Design Methodology for a MIPS Compatible Embedded Control Processor

Raymond Peck with Jay Patel
LSI Logic Corporation
1525 McCarthy Boulevard
Milpitas, California 95035
rpeck@lsil.com, jayp@lsil.com

Abstract
This paper describes the design methodology used for the LR33000 MIPS-1 Architecture-compatible highly integrated embedded-control processor. A mix of synthesis and schematic entry, hand and automatic optimization was used for logic design. In addition, a mix of hand and automatic layout was used to incorporate custom-style datapath layout, a CPU megacell, caches, and a high level of integration in a cell-based ASIC. The short (10 month) design time and bug-free first-run silicon have demonstrated the success of this design methodology.

1 Introduction
This paper describes the design methodology of a 700,000-transistor, 50MHz MIPS-1 compatible embedded control processor, the LR33000. This single-chip processor consists of an R3000-compatible CPU, 8 KB of instruction cache, 1 KB of data cache, DRAM controller, write buffer, timers, and a programmable system interface which directly connects to DRAM, SRAM and PROM.

The design engineers used a combination of techniques to do the logic design, verification, and layout of the LR33000, using all standard tools. The design constraints and each of these methods will be described below, accompanied by a figure showing the project schedule and manpower.

1.1 Direction and Definition
The object of the project was not just the LR33000 design itself, but the development of a reusable R3000-compatible CPU core that could be the basis for additional customized processors. The LR33000, which was the first use of this new core, was targeted to high speed, low cost embedded control applications (laser printers, network controllers, etc.). From the beginning of the definition stage, the chip's system-level features were customer-driven: customers desired a high level of integration on-chip to reduce board cost and area. The DRAM-controller, timers, cache requirements, 8-bit boot PROM interface, and other aspects were developed in a tight relationship with alpha-test customers.

It became evident that development of highly integrated chips such as the LR33000 around the core CPU would require extensive system simulation, which would have been difficult working from a layout database of the CPU. It was decided that the CPU core should be implemented from scratch from the R3000 architectural definition [1], rather than using a layout "cut-and-paste" technique to develop the integrated processor.

Reimplementation had additional advantages: it allowed for the addition of several useful new debugging features in the system control coprocessor (CP0), and for the ability to trap on unimplemented MMU instructions and MIPS' unused opcodes. It also allowed the design to be better optimized for the company's process technology to achieve shorter cycle time at higher yield.

1.2 Constraints
Limited engineering resources and the requirement of short time-to-market brought about several constraints:
custom cells and tools had to be virtually ruled out. The entire chip, including clock distribution and gating, datapaths, and control, was built from cells in the standard LSI Logic cell-based ASIC library (0.7 micron.). The only exception was a custom 1-bit fast-match-compare cell. By using this standard cell library, we obtained the benefit of a fully static design, allowing the system designer to freeze the clock for low-power uses. In addition, only standard ASIC tools were used, the only exception being some custom software written to verify the correctness and completeness of truthtables.

Since most of the chip was to be laid out by automatic ASIC place-and-route tools, layout-related timing constraints for much of the chip could not be guaranteed. To reduce the effect that this would have on clock distribution, it was decided that a single clock should be used both internally and externally, rather than dual non-overlapping clocks. It was also decided that using recirculation to implement processor stall would be impractical, due to both area and timing constraints, since monolithic recirculating latches were not available in the standard cell library. Therefore, gated clocks were used to generate processor stall. Gate-level simulations, delay-predicted with back annotation from layout, allowed the design to be tuned to meet timing requirements. This point is further discussed below.

2 Logic Design

2.1 Definition and Division of Labor

While part of the engineering team developed a microarchitecture definition for the CPU core megacell, the rest defined the Bus Interface Unit (BIU), cache system, and DRAM control with prospective customers. After the CPU microarchitecture and the BIU definition were complete and reviewed, the design was partitioned according to functionality, layout, and pipeline stages. The CPU was divided into Register File (RF), Instruction Address (IA), Read Operand and Decode (RD), ALU, Memory Transaction and Alignment (MEM), and system control coprocessor (CPU) modules, while the CPU-external ("Cobra") part of the design was divided into the BIU, Cache subsystem, DRAM control, and Timers. These pieces were distributed among eight engineers, four for the CPU, and four for Cobra.

Working from the CPU microarchitecture, the system-level specs, and the MIPS-1 architectural definition, the engineers wrote definition documents for their respective modules. The entire team reviewed these module definitions, which outlined the organization, functionality, and interface of the modules.

2.2 Netlist Generation

A mix of tools was used to generate module netlists. The tool flow is shown in Fig. 2. In addition to the standard schematic entry tool (LSED), we had a RAM compiler, MEMCOMP, and a high-level description language (HLL), LES. We used these tools, along with the Espresso, LOP and Synopsys optimizers to generate the entire design. MEMCOMP created all the RAM arrays for the register file and two onboard caches. Statistics showing how many gates were generated with each technique are shown in Fig. 3.

