APPLICATION NOTE
68030 DRAM Controller Design Using Verilog HDL

by Phil Rauba, Motorola Field Applications Engineer

Purpose

This article is intended to give a hardware engineer insight into the design methodology of using the Verilog Hardware Descriptive Language (HDL), targeting Motorola's field Programmable Array (MPA) and H4C gate array families. The advantage of using an HDL, such as Verilog, is the ability to retarget the design to other device technologies, by only resynthesizing the design description. A 68030 Dynamic Ram Controller design was used to demonstrate the portability of the Verilog language, and included all of the circuits necessary to interface DRAM to a 68030 microprocessor including: memory decoding, STERM generation, refresh request generation, CAS before RAS refresh, burst address sequencing, DRAM address multiplexing, and bus error time–out.

Design Methodology

The DRAM Logic was designed with a synchronous state machine design technique and described using the Verilog Hardware Descriptive Language, with the intent of providing a portable and easily maintainable design. The design tools used for this project are listed in Appendix A. The steps included in the design process include the system block diagram definition, state diagram generation, Verilog HDL logic definition, Verilog logic simulation, Verilog logic synthesis, place and route, and Verilog post simulation.

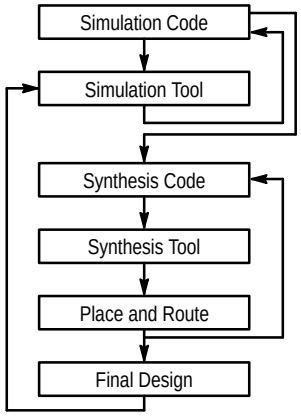


Figure 4–1. Verilog Design Method

The Verilog Design Methodology, Figure 4–1, illustrates the design flow beginning with the generation of the Verilog RTL source code. The DRAM design used a hierarchical module development methodology, which partitioned the design into eight submodules and instantiated each of the submodules into the design through a top module description designated as module glue68k. A stimulus module was also created that provided the test bench for verification of simulation code. The stimulus module included a 25MHz clock generator module, a behavioral 68030 bus controller module, and the instantiation of the top glue68k module, designated with the instance name of u1. As each Verilog submodule was written, the code was verified logically with the use of the stimulus module and the waveform capabilities of the Verilog simulator.

Once the design was logically verified, the original Verilog source code, that was used for simulation, was also used for synthesis. Each module of the hierarchical design was synthesized separately so that if a module needed to change, only that module would have to be resynthesized, saving considerable time by not having to resynthesize the entire design.

Since different tools were used for the logic synthesis, when targeting MPA and gate array, different methods were used during the synthesis process. For MPA development, the Exemplar tool was used for synthesizing each of the submodules individually. When synthesizing the top glue68k module, two passes were used with the Exemplar tools, one to generate submodule connectivity and the other to read and reformat the Verilog netlist. Empty submodules were instantiated in the design during the first pass of the Exemplar logic synthesizer with a Verilog netlist being generated. The Verilog glue68k netlist was then edited to add the links to the submodules by using the include command, referencing each of the submodule's file pathname. A final netlist was output from the second pass of the synthesizer, which read the eight Verilog netlists from the links in the top module, and reformatted the file to an EDIF netlist.

For gate array development, Synopsys was used for synthesis of the design. Each of the eight submodules and the top module, glue68k were read into Synopsys and synthesized all at one time without having the need to use the include command.

The EDIF netlist is used by the MPA and gate array place and route tools to generate the final design files. The MPA design procedure was to create a project, select the target device (MPA1036 181 pin PGA), input the EDIF netlist, place and route the design, and back–annotate into a structural Verilog netlist.

After placing and routing the DRAM MPA design, the structural Verilog netlist was used for post simulation. The structural Verilog netlist generated by the place and route back–annotation tool, contains precise MPA1036 gate and path delays to accurately predict the timing behavior of the final placed and routed design. Post simulation is useful for verifying and altering the design, if needed, before a printed circuit board is required. For post simulation, a modified version of the presimulation stimulus file was written to reflect the net name changes that were incurred by use of the design tools, but included the same clock and 68030 bus controller test suites as before. Final simulation of the DRAM design required the structural Verilog netlist module, the stimulus module, and the Verilog MPA1000 series gate primitive library, that was supplied with the MPA design system.

System Description

Bursting is a feature in the newer generation of CISC/RISC microprocessors that is comprised of a memory access of four long words of 32 bits each. The DRAM burst cycle is initiated by first generating a RAS cycle access and a CAS cycle access for the first long word, and then fetching the next three long words by generating only CAS cycles thereafter. The intention of the burst cycle is to divide the RAS cycle generation overhead of the first access amongst all four longword fetches; thereby, providing an overall access performance improvement as compared to single RAS generation for each longword.

The system block diagram is shown in Figure 4–2 and includes a 68030 microprocessor, a 16MByte dram array using 4Mx4 DRAMs, data bus drivers, and the MPA (or gate array). The MPA provides all of the DRAM interface circuitry needed to support 68030 bursting.

MPA Functional Description

The block diagram functional description of the MPA is shown in Figure 4–3 and shows all of the modules within the design, including the refresh timing generator, the refresh request state machine, the address decoder, the RAS/CAS state machine, the RAS/CAS decoder logic, the burst control state machine, the burst address generator, the DRAM address multiplexer, and the bus error time–out state machine.

MPA Timing Synchronization

As indicated in the MPA functional block diagram Figure 4–2, the main clock for all the state machines is the inverted 25MHz clock to the 68030. Since all of the output timing out of the 68030 is referenced to the falling edge of the processor's 25MHz clock, the clock is inverted and is used for clocking the MPA's internal registers. A delay line (not shown) will be needed for moving the assertion point of the address strobe signal with respect to the internal clock skew within the target device to prevent flip–flop metastable conditions. The delay line value will be dependent on the actual clock skew within the MPA or gate array.

Refresh Timing Generator

The refresh timing generator provides a 97.656KHz refresh request square wave with a period of 10.24usec for the refresh request state machine. The generator is comprised of a eight bit free running up counter with a 25MHz clock source.

Refresh Request State Machine

The refresh state machine receives the 97.656KHz refresh square wave and generates a ref_rq signal to the RAS/CAS state machine. The refresh state machine requests a refresh cycle only once when ref[7] is asserted high and inhibits the request after the RAS/CAS state machine has initiated a CAS before RAS refresh cycle.

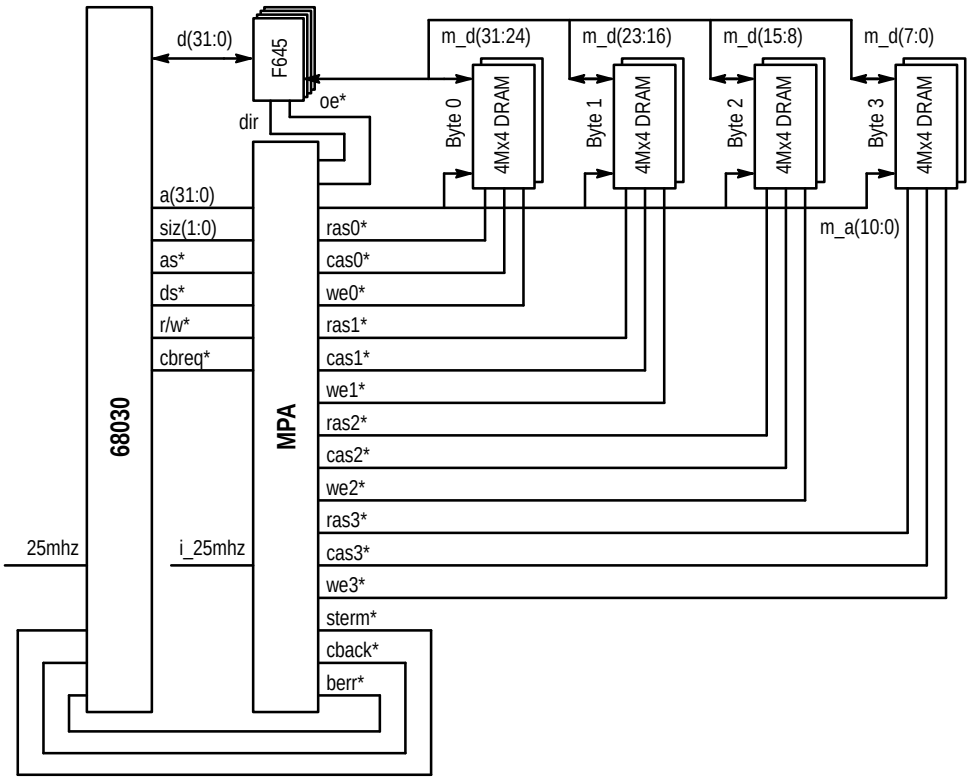


Figure 4–2. MPU–DRAM Controller Interfacing

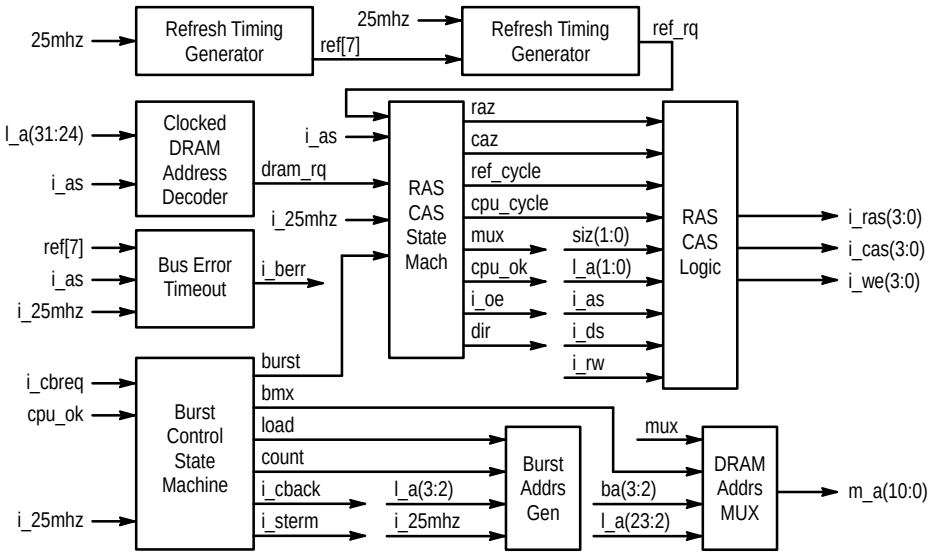


Figure 4–3. MPA DRAM Controller Detail

Figure 4–7 shows the refresh cycle that was initiated by the rising edge of ref[7] and by the assertion of ref_rq to the RAS/CAS Controller. The RAS/CAS Controller generates the timing for a CAS before RAS refresh cycle in synchronization with the refresh request state machine sequencing through the request operation. At the end of the refresh cycle the refresh request state machine is in the REFEND state waiting for negation of ref[7].

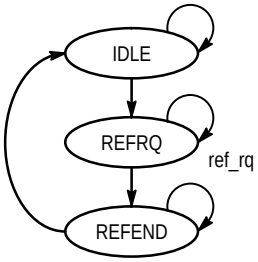


Figure 4–4. Refresh Request State Diagram

Synchronized DRAM Address Decoder

The DRAM Address decoder is used to decode the 68030 address 01xxxxxxh with qualification of address strobe to generate a access request to the RAS/CAS state machine. The dram_rq timing is shown in Figure 4–5.

RAS/CAS State Machine

The RAS/CAS State machine arbitrates between refresh requests and 68030 access requests via signals, ref_rq and dram_rq respectively. The arbitration between the two requests occurs in the IDLE state, where refresh has the highest priority. If a refresh request is pending the state machine will take the refresh branch and generate a CAS before RAS refresh timing sequence.

