locations written, then read back
 - Test disclosed major design error in VME logic of video and data distribution card
 - Misinterpretation of VME specification with respect to address lines A1 and A2; i.e., they were not used for decoding and therefore limited addressing to 32-bit locations

Case Study IRST: Simulation Results

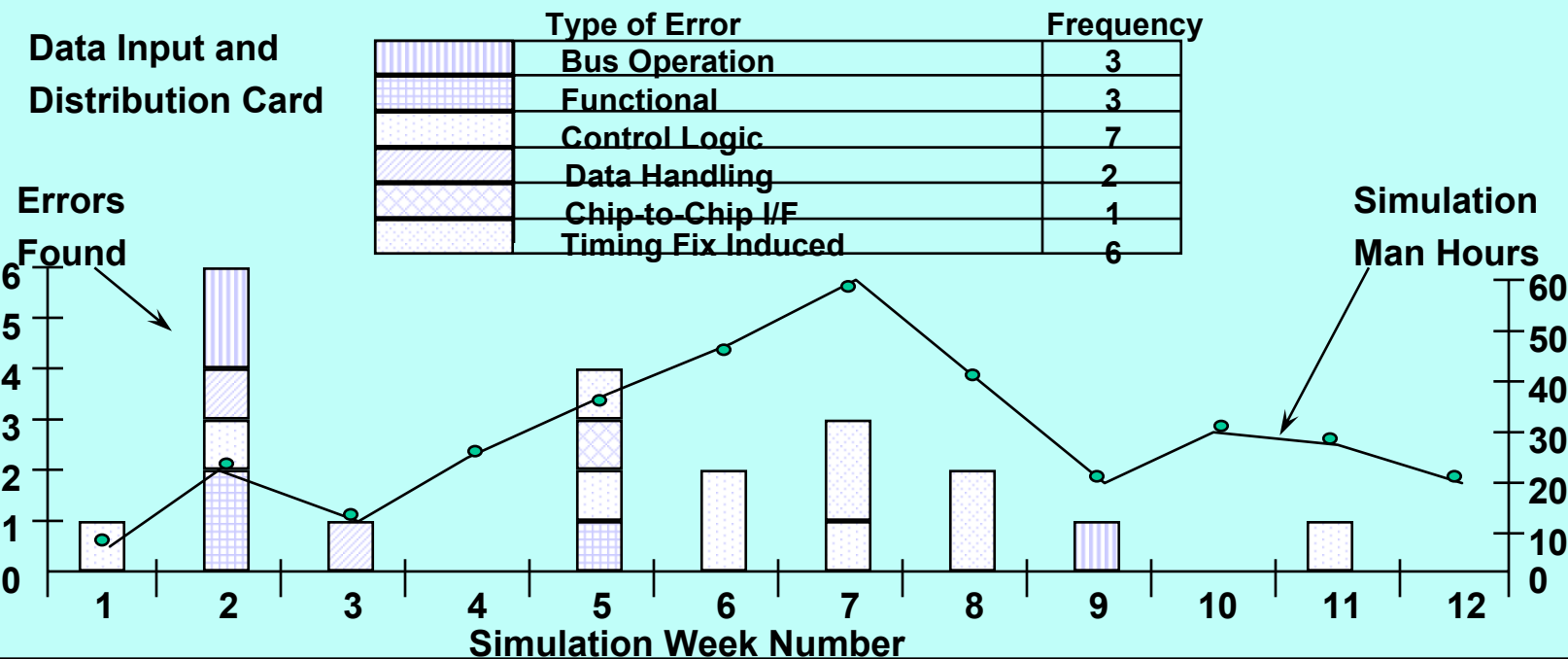
- Events happening far into simulation timeline not able to be seen
- Large amount of disk space required for results database storage
- Test plan must be established early in the prototyping effort to account for long simulations
- Methods must be devised to stop long simulations if bad results occur early in the simulation

Class of Machine: Sparc10 /128 Meg. RAM / 1.2G Disk
Signals archived: 1,048

Simulation Time (msec)	CPU Time (hours)	Disk Space (MB)
1.0	3 - 4	10 - 20
2.0	5 - 6	60 - 80
3.0	7 - 9	90 - 110
4.0	10 - 11	130 - 150
5.0	11 - 13	170 - 190

Case Study IRST: HW/SW Integration

Simulation of Hardware Virtual Prototype and Errors Found



- Board Fabrication Avg. = 5 days
- Board Assembly Avg. = 3 days
- HW Checkout Avg. = 3 days
- HW/SW Driver Integ. Avg. = 12 days

**Approx. 23 days
for Initial integration**

Summary

- VP Technologies' has extensive experience in designing, testing and verifying complex processor models ranging from microprocessors (i860, PowerPC) to DSPs (ADSP 21060 (Sharc)), and several interconnection fabric (fibre-channel, VME, SCI, Raceway), that are the basic building blocks of complex defense systems
- We look forward to working with you on creating accurate and cost-effective processor emulations.