LES was used extensively for both control and datapath design. Its ability to synthesize from relational expressions, structural descriptions, truthtables, and state-gener-
ating expressions allowed the team to express hand-laid-
out datapath and both synthesized and hand optimized
control with one tool. Its high compiling speed\(^1\) allowed
fast design turnaround, speeding both design optimization
and bug fixing. Short examples of the HLL techniques
used are shown in Figs 5-7.

The main CPU and CP0 datapaths were carefully de-
signed with the standard cell topocells to minimize cell and
wiring area. Each module’s datapath was implemented in
a hierarchy of bitslice “hardmacros”, culminating in a 32-
bit horizontal datapath chunk. Each of the engineer’s data-
paths were designed to connect by abutment to the next,
while reserving vertical routing tracks for flow-through
busses. Datapath layout and routing is described below.

2.3 Pre-integration Verification

It was seen that simulation, bug detection, and bug fix-
ing would essentially be a serial operation immediately af-
after integration. This bottleneck was reduced by doing fairly
extensive pre-integration testing of the modules. The engineers tested their individual modules either with hand-
generated vectors or with behavioral models, some of
which were quite sophisticated. In one example, the multi-
plier/divider was tested with a behavioral model which
checked the results of thousands of instructions. In anoth-
er, the RD, IA, MEM, and RF modules were pre-integrated
together with a behavioral ALU and system which loaded
and ran assembler code. This pretesting, which took less
than a week, made the integration and verification task
quite simple: it took 10 days from the start of module inte-
gration to have the first ATV (Architectural Test Vector)
assembler program successfully run in gate-level simula-
tion. During these 10 days, 17 bugs were found and fixed.
Post-integration verification and its related behavioral
models are described below, and extensively in [2].

2.4 Optimization

At all stages of design, the engineers optimized the
speed and area of their modules using static timing analy-
sis. Speed optimization was done almost completely by
updating the design source (schematic or LES file), while
area optimization was done on some blocks of the design
with the Synopsys Design Compiler tool.

At the beginning of the design phase, individual wire de-
lays were automatically estimated based upon fanout and
estimated chip area. As the layout progressed, the estimated
net delays were replaced by wire delay models extract-
ed from the layout, allowing careful optimization of clock
nets and critical paths. It was found that the delay estimator
was pessimistic in determining delays of datapath nets, es-
specially of control signals that drove across the datapaths,
due to the unusual compactness of the datapaths. There-
fore, we obtained higher performance than had been ini-
tially estimated in several areas.

At no time was Spice run on any logic circuits. The buff-
ering and gating scheme used for clocking depended on the
delay models being accurate, and bug-free chips show that
this approach to delay prediction was successful.

3 Layout

The keys to meeting both the integration and design-
time requirements of the project were the use of careful
CPU core and chip floorplanning, and a mix of standard
place-and-route control sections with hand placed-and-
routed datapaths.

3.1 Initial Floorplanning.

The initial floorplan for the CPU was developed by
looking at datapath connectivity, and arranging the CPU
modules to minimize vertical routing tracks used for buss-
es. A size estimate was developed, based on the datapath
cells called for by the module definitions, an appraisal of
how many wiring tracks would be required to interconnect
the datapath cells, and some factor for control logic. As the

1. Compiling and LOP-optimizing the RD’s 20 sepa-
rately compiled modules (1532 lines of code) takes 2.5
min. on a 16-SPEC machine, yielding 1400 gates.
layout proceeded, the floorplan was updated with actual sizes. The careful estimates turned out to be very accurate. The initial Cobra floorplan was developed by arranging the CPU core and cache RAMs to minimize bussing, and estimating the area required for cache control, BIU, DRAM controller, and write buffer logic. These were eventually placed and routed automatically between the large CPU and RAM macrocells.

3.2 Hand Macrocell Layout

As described above, the engineers designed the two datapaths in the CPU in bitslices, defining the cell layout and wire placement for each slice so that the vertical data buses and horizontal control signals for each slice connected by abutment in all four directions. Four members of the company’s layout group placed, routed, and hierarchically assembled these bitslice “hardmacros” under direction from the engineers. Eventually, 113 types of hardmacros were combined into the two datapaths seen on the layout. The main datapath seen in the die plot photo is 32 bits wide, consisting of the RF, IA, MEM, ALU and multiplier/divider. The narrow datapath on the lower right of the CPU is the CPO (System Control Coprocessor) datapath, which was folded onto itself to achieve a better floorplan. For the same reason, the 32-bit × 32 register file was designed with a 64-bit by 16 RAM to bring its width close to that of the main datapath.

3.3 Automatic Layout

After the two large datapath hardmacros were hand placed, the layout group automatically placed-and-routed the control logic around them, using a tool to indicate regions in which various control blocks should be placed. They then extracted wire delay information for simulation and static timing analysis.