If a DRAM request is pending from the 68030 and a refresh request is not present, the state machine will take the 68030 access branch and generate the RAS, MUX, and CAS timing for a random access into the DRAM array. The assertion of RAS latches the row address into the DRAMs, the MUX signal will present the column address to the DRAM array, and the CAS signal will then latch the column address into the DRAMs. The signal cpu_ok will indicate to the burst controller to start wait state generation. If a burst request sequence is requested by the burst controller via the signal burst, the RAS/CAS state machine will sequence through the burst control states. For a full four long word burst, the burst address generator will provide the column addresses for each of the long word accesses. The RAS/CAS controller state machine will exit bursting upon the negation of address strobe, and has the capability of exiting a burst upon a premature ending of a full four long word burst.

RAS/CAS Logic

The RAS/CAS logic is comprised of combinational logic that encodes the CAS signals for selecting which byte lanes of the DRAM array that are going to be accessed during a cycle. For a CPU write access, the logic supports the misalignment capabilities of the 68030, providing CAS signals only to the bytes of the DRAM array that will be accessed for the write operation. For a CPU read cycle all of the CAS signals will be asserted. During refresh cycles all of the CAS and RAS lines will be asserted.

Burst Control State Machine

The burst control state machine provides all of the bursting control for a 68030 DRAM access and is synchronized to the RAS/CAS controller. Upon the receipt of a dram_rq, the RAS/CAS controller will generate RAS and CAS timing to the DRAM array and will assert the signal cpu_ok, indicating to the burst controller state machine to start a burst cycle. The burst controller will leave an idle state and assert the i_cback signal indicating a synchronous burst access. The burst controller will then insert wait states during the burst operation and be responsible for asserting i_sterm indicating the availability of

DRAM data. The burst controller will also generate the counting and load controls for the burst address generator and provide the burst multiplexing control to the DRAM address multiplexer.

Burst Address Generator

The burst address generator provides the two least significant bits of the DRAM address during a 68030 burst cycle. The burst controller will initiate the loading of the first burst address into the burst address generator and then control the incrementing of the addresses for the next three long word accesses of the burst cycle.

The burst address generator sequences through the long word addresses, which are generated from the ba(3:2) signals. Entry into the counter state machine can occur at any state and will be defined as the starting 68030 starting address plus one. During the first long word access of the burst, the first address will be supplied by the 68030; the next three long word addresses will be supplied by the burst address generator. After the first 68030 address, the address generator will enter the state of the next address from the load signal from the burst controller. The burst address generator will then be incremented two more times for the next two long word addresses.

DRAM Address MUX

The DRAM address MUX provides the row and column addresses on a eleven bit multiplexed address bus to the DRAM array. The 68030 provides the row address to the DRAM array, and the column address of the first long word access of a burst cycle. The two least significant bits of the column address will be supplied by the burst address generator for the next three long word accesses in the burst cycle. Gating of the addresses onto the DRAM multiplexed address bus is controlled by both the burst controller and the RAS/CAS controller. The DRAM address MUX generates a 68030 burst access to long word address locations 01xxxxx0h to 01xxxxxCh, when the starting 68030 address is 01xxxxx0h.

Bus Error Time–out

A bus error watch dog time–out function is provided by the DRAM controller to keep the 68030 from locking up due to accesses into unused memory. The bus error time–out controller monitors the assertion of address strobe and will generate a bus error to the 68030 if it has kept address strobe asserted from 40.96 usec to 46.08 usec. This time–out may vary depending on where the 68030 started a memory access in relationship to ref[7] of the refresh timing generator. The bus error time–out controller monitors ref[7] for its state transitions, while watching the assertion of address strobe. If address strobe is negated before reaching the state NOACK of the bus error time–out state machine, a bus error will not generated.

System Timing

The system timing is shown in Figure 4–5 and gives the overall operation of the modules within the DRAM design for a DRAM array access. The diagram shows the logical implementation of the design with zero propagation delay and is meant to give a relationship of the signal handshaking between the submodules of the design. The system timing diagram shows a four long word burst read access to the DRAM array and is comprised of a 14–7–7–7 burst for a total of 35 cycles. The design has not been optimized for speed at this time, with the intent of generating a reasonable amount of logic for timing verification targeting lower cost designs using slower DRAMs.

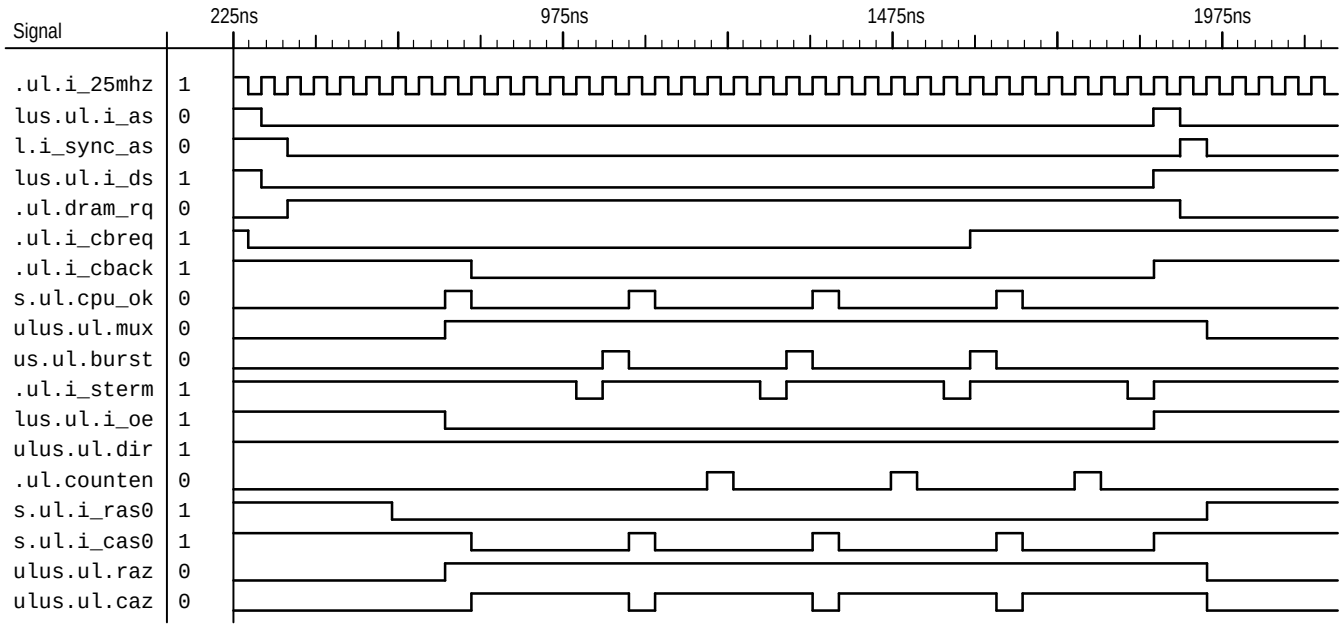


Figure 4–5. DRAM Burst Timing

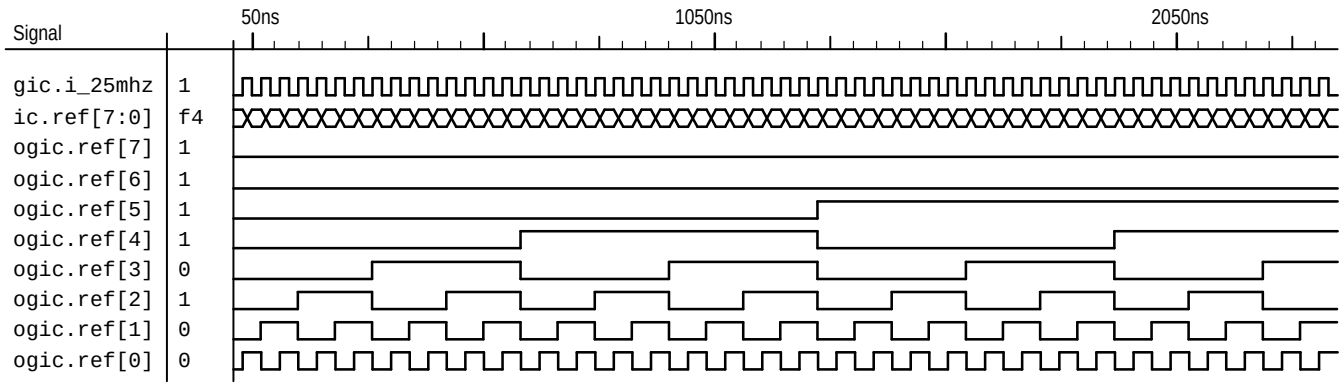


Figure 4–6. Refresh Counter Timing

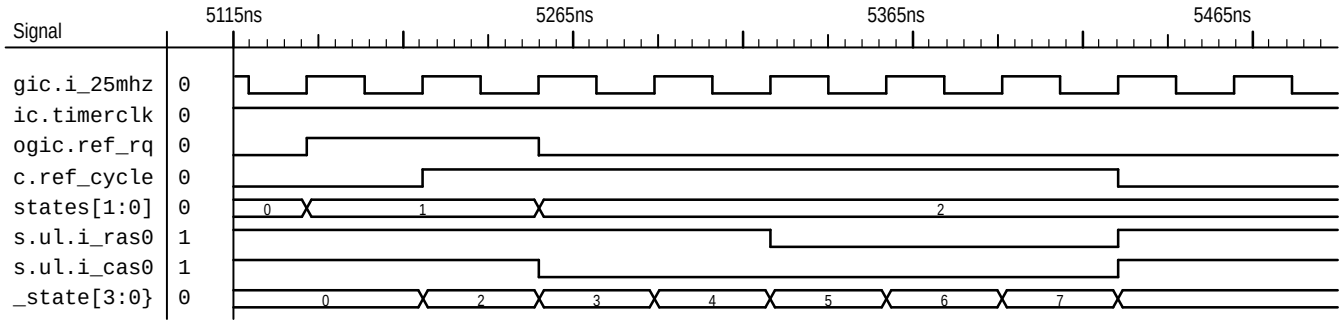


Figure 4–7. Refresh Timing

Figure 4–5, Figure 4–6 and Figure 4–7 timing diagrams are logical simulations of the design copied from the Frontline waveform timing analyzer and do not include final place and route timings.

Verilog Coding Example

The following Verilog synthesis code describes the refresh module:

```
module refresh (i_25mhz, ref_cycle,
                timerclk, ref_rq);

input  i_25mhz;
output timerclk;
wire [7:0] temp;
reg   [7:0] ref;

assign timerclk=ref[7];

assign temp[0] = ~ref[0];
assign temp[1] = ref[1] ^ ref[0];
assign temp[2] = ref[2] ^ (&ref[1:0]);
assign temp[3] = ref[3] ^ (&ref[2:0]);
assign temp[4] = ref[4] ^ (&ref[3:0]);
assign temp[5] = ref[5] ^ (&ref[4:0]);
assign temp[6] = ref[6] ^ (&ref[5:0]);
assign temp[7] = ref[7] ^ (&ref[6:0]);

always @ (posedge i_25mhz)
begin
ref[0] = temp[0];
ref[1] = temp[1];
ref[2] = temp[2];
ref[3] = temp[3];
ref[4] = temp[4];
ref[5] = temp[5];
ref[6] = temp[6];
ref[7] = temp[7];
end

// Refresh request state machine

input ref_cycle;
reg  [1:0] ref_states;
output ref_rq;
reg  ref_rq;

parameter IDLE  = 'b00; // idle state
parameter REFRQ  = 'b01; // assert ref rqst
parameter REFEND = 'b10; // wait for end

always @ (posedge i_25mhz)
begin
      case (ref_states)
      IDLE:begin
            if (ref[7])
                  begin
                  ref_states = REFRQ;
                  ref_rq = 1;
                  end
            if (~ref[7])
```

```
                begin
                ref_states = IDLE;
                ref_rq = 0;
                end
        end
    REFRQ:begin
        if (ref_cycle)
                begin
                ref_states = REFEND;
                ref_rq = 0;
                end
        if (~ref_cycle)
                begin
                ref_states = REFRQ;
                ref_rq = 1;
                end
        end
    REFEND:begin
        if (~ref[7])
                begin
                ref_states = IDLE;
                ref_rq = 0;
                end
        if (ref[7])
                begin
                ref_states = REFEND;
                ref_rq = 0;
                end
        end
    endcase
end
endmodule
```

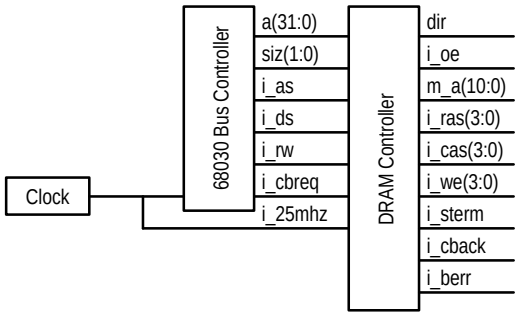


Figure 4–8. Simulation Model

MPA – Verilog Simulation

Figure 4–8 shows a block diagram of the Verilog stimulus module used for logic simulation and includes the 25MHz clock module for generating a system clock, the 68030 bus controller module for generating the timing for a 68030 burst cycle, and the glue68k DRAM Controller module. Figure 4–5 to Figure 4–7 show the results of the simulation, which was run from 0ns to 10,000ns. The stimulus module included the vectors for generating the 68030 bus signal timing for a burst read cycle. Once the simulation code was verified logically by the simulator and waveform analyzer, the code was determined to be free from syntax errors and matched the expected timing for the design. Note that at this point, the design has not been verified with the gate and path delays generated from the final place and route tool.

MPA – Exemplar Synthesis

Prior to synthesis, the MPA1000 libraries, that are supplied with the MPA design system place and route software must be properly installed into the directory pathname C:/exemplar/lib and include the filenames p_mpa20.syn and p_mpa23.syn. Exemplar will reference these libraries for gate type selection when targeting a MPA1000 series part.

Although Exemplar has a graphical user interface, this designer preferred to use DOS synthesis commands included in (.bat) batch files. As the design was synthesized, the file manager was used to navigate through the design's subdirectories and to synthesize by double clicking on the batch file contained in a submodule's directory.

The refresh submodule example is synthesized with the DOS command "fpga refresh.v refresh.vg –target=p_mpa –save –macro". The save option stores all the optimization passes of the logic synthesizer allowing the user to select the best pass based on timing and cell count size. The macro option is used for inhibiting the assignment of I/O pads to the inputs and outputs of the submodules, reserving I/O pad assignments to the top module. Each submodule of the design is synthesized separately using the same command, but with different filenames that are identical to the submodule name. The top module glue68k is synthesized with "fpga glue68k.v glue68k.vg –target=p_mpa –pass=2" and uses a control file within its directory called glue68k.ctr, which includes the command:

```
BUFFER_SIG IPCLK i_25mhz
```

to assign the signal i_25mhz to a clock tree within the MPA1036. The glue68k.vg Verilog netlist is edited manually, adding include commands to the end of the file to read in all the Verilog netlists into the top hierarchical module during the Exemplar reformat pass. The include command:

```
‘include “c:\fpga\refresh.vg”
```

reads in the refresh submodule into the glue68k.vg module when the DOS command "fpga glue68k.vg glue68k.edif –source=p_mpa –target=p_mpa –effort=reformat", is executed in the fpga directory with the glue68k.ctr control file removed. The final design resulted in 440 gates including 44 DFs, 40 inputs. and 25 outputs , when targeted to a Motorola Programmable Array.

The types of gates that were synthesized by the Exemplar tool for the DRAM Controller design included:

AN2	INV	ONE
BUFF	IPBUF	IPCLK
OPBUF	DF	ND2
OR2	DFR	NR2
XN2	XR2	

The synthesis times for the modules varied with the complexity of the logic, but were relatively fast. For instance the dram module, which is a fairly complex state machine design, took seconds to synthesize as shown:

Pass	Cells	Delay (ns)	Min:Sec
1	153	28.8	00:13
2	93	8.4	00:10
3	155	28.8	00:18
4	88	18.0	00:09
5	153	27.6	00:14
6	88	18.0	00:08
7	155	28.8	00:15
8	88	18.0	00:08
9	187	26.4	00:21
10	98	9.6	00:35
11	91	16.8	00:33

The passes of the Exemplar synthesizer are related to eleven different types of optimization algorithms. The best pass for the dram module, based upon timing, is pass 2, with an estimated gate delay of 8.4ns. In general all of the modules synthesized in this design, had pass two consistently generate the least amount of level delays. One may save some CPU time by specifying that a particular pass be executed by the Exemplar tool.

MPA Design System Place and Route

The design was processed by five steps using the MPA design system graphical user interface: Set Tool Options, Import, Autolayout, Generate Configuration, and Generate Back–Annotation. The input to the place and route tool is an EDIF netlist and requires the Exemplar EDIF netlist file glue68k.edi to be renamed to glue68k.edn to be imported.

MPA design system options that are required to be selected prior to place and route include: part number, package type, and mode. For this design the part was a MPA1036HI, which is a Motorola 181 pin, 8000 MPA equivalent gates, 3600 cell array. The mode determines how the device will be configured upon power up and reserves programming pins on the device to prevent the place and route tools from assigning them to user I/O. The mode selected for the design was “Boot From ROM”, where the MPA loads its program from a serial ROM. The Autolayout place and route option was set to use default settings and provided adequate delay timings at 25MHz. Optional parameter settings for high utilization and for minimum delay, which are intended for compact and high speed designs respectively.

The Autolayout tools at default settings, generated a design that used 376 cells, utilizing 20.9% of the device, with an estimated maximum frequency of 30.7MHz. With the minimum delay option selected, the design routed with an estimated maximum frequency of 39.4MHz. The user may experiment with different option settings to generate faster designs, but for this application the 30.5MHz output was adequate and was used for the final design.

Pin assignments for the design can be viewed in the pin report file glue68k.prp with a small section of the report shown here:

```
I/O Pin report file
Definition: glue68k
Layout: glue68k

Format: Port Name, Net Name, Device Coord, Internal
Pad No, Package Pin Name

Port      i_25mhz, net      i_25mhz @ ( 36, 0)
IO pad    15 pin n8
Port      i_sterm, net      i_sterm @ ( 20, 0)
IO pad    8 pin p4
Port      siz1, net         siz1 @ ( 78, 40)
IO pad    166 pin h13
```

The Back–Annotation tool is used to generate a structural Verilog netlist for post simulation and assigns the file extension of .vba, which was renamed to .v for input into the simulator. The designer can verify the final design with post simulation or by viewing the timing report generated by the Autolayout tool.

H4C Gate Array

The Verilog netlist files for the design were transferred on a DOS disk to a Sun workstation. Synopsys was used to read in the top hierarchical glue68k module and each of the submodules of the design. The design was synthesized and targeted to the H4C gate array family without any errors. The combinational area of the design was 446 and the noncombinational area was 344 (43 flip–flops using 8 gates per flip–flops) with a total used area of 790.

The types of gates generated by Synopsys include:

AND2	INV2	NOR8H
AND2H	INVB	OA211H
AND3	MUX2A	OA21H
AO22H	MUX2I	OA22H
AOI22H	MUX2IH	OAI211H
DFFP	NAN2	OAI22H
DFFRP	NAN3	ONDAI22H
EXNORA	NAN4	OR2
EXORA	NOR2B	OR3
INV	NOR2H	OR4

One observation of the synthesized design was that the Synopsys synthesizer added buffers to heavily loaded signals to minimize the wire delays and edge rates in the design. Another observation indicated that the gates that were generated for the MPA and H4C gate array were quite similar because of the fine grained nature of the MPA.

Final Simulation

The final placed and routed MPA design was post simulated with the back–annotated structural Verilog netlist using the Verilog simulator. Prior to post simulation, the MPA design system Verilog library file, located in the path C:\dpld\verilog\library.v was edited to enable Verilog XL compliance with the command ’define XL_comp. Another modification of the library needed to eliminate errors encountered during post simulation was that the library description of the module ONE was changed from:

```
'ifdef XLcomp
        pullup (strong1,strong0) (PU);
'endif
to:
ifdef XLcomp
        pullup (strong1) (PU);
'endif
```

The structural Verilog netlist file glue68k.v was also edited to declare the netlist as type back–annotated by enabling the line: 'define source_back_annotation

The post stimulation file stimulus.v is similar to the simulation file, but was edited to change input and output names from lower case to upper case, caused by the renaming of signals from the MPA design system back–annotation tool. The following is a subset of Verilog simulation module stimulus.v, that was used for the test bench and includes a 25MHz free running clock:

```
/
'timescale 1ns / 1ps
module stimulus ;
wire I_25MHZ;
...
wire DIR;
        clk25   clockGen (I_25MHZ);
        GLUE68K u1(I_25MHZ, I_AS, D_I_AS,
                I_DS, I_RW, I_CBREQ, SIZ1,
                SIZ0, L_A31, L_A30, L_A29,
                L_A28, L_A27, L_A26, L_A25,
                L_A24, L_A23, L_A22, L_A21,
                L_A20, L_A19, L_A18, L_A17,
                L_A16, L_A15, L_A14, L_A13,
                L_A12, L_A11, L_A10, L_A9,
                L_A8, L_A7, L_A6, L_A5,
                L_A4, L_A3, L_A2, L_A1, L_A0,
                I_CBACK, I_STERM, BCYCLE,
                M_A10, M_A9, M_A8, M_A7,
                M_A6, M_A5, M_A4, M_A3,
                M_A2, M_A1, M_A0, I_CAS0,
                I_CAS1, I_CAS2, I_CAS3, I_RAS0,
                I_RAS1, I_RAS2, I_RAS3,
                I_BERR, I_OE, DIR);
initial
begin
        u1.B_A3 = 0;
        u1.B_A2 = 0;
...
        u1.CONTROL_VL8 = 0;
end

// simulate a 68030 DRAM burst cycle
initial
begin
#5      u1.L_A31 = 0; u1.L_A30 = 0;
         u1.L_A29 = 0; u1.L_A28 = 0;
...
end
endmodule
module clk25 (clock);
output clock;
reg  clock;
initial
        #5 clock = 0;
always
        #20 clock = ~clock;
endmodule
```

Frontline's graphical user interface was invoked and a project called glue68k.dgn was created. Setup of the simulator included setting directory pathnames to the locations of the Verilog source files stimulus.v and glue68k.v and of the MPA design system Verilog library file library.v. The simulator was setup for maximum delay type and to use the +heirinstport command line option to allow the use of hierarchical pathnames used in the glue68k hierarchical design. The simulation was run and the timing of the design was verified with the waveform analyzer as illustrated in Figure 4–9 and Figure 4–10. Note that timing waveforms show accurate gate and path delays within the MPA1036.

Appendix A – Design Tools

The design tools were selected for a 486 PC Platform and included Frontline Design Automation, Inc's PureSpeed Verilog Simulator, Exemplar's CORE–TD–DOS PC Topdown Verilog Synthesizer, and MPA design system. The PC was upgraded to 24MBytes of DRAM memory, of which 16MBytes were the minimum required to run the Exemplar software. A CD–ROM drive was used for loading the MPA design system software. Waveforms included in this application note were captured from Frontline's waveform analysis tool for both logical and post simulation figures.

For targeting H4C gate arrays, the design development tool kit was Motorola's Open Architecture CAD System (OACS). Synopsys was used for Verilog logic synthesis on a Sun platform in one of Motorola's ASIC design centers.