3.4 Final Layout and Hand Optimization

After static timing analysis and gate-level program simulation, the engineers optimized their designs and worked with the layout group to hand-optimize the placement and routing of critical path nets. Members of the team iterated this process several times, particularly spending time improving layout of the system logic surrounding the core CPU: the automatic layout of this logic had been less constrained than that of the core, and the tools did not do as good a job of compact layout.

4 Verification

Since the CPU Core was designed to be a reusable megacell, and was developed in parallel with the BIU/Cache Subsystem/DRAM Controller, both needed to go through debugging in parallel. Three engineers wrote phase-accurate behavioral models for this purpose: one of the CPU for BIU testing, one of the BIU for CPU testing, and one of a system outside the LR33000. This last model allowed the team to run compiled object modules in gate level simulations, and worked as a memory-mapped IO Processor (IOP) for generating external events. Using these models, the chip was simulated in three ways: in mixed mode with either the CPU core or the CPU-external logic in gates with the other as a behavioral model, or as a full event-driven gate-level model. Running mixed-mode simulations gave two major advantages: both parts of the LR33000 could be debugged in parallel, and simulations ran much faster. Gate-level CPU with behavioral externals ran approximately 15 instructions / minute, while full gate-level LR33000 ran approximately 7 instructions / minute on a 16-SPECmark workstation.

Several different aspects of the design had to be verified, so different techniques were used to check them. To check architectural compatibility of the LR33000, the team wrote tests which were automatically compared cycle-by-cycle against an RTL model of the R3000. To check the functionality of each module of the design, engineers wrote specific self-checking programs which signaled the IOP if they found an error in the design. Interactions between modules (especially with exceptions and stalls) were also tested in this way. Tests of the system level logic used the IOP to simulate various external environments (e.g., different types of memory systems). To try to detect errors not contemplated, an engineer wrote a pseudo-random program and environment generator, to check the LR33000 against the RTL model. To exercise certain conditions heavily, the generator could also generate weighted non-uniform instruction mixes. For example, the asynchronous multiply and divide were heavily tested with exceptions. Finally, an alpha-site ran their application code on the gate-level model beginning just weeks after integration of the individual modules [2].

The design and verification methodology has proven successful. The PROM monitor was running in the team’s evaluation board within an hour after getting first silicon packaged parts. The alpha customer booted their application board from PROM the day they got chips, and had their entire application (a network router and gateway) running on their board four days later. As of this writing, parts have been running in customers’ systems for over 6 months, and there are still no known bugs in first silicon.

5 Acknowledgments

Acknowledgments

The authors wish to acknowledge and thank the additional members of the Cobra design team: Keith Au, Bob Caulk, Paul Chang, Sanjay Desai, Michael Fucito, Moshe Gavrielov, George Harper, Darren Jones, Mark Kwong,
6 References


/* Bottom third of IA datapath, bits 27-2. */
module BOT-27-2:
    input cibp, mrsp, j-1-in,..., iwheretopl21, inc_out, evec;
    output cib, mx-4-out;
    signal ijump, ijumpregp, incpcp:
    StTUCtUre i llcib) - FDNlC(CIBP, dly-PCLKP0:
    jr-l lijumpregpl
    = LDZAImrsp, dIy-PCLKPll;
    j-l lijumpl - LD2AI j-1-in, dly-PCLKP2l
    inc-l(incpcp) LD2AIinc-out. dly-PCLKPS):
    mx-4 lmx-4-out)
    MUX41LIevec. ijumpregp, ijump,
    incpcp, iwheretop[Ol, iwheretop[lll:
    end:

module BOT-27-2
*/

Fig 6: A Bitslice in LES
/* IA datapath, bits 31-0. */
module IADDRGEN:
    input dly-PCLKPO,. . .,MRSP[32];
    output CIAPl321, ilnkaddrp[321, cib-15:
    signal ibranchsumn[321,
    iadd outI321:
    VA3

#FOR IBIT := 0; IBIT < 31; IBIT := IBIT + 2)
    #IF (BIT < 2)
        BOT#BITicib[BIT], mx-4-0UtIBITl)
    else
        BOTbBIT[BIT], mx-4-0UtIBITl)
    end

/* Hook up main chunks of the datapath. */
TOP#BIT I.. I = TOP I.. I ;
#IF (BIT < 27)
    BOT#BIT(T11) = BOT-27-2T1;...;
else
    BOT#BIT(T11) = BOT-31-2T1;...;
    end:

end; /* module IADDRGEN */

Fig 7: A Datapath in LES

Fig 5: Relational Control Expressions in LES

Fig 3: Relational Control Expressions in LES

note: graph shows the number of engineers working on a given task over time. In many cases, an engineer was working on more than one task at a time. In cases like these, such an engineer is shown once for each appropriate task in the graph.

Fig 8: Chip Schedule and Activity