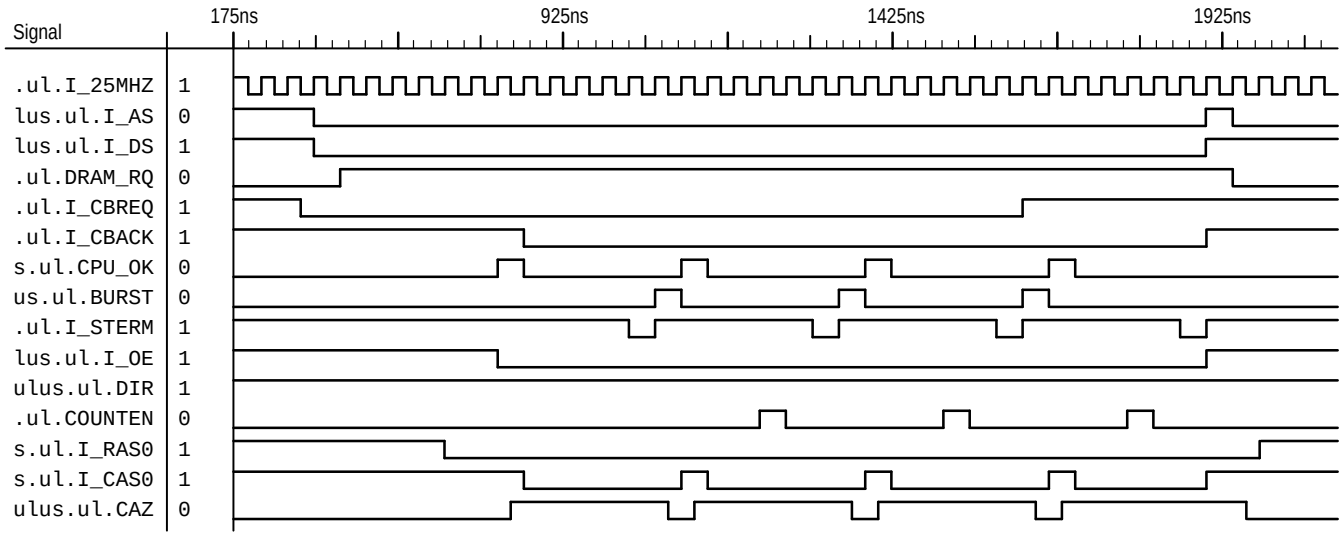


Figure 4–9. DRAM Burst Timing Final Simulation

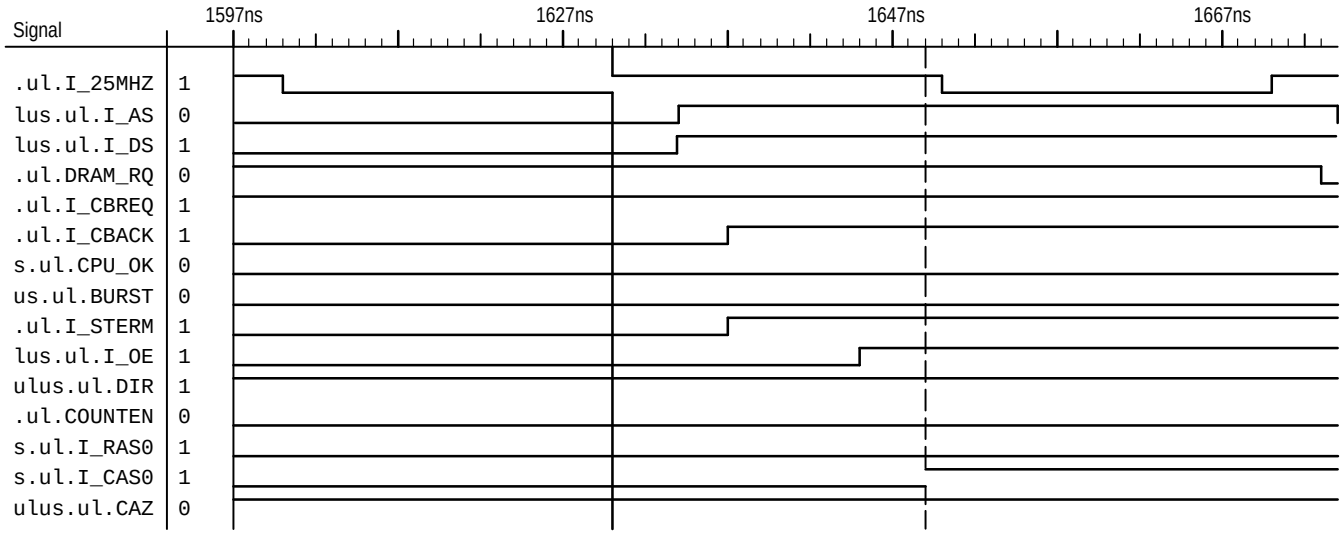


Figure 4–10. DRAM Burst Timing Final Simulation – Delay Example

APPLICATION NOTE
Programming Multiple MPA1000 Devices Using Serial Peripheral Interface (SPI)

by Ajay Matani, Motorola Field Applications Engineer

Introduction

Serial Peripheral Interface (SPI) is an efficient on–board Serial Data Transfer mechanism supported by most of Motorola Microcontrollers. MPA1000 series arrays offer various modes of loading “ConfigWARE” (Configuration data that defines MPA logic functionality and interconnect) data into the device. This application note details a microcontroller MPA configuration control interface using an SPI port.

Why Use SPI for “ConfigWARE” Download?

In–system programmability is not a new concept, as most SRAM based MPA’s provide a mechanism for the Microprocessor to configure functionality. For embedded systems, Hardware and Firmware constitute a typical system. Sophisticated embedded systems like Laser Printers provide support for downloading “SoftWARE” (as Fonts, Printer Emulation etc.) for example; while FLASH EEPROMS allow for “FirmWARE” upgrade as is a case in many new PC–motherboards that have BIOS in FLASH. As shown in Figure 4–11, flexibility offered by these different layers decreases as we approach the HardWARE layer, which is quite fixed.

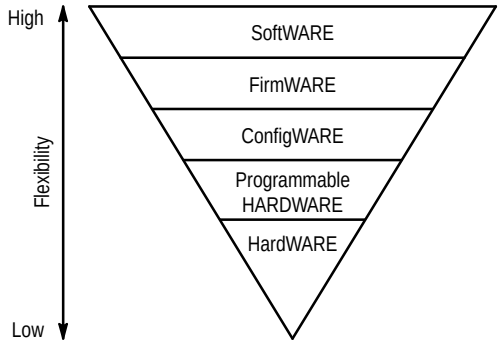


Figure 4–11. Programming Flexibility

“ConfigWARE” provides the flexibility to use the same “HardWARE” to carry out different functionality. The time and resources required to download “ConfigWARE” into the MPAs becomes critical as device size and number of devices in a system increase. It is also beneficial to store “ConfigWARE” along with “FirmWARE” in non–volatile memory like FLASH, Floppy, HDD or download over a Network connection.

Many of Motorola’s highly integrated MCU devices have on–chip resources (such as RAM, PROM, serial ports etc.) that enable independent boot–up and loading of the “ConfigWARE”. Considering that most of them also have SPI support, it is worthwhile to examine efficient use of SPI for downloading “ConfigWARE”.

MPA1000 Configuration Methods

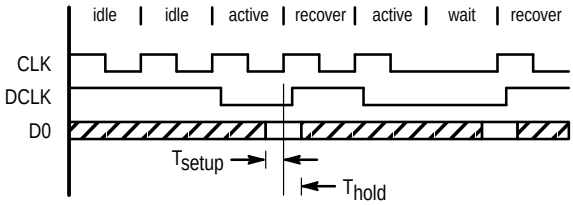
Four basic methods are available to download “ConfigWARE” into the MPA devices; one Micro Mode with typical peripheral bus interface and three BFR (Boot From ROM) modes. The Micro Mode, BFR Mode(1) and BFR Mode(3) support byte wide data transfer hence BFR Mode(2) which supports serial data transfer is the only one we consider for SPI interface. In fact, MPA configuration logic supports 8–bits at a time and thus accumulates the serial stream into a byte before loading it in the internal RAM array. This arrangement matches well with the SPI support of byte data transfer at a time on the serial protocol.

BFR Mode(2) operation is a very simple serial transfer mechanism that uses 3 signals, CLK(clock in), DCLK(clock out) and D0(data). This mode is intended to load from external serial PROM devices like MPA17128 (page 2–52). The signal relationships are as follows:

CLK (up to 20Mhz) is the master clock used by configuration logic. This could also be generated from the MPA internal Ring Oscillator.

DCLK is output from the MPA and can run as fast as 1/2 the CLK frequency.

A simple way of looking at DCLK is to consider it as a Data Strobe and clock for an Address Counter, where DCLK low to high transition is the critical edge for both operations (Figure 4–12). As each transition of DCLK is generated by a rising edge of the CLK signal, manipulating CLK allows controlling DCLK operation. D0 is data presented to the MPA.



NOTE: During “idle” phase of CLK, internal Reset or Configuration Sequence logic is being exercised. At the end of “active” phase of CLK, new configuration data is read in. The “wait” phase of CLK is created by stretching the CLK low phase of the active cycle. After every “active” or “wait” condition, a “recover” cycle of CLK is needed.

Figure 4–12. ConfigWARE Download Timing

By extending DCLK in its low state, wait states can be inserted in the access of serial data on D0. This can be easily achieved by keeping CLK in low state whenever needed. As CLK is used by the configuration logic also, the suggested clock stretching should be applied only during data access cycles denoted by DCLK low state. Since the configuration

logic is a static design, there is no minimum operational frequency requirement allowing large number of wait states if needed.

The window for which D0 should be stable with valid data is defined by Figure 2–28, Figure 2–29 and the accompanying table on page 2–26. This relatively narrow window requirement is easily achievable.

Serial Peripheral Interface

SPI operation as well, is quite straightforward. The SPI on a MCU can be configured as either a Master or a Slave. The serial transfer operation is carried out on four lines, SCK (Serial Clock), MOSI (Master Out Slave In), MISO (Master In Slave Out) and SS/(Slave Select) supporting synchronous bi–directional serial transfer of byte size data.

The primary difference between the Master and Slave Mode is the source of SCK. Though transfers are synchronous, SPI circuit is required to be a static design which allows the SCK to have no maximum phase or period time requirement. In Master Mode, the MCU based SPI circuit sends a burst of 8–bits, synchronized to a prescaled internal CPU clock.

The programmer's model of SPI consists of SPDR (8–bit SPI data register), SPCR (SPI control register) and SPSR (SPI status register). Refer to MC68HC11RM/AD for detailed discussion of SPI operation.

Once configured as a Master, writing to SPDR starts a transfer of the data byte uninhibited, on the MOSI signal. The data presented on MISO by the slave is de–serialized and made available in the SPDR at the end of byte transfer. As writing to the SPDR also starts a transfer sequence, there is no facility to carry out hand shake with the slave apriori to the transfer.

When SPI on a MCU device is configured as a slave, the SCK supplied by the external master controls the flow of transfer. MOSI now becomes input and MISO becomes output. The SS/ signal plays an important role in this mode, acting as a gate to the SCK. This facilitates selective transfer to multiple slave using common SCK signal.

DESIGN APPROACH

This application note is based on a design implemented in a working system with multiple MPA1036 devices daisy chained. The requirement of this system is to provide a flexible and efficient "ConfigWARE" download capability. The design uses MC68HC11K4 MCU with external FLASH EEPROM to store the "ConfigWARE".

Various possibilities were considered to establish the lower level handshake with MPA configuration logic. BFR Mode(2) was chosen as it sacrifices only **one** general purpose i/o signal for configuration, namely DCLK.

The easiest, gluless and trivial method of interfacing an MCU to an MPA device in BFR[2] is a single, 8–bit I/O port and 100% software controlled transfer. MCU software overhead results in a long MPA subsystem start–up time if this method is used. As the size of the MPA subsystem increases, this problem is compounded. Using the MCU SPI hardware in Slave Mode and a minimal amount of external logic, relatively fast configuration times xan be acheived using a serial data stream.

FUNCTIONAL DESCRIPTION

We have established the basic serial transfers in terms of MPA BFR Mode(2) and SPI in our discussion up to this point, but there are a few more functionalites that need to be examined at system level. Consider that there are *n* MPA devices daisy chained as shown in Figure 3. The MCU (MC68HC11 in this discussion) and PAL device constitute the controller.

The controller requires only three outputs (CLK, D0 and RESET1/) to carry out the task. The POWRUPn signal may be monitored to confirm the end of download sequence, though any error condition may be detected alternatively by making sure that exact number of bytes are downloaded.

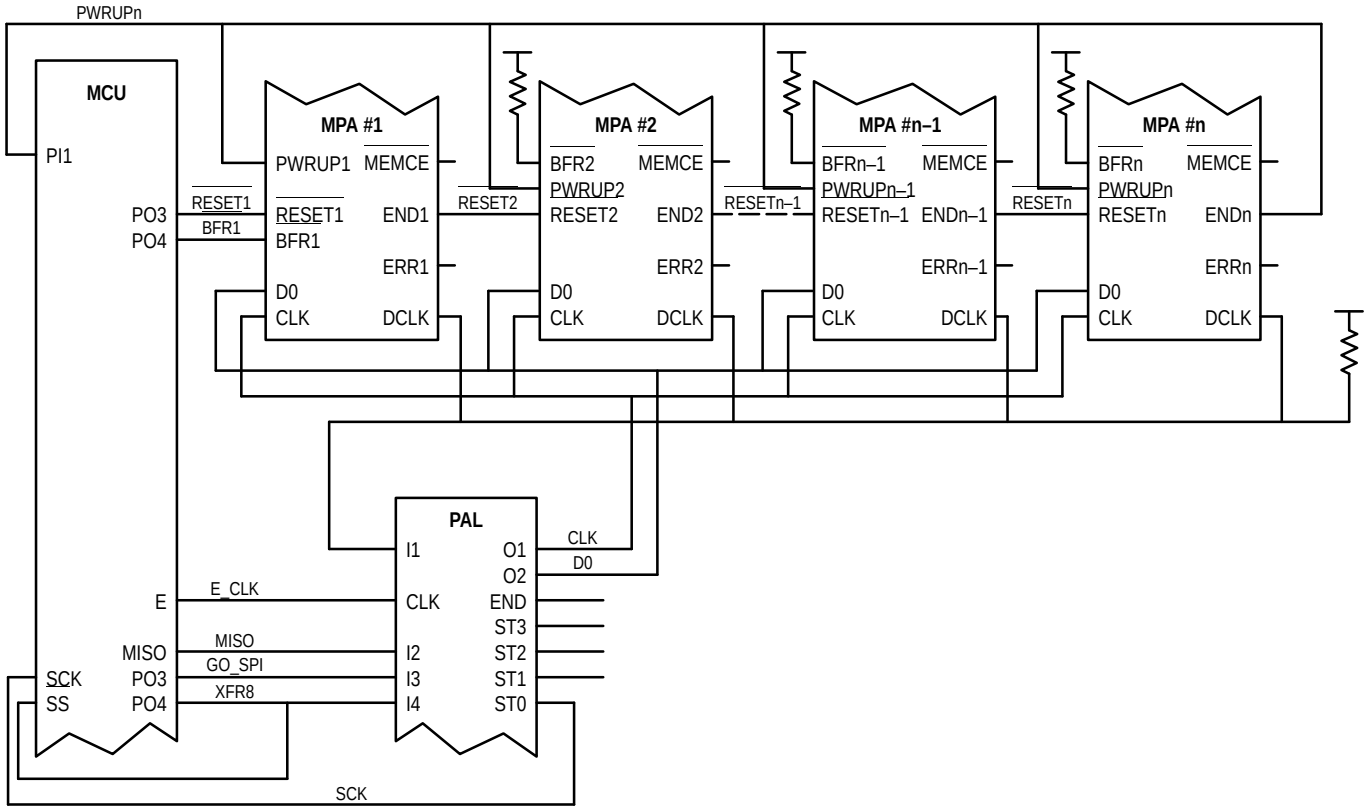


Figure 4–13. Multiple Daisy Chained MPA Devices

Let us look at all the signals in detail.

Signals outside of the controller :

CLK Derived from PAL External clock input. Used during reset and configuration sequence. Clock stretching is used in this application during the Configuration sequence to establish proper handshake between the MPA and program controlled sequence on the MCU.

$\overline{\text{RESET1}}$ Output from MCUReset to the First MPA in the daisy chain. The falling edge initiates reset sequence. At the end of reset sequence, if the signal is still asserted, configuration sequence is delayed till the rising edge of signal is recognized. Otherwise, the configuration sequence immediately follows the reset sequence. CLK must be active during these sequences.

$\overline{\text{RESET}}$*n* Connect ENDn–1 to RESETn as configuration logic keeps END low till the present device completes the configuration sequence and lets the next MPA device start its own reset and configuration sequence.

$\overline{\text{BFR1}}$ Output from MCU. Active low signal initiates Reset and Configuration sequences. Acts same as RESET1 except that the Configuration sequence immediately follows the reset sequence regardless of the state of $\overline{\text{BRF1}}$ signal. (The use of this signal is optional).

$\overline{\text{BFR}}$*n* These are pulled up.

PWRUP*n* Last MPA END connects to all PWRUP signals. When this signal on the MPA is low, all the user I/O are disabled (in tri–state). A desirable condition until all the devices are configured properly.

$\overline{\text{MEMCE}}$ Not used.

$\overline{\text{ERR}}$n These output signal gets asserted during configuration sequence if Device ID mismatch or Checksum error is detected. They can be left open as such error condition stops configuration sequence and can be detected alternatively.

DCLK Input to PAL. Is the wired–or signal (requires external pull–up) from all the MPA devices that gets pulled low by the currently configuring device during data transfer.

D0 Output from PAL. This is the data input to all the MPA devices.

Signals inside the controller (between MCU and PAL) :

E_CLK Output from MCU. System clock.

MISO Output from MCU. SPI data out of MCU in Slave Mode.

GO_SPI Output from MCU. Sequence Master Control output, resets and enables configuration sequence logic under program control.

XFR8 Output from MCU. Byte Transfer Control output, initiates data transfer sequence logic under program control. Also acts as Slave Select for SPI logic with connection external to the MCU.

SCK Output from PAL. SPI clock input from MCU. Controls bit data transfer out of SPI shift register.

CONTROL LOGIC IMPLEMENTATION

The MC68HC11K4 and a PAL22V10 device is used for the control logic.

The base clock for the circuit is the “E” clock from MCU that is also used by the SPI logic internal to the MCU. Typical frequencies for 8–bit MCUs for the system clock “E” are 4 Mhz and as high as 25 Mhz for some 32–bit MCUs.

The circuit uses four inputs, three outputs and four state bits on the PAL. One of the outputs, SCK is considered as a state variable too. Refer to Appendix A, where the PAL design in CUPL (need to check trade mark) source language is described. Appendix B lists the Boolean equations in AND–OR form for each of output and state variable as generated by CUPL assembler.

Two general purpose Output Port bits (GO_SPI and XFR8) from the MCU are required besides the SPI signals. Considering that data is flowing from MCU to the MPAs, only data out signal (MISO in slave mode) of the SPI logic is used along with the SCK and SS signals.

Appendix C lists the assembly source code for MC68HC11K4 MCU for the subroutines needed to carry out the ConfigWARE download.

CONTROL FLOW

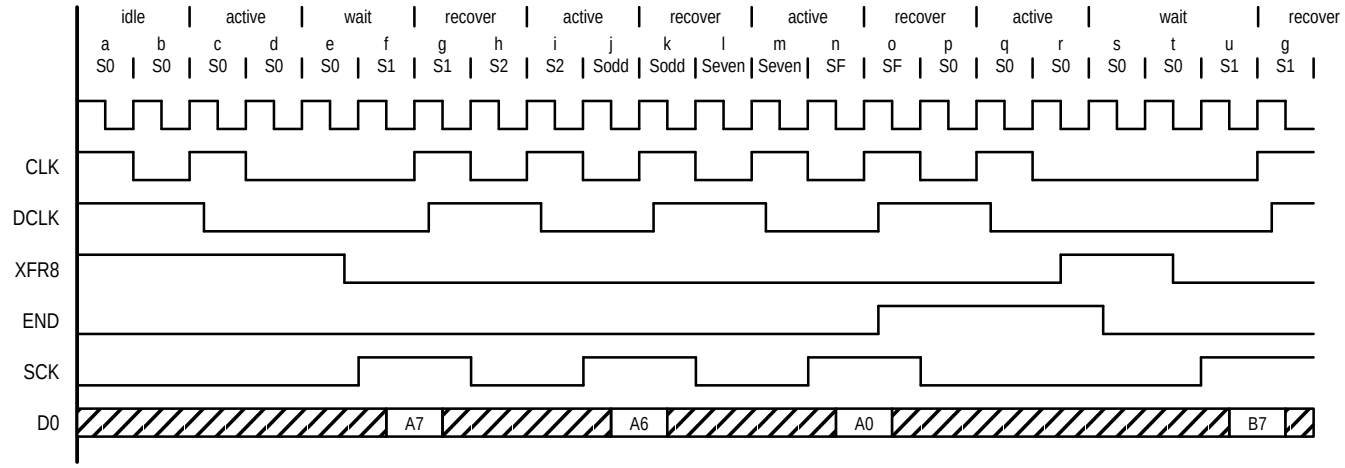
For ease of understanding, let us follow the firmware in Appendix C to track the operation of control sequence.

The calling function to “LD_FPGA” is assumed to have set the general purpose port bits to the correct direction and have made a call to “RST_FPGA” which forces RESET1/. The first thing LD_FPGA does is to call “EN_FPGA” which makes the control logic ready by negating XFR8, correctly set up the SPI on MCU, assert GO_SPI and negate RESET1/.

This forces the sequencer in the PAL to STATE S0 and stay there till sequence begins when XFR8 is asserted (active low). The CLK output of PAL keeps on toggling while in STATE S0, defining the the “idle” phase of of CLK (Fig. 4). Negated XFR8 also negates END state variable.

Next, LD_FPGA makes sure that the correct ConfigWARE for the first device is made available (starting at address $4000) before it calls the “IM2FPGA” (Image to FPGA) function. IM2FPGA calls “BY2FPGA” for the exact number of times to download the complete image for a single FPGA device. For FPGA1036 for example, the number is 14600 ($3908) bytes as explained in Appendix D.

BY2FPGA is the lowest level routine that directly controls the transfer of byte to the current FPGA device. The data byte is written to the SPI data register and XFR8 is made active to begin the transfer. While the sequencer in PAL and MPA synchronize and carry out the transfer, the program waits for SPI transfer to complete within a certain period of time. If the code times out, it returns with error condition set. As only the completion of SPI transfer is waited on, there is no need for any external signals to indicate error conditions.



NOTE:

```
a ---> b : S0>S0,     as DCLK is high
b ---> c : S0>S0,     as DCLK is high
c ---> d : S0>S0,     as XFR8 is high
d ---> e : S0>S0,     as XFR8 is high
e ---> f : S0>S1,     as DCLK and XFR8 low
f ---> g : S1>S1,     as DCLK is low
g ---> h : S1>S2,     as DCLK is high
h ---> i : S2>S2,     as DCLK is high
i ---> j : S2>Sodd,   as DCLK is low
j ---> k : Sodd>Sodd, as DCLK is low
k ---> l : Sodd>Seven, as DCLK is high
l ---> m : Sodd>Seven, as XFR8 is high
m ---> n : Seven>SF,   as DCLK is low
n ---> o : SF>SF,      as DCLK is low
o ---> p : SF>S0,      as DCLK is high (END set)
p ---> q : S0>S0,      as DCLK is high
q ---> r : S0>S0,      as DCLK is low but END is high
r ---> s : S0>S0,      as DCLK is low but END is high
s ---> t : S0>S0,      as DCLK is low, END is low but XFR8 is high
t ---> u : S0>S1,      as DCLK is low, END is low and XFR8 is low
u ---> g  catch the sequence at g again
```

Figure 4–14. System Training

For the normal transfer, the sequencer in the PAL waits in STATE S0 when MPA device is not ready for transfer as indicated by DCLK high. When FPGA is ready to receive data, it asserts DCLK low. State m/c responds by stopping the transitions on CLK signal, forcing wait condition till MCU holds XFR8 signal high. When MCU program sequence catches up and asserts XFR8 low, STATE S1 is entered.

All odd numbered states are similar and correspond to the time when DCLK is high. Except for STATE S0, all even number states correspond to the time when DCLK is low. For a byte transfer, XFR8 going low should run the sequencer from STATE S0 to STATE SF in a sequence, and back to STATE S0. When the state m/c is in STATE SF, END state variable gets set on the next clock, indicating completion of a transfer. Until XFR8 is negated by BY2MPA when it detects completion of SPI transfer, the END condition forces the state m/c to stay in STATE S0, even when the MPA is ready to receive the next bit, indicated by DCLK low. This mechanism ensures that one and only one byte is transferred on every MCU controlled cycle of XFR8.

The new transfer will not start till program sequence makes a new call to BY2MPA, which in turn will begin with XFR8 active. Timeout or SPI error condition, if any, makes BY2FPGA return a non zero value indicating error. The address value of the image byte of the erroneous transfer is saved in SPI_ERR variable. IM2FPGA passes the error back to LD_FPGA.

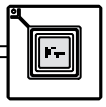
If there is no error, LD_FPGA repeats the above process for additional devices, making sure that the correct ConfigWARE for that device is addressed.

If an error ocurrs, the sample code jumps to “LD_FERR”. The handling of error is left to the calling routine and user interface.

APPENDIX A.

```
/*****************************************************************************/
/* MPA1000 series FPGA configuration logic with SPI                          */
/* HC11 companion PAL                                                        */
/*****************************************************************************/
Device      p22v10lcc;
/***  Pin Assignments                                                        */
/* Clock and Inputs                                                          */
PIN  2       = MCEK;      /* Input –  Register Clock                       */
PIN  3       = MISO;      /* Input –  Serial data, HC11                    */
PIN  4       = XFR8;      /* Input –  Transfer control, HC11               */
PIN 11       = GO_SPI;    /* Input –  Master Control,  HC11                */
PIN  6       = DCLK;      /* Input –  CLK feedback from FPGA               */

/* Outputs and State variables                                               */
PIN 20       = CLK;       /* Output – clock   to FPGA                      */
PIN 21       = D0;        /* Output – Data out to FPGA                     */
PIN 23       = ST3;       /* State  – var 3                                */
PIN 24       = ST2;       /* State  – var 2                                */
PIN 25       = ST1;       /* State  – var 1                                */
PIN 26       = ST0;       /* Output – SPI clk to HC11                      */
                          /* State  – var 0 (Dual function)                */
PIN 27       = END; /* State  – End condition        */

/*** Declarations and Intermediate Variable Definitions                     */
field count = [ST3..0];    /* declare counter bit field                     */
$define S0 'b'0000         /* define counter states                         */
$define S1 'b'0001
$define S2 'b'0010
$define S3 'b'0011
$define S4 'b'0100
$define S5 'b'0101
$define S6 'b'0110
$define S7 'b'0111
$define S8 'b'1000
$define S9 'b'1001
$define SA 'b'1010
$define SB 'b'1011
$define SC 'b'1100
$define SD 'b'1101
$define SE 'b'1110
$define SF 'b'1111


/*** Logic Equations                                                        */

CLK.d        =      !CLK &  DCLK
                          /* High if idle or restore state                 */
               #    !CLK & !DCLK & !(!ST3 & !ST2 & !ST1 & !ST0);
                          /* High if data ready in active state            */
CLK.sp       =      'b'0;
CLK.ar       =      'b'0;

END.d        =      ST3 & ST2 & ST1 & ST0
               #    END & !XFR8;
                          /* Count has expired but no new XFR8             */
END.ar       =      'b'0;
END.sp       =      'b'0;

D0           =      !GO_SPI &  MISO     /* active low GO_SPI                */
               #     GO_SPI & XFR8;     /* Slave Select signal              */
```

```
ST0.ar       =     'b'0;
ST1.ar       =     'b'0;
ST2.ar       =     'b'0;
ST3.ar       =     'b'0;

ST0.sp       =     'b'0;
ST1.sp       =     'b'0;
ST2.sp       =     'b'0;
ST3.sp       =     'b'0;

ST0.oe       =     !GO_SPI;     /* Master control                        */

sequence count {                /* free running counter                  */

present S0   if    !DCLK & !END & !XFR8
               next S1;
             if     DCLK
                 # !DCLK &  END
                 # !DCLK &  XFR8
               next S0;

present S1   if    !XFR8 &  DCLK
               next S2;
             if    !XFR8 & !DCLK
               next S1;
             if     XFR8
               next S0;

present S2   if    !XFR8 & !DCLK
               next S3;
             if    !XFR8 &  DCLK
               next S2;
             if     XFR8
               next S0;

present S3   if    !XFR8 &  DCLK
               next S4;
             if    !XFR8 & !DCLK
               next S3;
             if     XFR8
               next S0;

present S4   if    !XFR8 &  !DCLK
               next S5;
             if    !XFR8 &  DCLK
               next S4;
             if     XFR8
               next S0;

present S5   if    !XFR8 &  DCLK
               next S6;
             if    !XFR8 & !DCLK
               next S5;
             if     XFR8
               next S0;

present S6   if    !XFR8 & !DCLK
               next S7;
             if    !XFR8 &  DCLK
               next S6;
             if     XFR8
               next S0;
```

```
present S7    if    !XFR8 &  DCLK
                next S8;
              if    !XFR8 & !DCLK
                next S7;
              if     XFR8
                next S0;

present S8    if    !XFR8 & !DCLK
                next S9;
              if    !XFR8 &  DCLK
                next S8;
              if     XFR8
                next S0;

present S9    if    !XFR8 &  DCLK
                next SA;
              if    !XFR8 & !DCLK
                next S9;
              if     XFR8
                next S0;

present SA    if    !XFR8 & !DCLK
                next SB;
              if    !XFR8 &  DCLK
                next SA;
              if     XFR8
                next S0;

present SB    if    !XFR8 &  DCLK
                next SC;
              if    !XFR8 & !DCLK
                next SB;
              if     XFR8
                next S0;

present SC    if    !XFR8 & !DCLK
                next SD;
              if    !XFR8 &  DCLK
                next SC;
              if     XFR8
                next S0;

present SD    if    !XFR8 &  DCLK
                next SE;
              if    !XFR8 & !DCLK
                next SD;
              if     XFR8
                next S0;

present SE    if    !XFR8 & !DCLK
                next SF;
              if    !XFR8 &  DCLK
                next SE;
              if     XFR8
                next S0;

present SF    if    !XFR8 &  DCLK
                next S0;
              if    !XFR8 & !DCLK
                next SF;
              if     XFR8
                next S0;
}
```

APPENDIX B.

```
/*******************************************************************************/
/* LOGIC reduced to AND-OR equations (LISTING)                                 */
/*******************************************************************************/

CLK.d  =>
    !CLK & DCLK
  # !CLK & !DCLK & ST3
  # !CLK & !DCLK & ST2
  # !CLK & !DCLK & ST0
  # !CLK & !DCLK & ST1
D0 =>
    !GO_SPI & MISO
  # GO_SPI & XFR8
END.d  =>
    ST0 & ST1 & ST2 & ST3
  # END & !XFR8
ST0.d  =>
    !DCLK & !END & !ST0 & !ST1 & !ST2 & !ST3 & !XFR8
  # !DCLK & !ST0 & ST1 & ST2 & ST3 & !XFR8
  # !DCLK & ST0 & !ST2 & !ST3 & !XFR8
  # !DCLK & !ST0 & ST1 & !ST3 & !XFR8
  # !DCLK & !ST1 & ST2 & !XFR8
  # !DCLK & ST0 & ST1 & ST2 & !XFR8
  # !DCLK & !ST2 & ST3 & !XFR8
ST1.d  =>
    !DCLK & ST0 & ST1 & !XFR8
  # DCLK & ST0 & !ST1 & !XFR8
  # !ST0 & ST1 & !XFR8
ST2.d  =>
    !DCLK & ST0 & ST1 & ST2 & !XFR8
  # DCLK & ST0 & ST1 & !ST2 & !XFR8
  # !ST0 & ST1 & ST2 & !XFR8
  # !ST1 & ST2 & !XFR8
ST3.d  =>
    !DCLK & ST0 & ST1 & ST2 & ST3 & !XFR8
  # DCLK & ST0 & ST1 & ST2 & !ST3 & !XFR8
  # ST0 & ST1 & !ST2 & ST3 & !XFR8
  # !ST1 & ST3 & !XFR8
  # !ST0 & ST1 & ST3 & !XFR8
```

APPENDIX C.

```
; Code Excerpts for HC11 using SPI to configure MPA1000 series FPGA

; ***************
; *     Define  *
; ***************
DDRD:  equ   REGBS+$09    ; port D Data Direction reg
SPCR:   equ   REGBS+$28    ; spi control reg
SPSR:   equ   REGBS+$29    ; spi status reg
SPDR:   equ   REGBS+$2A    ; spi data reg
PORTG: equ   REGBS+$7E    ; port G data reg
PORTH: equ   REGBS+$7C    ; port H data reg

CFG1:   equ   $11          ; Page 1     image for memory mapper
CFG2:   equ   $22          ; Page 2     image for memory mapper
; more or less depending on number of FPGA devices in chain
CFGn:  equ   $ff          ; Page n     image for memory mapper
```

```
; ***************
; *     RAM     *
; ***************

       org    $200          ; FLEXVAL ram area
SPI_ER: ds.b  2             ; Address of failed transfer if any
;SEC_IMG: ds.b      1       ; Sector image number for current 16K byte block
                            ; in external FLASH EEPROM paged at address $4000
                            ; Consider writing to this address for proper page
                            ; selection of the ConfigWARE image.
==============
; **********
; Enable FPGA Transfers via SPI
; Entry       :      none
; Exit :      SPI is enabled as SLAVE
;             :      GO_SPI is active
; **********
EN_FPGA:
       bset   PORTH,#$00    ; make sure XFR8 is inactive (high)
       bclr   PORTH,#$04    ; assert GO_SPI
       ldaa   #$44          ; SPE = 1 and CPHA = 1, rest 0
       staa   SPCR          ; to SPI control register
       bset   DDRD,#$04     ; MISO is made output PD[2]
       bset   PORTG,#$40    ; negate RESET1* at bit-6
       rts

; **********
; Disable FPGA Transfers via SPI
; Entry       :      none
; Exit :      SPI is disabled
;             :      GO_SPI is inactive
; **********
DI_FPGA:
       bset   PORTH,#$00    ; make sure XFR8 is inactive (high)
       bset   PORTH,#$04    ; negate GO_SPI (high)
       bclr   DDRD,#$04     ; MISO is made input again PD[2]
       ldaa   #$04          ; SPE = 0 and CPHA = 1, rest 0
       staa   SPCR          ; to SPI control register
       rts

; **********
; Transfer byte to FPGA via SPI (ignore SPI errors if any)
; Entry       :      x  = pointer to the byte
; Exit        :      a  = 0 if o.k.
;             :      a != 0 if time out
;             :      x  = x + 1 if o.k. else x is unchanged
; **********
BY2FPGA:
       ldaa   0,x
       staa   SPDR          ; write to SPI data register
       bclr   PORTH,#$01    ; XFR8 = low (active)
       ldaa   #$ff          ; time out counter
BY2_F1:
       tst    SPSR          ; check if spif is set
       bmi    BY2_F2        ; jump if transfer complete
       deca                 ; not complete, decrement time out
       bne    BY2_F1        ; check for time out
BY2_F9:
       stx    SPI_ERR       ; address at which error occurred
       ldaa   #$ff          ; indicate error condition
BY2_F8:
       bset   PORTH,#$01    ; XFR8 = high (inactive)
       rts
```

```
BY2_F2:
       tst    SPDR          ; dummy read to clear SPIF bit in SPSR
       ldaa   SPSR
       bne    BY2_F9        ; some error ?
       inx
       bra    BY2_F8        ; normal exit

; **********
; Transfer image to single FPGA (MPA1036 case)
; Entry      :      SEC_IMG has the current sector
; Exit       :      passes end condition of BY2FPGA to calling routine
; Notes      :      MPA1036 has 116800 bits
;                      14600 bytes as follows
;                        5 bytes of header
;                     14595 bytes of data as follows
;                       139 rows containing
;                       105 bytes/row
;                     address of byte after end of image
;                     $3908 (when starting address is $0 )
;                     $7908 (in this code, as ConfigWARE image is at $4000 )
; **********
IM2FPGA:
       ldx    #$4000        ; start of image in FLASH
IM2_F1:
       jsr    BY2FPGA
       tsta
       beq    IM2_F2        ; jump if no error
       rts                  ; pass error up
IM2_F2:
       cpx    #$7908        ; end of image
       bne    IM2_F1        ; keep looping
       rts                  ; return with a = 0

; **********
; Reset  FPGAs
; Entry      :      none
; Exit :      FPGAs are forced into reset state
; **********
RST_FPGA:
       bset   PORTH,#$00    ; make sure XFR8 is inactive (high)
       bset   PORTH,#$04    ; negate GO_SPI
       bclr   PORTG,#$40    ; assert RESET1* at bit–6
       rts

; **********
; Program FPGAs
; Entry  : none
; Exit   : FPGAs are loaded and active unless error
;          : In error case, a jump to LD_FERR routine (not shown here) is
made.
; **********
LD_FPGA:
       jsr    EN_FPGA

LD_F1:
       ldaa   CFG1
       staa   SEC_IMG
       jsr    IM2FPGA       ; do transfer
       tsta
       bne    LD_FERR       ; jump if error

LD_F2:
```

```
        ldaa   CFG2
        staa   SEC_IMG
        jsr    IM2FPGA
        tsta
        bne    LD_FERR             ; jump if error

; more or less segments depending on number of FPGA devices in chain

LD_Fn:
        ldaa   CFGn
        staa   SEC_IMG
        jsr    IM2FPGA
        tsta
        bne    LD_FERR             ; jump if error

LD_END:
        jsr    DI_FPGA
        rts
```

APPENDIX D.

ConfigWARE RAM Array Sizes for MPA1000 Device Family

	MPA1016	MPA1036	MPA1064	MPA1100
HEADER (bytes)	5	5	5	5
ROWS	95	139	183	227
Bytes/Row	72	105	138	170
Total Bytes	6845	14600	25259	38595
Total Bytes (Hex)	1ABD	3908	62AB	96C3
RAM Array Size (Relative to MPA1016)	1.00	2.13	3.69	5.64
Number of Cells (Relative to MPA1016)	1.00	2.25	4.00	6.25

APPLICATION NOTE
Effective Synthesis Techniques for MPA1000 Devices

by Thomas G. Felske and Wanhao Li, Motorola Programmable Logic Products

Introduction

Logic synthesis has become an increasingly important issue in the FPGA design area due to the rapid growth of the FPGA design complexity. Many FPGA vendors offer synthesis design flows for designers who prefer synthesis over schematic design entry. This paper presents the critical architectural components of the MPA device and HDL techniques to achieve the best design performance from the MPA10000 synthesis design flow. The synthesis design flow includes Synopsys Design Compiler (Version 3.3b), Exemplar Galileo (Version 3.1), and Mentor Autologic which all map to the Motorola MPA1000 FPGA technology. Although this application note focuses on the Synopsys tool, most of the techniques are tool–independent and shall apply to the other synthesis tools.

Direct Mapping For Design Compilers

One of the biggest advantage of using the MPA1000 synthesis flow is that the MPA architecture requires a single mapping process for synthesis. The MPA1000 architecture is fine–grained and each basic cell can be configured directly to logic gates such as AND, OR, XOR, and Multiplexer. Since the MPA technology mapping is very ASIC–like, the synthesis tools can map the design with regular ASIC logic optimization algorithms. Consequently, this provides an convenient FPGA synthesis environment for designers who have ASIC experience or want to retarget FPGA designs to an ASIC at a later time. This is important to Motorola to be able to use the same tool for both ASIC and FPGA design. It not only saves designers from buying another tool but also prevents them from having to learn another synthesis tool.

To re–target Synopsys' Design Compiler from the FPGA technology to ASIC, Design Compiler's ".synopsys_dc.setup" file is modified to link to the ASIC target library. Upon startup, Design Compiler loads the library setup file to link to the desired synthesis library. The command lines in the setup file that link a target library are:

technology = ASIC or FPGA technology name

link_library = {technology + ".db" .....}

target_library = {technology +".db"}

search_path = {Regular Synopsys path ASIC or FPGA library path}

Logic Optimization and Technology Mapping

MPA1000 Architecture Resources

When partitioning functional blocks at the system level, the design capacity of the MPA device must be known to calculate how many FPGAs may be required for the system design in order to distribute the logic amongst the FPGAs. The calculations are bound by the physical limitations of the MPA device. The following sections define the physical limitations of the MPA architecture for various design resources. The limitations are related to the routing resources available to connect to the clock and reset pins for IOBs, wire–or and wire–and bus limits, clock resources, and set/reset resources for registers.

Although most of the logic optimization is done at the synthesis level, the MPA1000 backend system further optimizes logic by stripping back logic of unused output pins or signals, and optimizing inverters by cancelling out or pushing the inversion into the driven macro's input signal (bubble pushing).

Clock Signal Resources

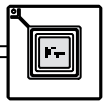
Clock resources can use up to eight dedicated clock pad sites to connect to the dedicated clock tree resource.

The core array contains routing paths to go from primary clock resources to regular routing resources and vice versa. Therefore, the clock pins of the registers located inside core array can be routed from either primary clock tree or secondary clock tree which are implemented by regular routing resources. However, the clock pins of the I/O registers can only be driven by primary clock resources. If a clock signal is generated inside the core array, it can only drive registers inside the core array unless the signal is connected to the primary clock tree.

Regular routing signals inside the core array can be routed to primary clock resources and back, some high–fanout regular signals can be implemented with the primary clock tree and improve both routing congestion and routing delays. However, the designer must be aware of the limitation of the total number of primary clock trees.

Due to potential routing resource limitation, the total number of clock and reset signals (primary and secondary) should not exceed 15. If the total number exceed 15, place and route is very likely to fail.

Only five different clock/reset signals are allowed in each cell zone (10x10 cells) since each cell column is connected by the same clock signal. Among the five signals, only two can come from primary clock tree. In the I/O area, five I/O macros are aligned as an I/O segment and connected to a cell zone. Each I/O segment only allows 2 clock signals which need to come from primary clock trees. In general, place and route tools handle all the registers and clock signal placement to make sure no violations occur. However, if the designer changes or influences the cell or I/O placement manually by a control file, the clock signal limitation has to be followed. For instance, if all the I/O pins are fixed by a control "<design>.pat" file, it will cause partition failure if three of the I/O pins inside a I/O segment are driving three different clock signals.



Three–State/WOR Resource

The MPA1000 devices offer real three–state signals only in the I/O areas. Inside the core array, WOR structures are supported instead of an three–state structure. The I/O macros can utilize an WOR structure that connects to the P–bus resource. The internal WOR bus requires the designer to utilize the WPUP pull up macro with each bus and the PWPUP pull up macro for each P–bus WOR structure.

IOB Resource

When a designer assigns “port_is_pad” attributes and “insert_pads” to the top–level I/O “ports” that do not contain instantiated I/O macros, each I/O port will be inferred as an I/O macro. The default I/O inference for MPA1000 devices are:

clock input port	IPCLK
regular input port	IPBUF
regular output port	OPBUF
three–state port	OPTBUF

Each input macro (IPBUF) can be configured as TTL or CMOS levels. Each output macro (OPBUF) can be configured into 3V/5V, high drive or low drive, slow slew or fast slew rate. Those parameters can be inserted into the system through several different ways. The most convenient way is to use Design Compiler’s “set attribute” command as follows:

```
set attribute {I/O instance name} attribute_name
attribute_value –type string
```

The I/O properties can also be inserted in a separate ASCII control (<design>.pat) file which will input those properties during the backend “import” stage. Refer to the on–line documentation for details of the “<design>.pat” control file.

There are two I/O structures that cannot be inferred by an behavior description; the open–drain structure and the registers inside I/O macros. Designers who need to use these two structures need to instantiate the desired macros. The following is a VHDL example for an open–drain macro:

```
U01: OPBUF port map (O=>internal_signal,
EXTOUT=>output_port);
```

A list of the special I/O macros with registers are in the on–line manual for more details on the available macros.

Peripheral Bus Resources (to route to IOB clock/reset pins)

The MPA1000 architecture contains an eight bit bus that runs the lengths of the chip next to the IOBs. This is the peripheral bus or P–bus. This resource is used for an interface resource for the IOB pins and global routing. This resource is not automatically used by the routing tool and it is up to the designer to instantiate the special P–bus macros for access to and from the P–bus. The special buffers are the APBUF, PABUF, PWPUP, APWBUF, and APWINV. The prefix AP refers to an Array to P–bus connection and visa versa for PA. The W refers to the wire–or capability.

Set/Reset Resources

The internal core and the IOB registers have asynchronous set and reset capability that are mutually exclusive. The register contains a single pin that can be programmed for a set or reset function. The dedicated low skew reset tree is physically the same as the clock tree until the signal enters a zone. There the port cell(s) direct the signal to either a clock pin or reset/set pin.

MPA1000 Design Resources

The MPA1000 synthesis library contains special macros that aid in the efficiency of the FPGA design. Efficiency shall be described as guiding critical routes of the design to special or dedicated routing resources, utilizing special function macros optimized for the MPA architecture, and utilizing MPA functional resources that can be only be structurally instantiated from the MPA synthesis library. The following sections describe the special types of macros that can be found in the MPA synthesis libraries. These macros are for specific design requirements. Please peruse the synthesis libraries for more macro specific details.

Clock Tree macros

The clock tree macro resource consists of the IPCLK clock pad macro and the ACLK internal clock buffer. The “ACLK” macro must be instantiated by the designer to connect internally generated clock signals to the primary clock tree. An example of a structural description applying an internal clock buffer signal to the primary clock tree.

```
(VHDL) Buffer_instance_name: ACLK port map
(A=>clk_input, Q=>clk_output);

(VERILOG) Buffer_instance_name ACLK
(.A(clk_input), .Q(clk_output));
```

The “clk_output” signal will be driving register clock pins from the primary clock tree.

There are only eight primary clock signals in a MPA1000 device. The total number of IPCLK and ACLK macros must not exceed eight. If it does, partitioning failure will occur.

Three–state/WOR macros

To use WOR structures in the MPA1000 devices, There are several macro resources. They are WBUF, WINV, WND2, WOR2 for an internal WOR bus and there are several IO macros that have the WOR capability that utilize the P–bus resource. An example would be the IPWINV or APWBUF macros. For each WOR bus created, an pull up macro must be attached. The pull up resources are WPUP for the internal WOR bus, and PWPUP for the P–bus pull up. When creating an internal or P–bus WOR structure, both the WOR buffer and the pull up macros must be instantiated. Currently, there is not a method to inference an WOR structure when using Synopsys. The following is an VHDL example of an internal WOR instantiation :

```
U01:  WBUF port map (A=>sig1, W=>worbus);
U02:  WBUF port map (A=>sig2, W=>worbus);
U03:  WBUF port map (A=>sig3, W=>worbus);
U04:  WPUP port map (W=>worbus);
```

The “WPUP” macro is necessary to pull up the wor bus signal. The on–line documentation has more information on how many “WPUP” macros should be placed on the wired–or bus.

Three state functionality can be inferred in an HDL when it will be used in the I/O area of the FPGA. The “m” signal in this

example needs to be an external I/O signal which is assigned “out port” “ port is pad” attributes . A three–state inference:

```
if (sli = ‘1’) then
m = e; else m = ‘z’;
```

IO macros

The MPA input output block (IOB) is a complex logic block with data registers on the input and output data signals. The registers cannot be inferred with RTL code, but can be accessed through a structural description. In general when the synthesis tool places pads onto the design’s peripheral signals, the pad macros are simple pad buffers e.g. IPBUF and OPBUF. To utilize the registers in the IOB, an structural description must be used. There are more that seventy different IOB macros. The on–line help describes the many different types.

Clock Signals/Clock Tree Generation

Without careful attention, clock tree implementation can cause problems for both synchronous and asynchronous designs. There are several clock tree related issues, the HDL designer should know :

- Design Compiler does have the capability to infer clock pads for those assigned clock signals. Internal clock buffers require a structural description. However, Design Compiler cannot insert internal clock buffers automatically. For instance, a gated clock signal which needs to drive the clock pins in I/O flip–flops will require an internal clock buffer to get onto the primary clock tree. This is an MPA1000 architecture limitation since the I/O register clock pins can only be accessed from the dedicated clock tree. By structurally inserting an internal clock buffer, the gated clock signal will connect to the dedicated primary clock tree through the clock buffer. Since the number of primary clocks (buffers and pads) of MPA1000 is eight, it is important to limit the total number of inferred clock pads/buffers and inserted clocks to eight.
- If a signal is assigned as an “input port” and as “clock”, Synopsys will map the signal into a “IPCLK”. Even if the designer doesn’t assign “clock” for an “input port” signal, if the input signal is driving a clock input of a register directly, it will also be assigned as an “IPCLK” macro automatically. All the IPCLK pads will connect to the primary clock tree by default. The designer must make sure that “don’t touch” was assigned to all those primary clock trees. If Synopsys “optimize” or “balance” the clock tree, it might generate many secondary clock trees which ends up causing clock tree routing problems.
- If an internally generated clock signals need to drive both primary clock tree and a regular output pad, a separate output macro “OPBUF” needs to be added to bring the signal to an output pad. The following is a VHDL example:

```
clkbuf1: aclk port map (A=>internal_clock,
Q=>primary_clock);

outbuf1: opbuf port map (O=>internal_clock,
EXTOUT=>output_pad);
```

General HDL Techniques

The description of an design in an HDL is very important. The synthesis tool interprets the description and then maps the design to the target library. Performance of the design depends on the interpretation of the HDL description and the style of HDL coding. These influence the logic that will be mapped to the design.

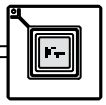
Synopsys does not optimize logic based on XOR logic reduction (Mueller–Reed). However, since 50% of the MPA1000 cells can be implemented as an “XOR”, the designers can build macros based on XOR logic. The outcome of the XOR based logic can potentially be faster and use fewer number of cells.

- Merge registers into the I/O macros. Registers that are directly connected to input macros or output macros with out feedback can be pulled into an I/O location by instantiating MPA I/O macros such as “OPDFR” and “OPDLR” in an HDL design. If specialized I/O macros are not used, the register resource will come from inside the core array area.
- For multiple clocks within a design, it is recommended to implement a clock enable signal for the various registers and have the registers all clocked with the top level clock.
- For state machines, an important issue when compiling a state machine is that the one–hot state assignment approach is very viable for a fine–grained architecture such as the MPA1000. In general, Design Compiler supports five types of state assignment: manual, auto, one–hot, binary, and gray. The one–hot state assignment assigns one unique flip–flop per state. In coarse–grained architectures such as Xilinx 3000/4000, there are large combinational circuits attached before the flip–flops in each block (CLB). One–hot state assignment will cause significant penalties of areas for 3000/4000 architectures. On the other hand, the MPA1000 architecture has no combinational circuits attached to each flip–flop within a cell. The area penalty is, therefore, minimal. Since our both designs are utilizing less than 30% of the cells, In general, it is recommended to use the one–hot state assignment if timing is more critical than area. The “one–hot” mode in most cases achieves faster timing due to its much simplified combinational logic.

Synchronous Design Style

The synchronous design style is the preferred method of design recommended by synthesis tool and FPGA vendors. Few designers use the asynchronous design style since it can result in a little bit faster performance in an ASIC or custom design environment. In an FPGA environment, this assumption can be false. In general, asynchronous clock signals are not driven from primary clock resources and are therefore implemented in an “secondary clock tree” in MPA1000. The secondary clock trees use the regular routing resources to route the signal and have significantly longer clock delays and clock skews. The extra clock signals also occupy routing resources such as ports and global busses that result in increased routing congestion. All those combined, the asynchronous design typically will not get the faster–speed advantage the designer might expect.

The only asynchronous design styles suggested by Synopsys are designs with gated–clock and designs with asynchronous reset. Designers who use gated–clocks have to



be aware of all the asynchronous clock tree problems that exist.

The asynchronous reset, however, is a popular design style and generally will not cause serious design problem since it is usually implemented as a global signal and tends to be mapped into primary clock/reset tree.

Area Estimation and Area Optimization

The designer should do some rough area estimations before trying to fit the design into the target MPA device. The rule of thumb is :

- The total number of cells (components) created by synthesis should not exceed 40% of the total raw cells of each device since some cells will be used as routing resources. The routability is usually design–specific. However, if there are more high–fanout nets, the routability usually decreases. There is still some chance to route designs with more than 40% cell utilization. However, the performance usually is not as good and the probability of successful routing is much lower.
- The total number of flip–flops should not exceed 80% of raw flip–flops.
- The total number of clock/reset signals should not exceed 15.
- In general, device area is not a concern for FPGA unless it cannot be fitted into a device. There are some techniques to improve device areas:
- Instantiate special I/O macros to utilize registers inside I/O macros. There are two registers available inside each I/O macro. It will reduce the device area, reduce the required routing resource, and help in synchronizing on/off–chip timing.
- Use wired–or structure can reduce the number of gates. However, special attention should be given to timing and routability issues.
- Using the primary clock tree to implement high–fanout net will not only improve the wiring delays but also save a lot of routing resources. However, an active primary clock tree will consume more power than an idle clock tree. If power consumption is an important issue, designers should do some calculations as to the trade offs of using the clock tree.
- Use Synopsys techniques of resource sharing and area optimization options.

Techniques for Timing Optimization

In general, most of the timing improvement will come from the design and HDL code changes. By knowing the behavior of HDL compilers will generally lead to big improvements in device timing. the following describes some of the common timing optimization techniques used in HDL code structure and design synthesis:

- Avoid long “if_elsif” or “case” statements. Each elsif or “when” statement usually will add at least one logic level to the circuit. A better technique would be to see if the statements can be broken down and inserted into different concurrent processes or blocks. The same thing applies to the other statements inside each process. Since they are all implemented sequentially, try to see if they can be implemented as concurrent statements outside of the process.
- Be aware of the default inferred latches and registers. Make sure that the if/elsif statements always close with an “else” statement if no latch element is expected.
- High–fanout nets usually cause very long wiring delays. Try to reduce the number of fanouts if possible.
- Use clock file to differentiate slow clock and fast clock. It will help timing–driven layout to optimize and report the right critical paths.
- Use Synopsys “timing optimization” option.

Timing Analysis and Delay Estimation

Since FPGA place & route can take a long time for larger designs, it is important to do timing analysis and delay estimation during the synthesis stage to avoid many design iterations. In general, accurate FPGA delay estimation is very difficult to achieve due to large metal wire delays and large switching element delays. However, by using component delays and wiring delays derived from statistical analysis, some obvious problems can be identified and avoided in the early design stages.

Critical path analysis is the most popular way to estimate device frequency. The post–layout critical–path delays are reported in the “timing file” (design.tim). The path delays in the timing file include both the component and wiring delays. Each level of logic in the critical path needs at least 1.2ns (0.7ns cell delay and 0.5ns direct interconnect delay). In general, for delay paths longer than 5 logic levels, the average delay for each level of logic is between 2.5ns to 3.5ns.

To estimate the critical path delay, the designer should count the levels of logic of the critical path by using the “highlight_path” command to highlight the critical path. Assuming the number of logic levels for the critical path is N, the theoretical lower bound of the critical path delay (LD) can be calculated with the following equation:

LD = 1.2 *N + setup + clk_to_q

For the MPA1000 family, the setup time is about 1ns while the clk_to_q is about 1.5ns.

In a real design, the average delay for each level of logic is around 3ns. Therefore, the expected delay (ED) for the critical path should be:

ED = 3*N + setup + clk_to_q

General timing analysis guidelines for MPA1000 synthesis designers are:

- If timing budget is smaller than or only slightly more than LD, there is no chance to reach the timing goal. The designer should try to modify the design by modifying the HDL code structure, or modifying the timing budget to solve the problem.
- If the timing budget is close to ED, the place&route tool has a good chance to reach the timing goal. The designer should utilize the timing optimization techniques in the Synopsys environment and use timing driven layout options for the place and route software. For a multiple clock system, the slower clock usually will not create a critical path. However, the software will not be able to differentiate the less critical clock signal unless the designer creates a timing group for

each clock in the “clock file”. Please refer to the online documentation for detailed information regarding the “clock file” format.

- If the timing budget is far exceeding ED, the designer will have a large slack time for the critical path. Even though timing–driven layout is still suggested for place&route, the designer may have some flexibility in using a less aggressive timing goal to reduce run time and focus more on the area optimization when synthesizing the design.

Behavioral vs. Structure Synthesis

In an traditional synthesis flow, structure design is generally recommended at the top–level for constructing logic hierarchies. Behavioral design, however, is usually much easier and maintainable for designers. For example, in an FPGA design, critical portions of the design utilized the structural description method to capture specific logic modules. One area that used an structural description was the I/O logic module. Although some FPGA technology mappers have the capability to do special mapping processes for the I/O, such as mapping flip–flops into an I/O, it is generally much more controllable to construct an I/O logic module with an structural description. It is also easier to insert attributes to the macros which are described structurally. The current MPA1000 synthesis library includes a large variety of I/O macros which have many combinations of input/output, clock, flip–flop/latch, and delay elements. By instantiating I/O macros directly from the MPA1000 library, the designer can avoid some potential problems caused by the synthesis logic inference process.

An interesting note when using an structural description is that Design Compiler is still capable of optimizing the structural description even it is supposed to be optimized by the designer already. It can be concluded that the software is usually capable of performing a much more thorough logic optimization search compared to human beings under a well–defined environment. In one example, an 20% improvement was achieved by running optimization on hand crafted structural modules.