Computer Sciences Department

OpenSPLySER: The Integrated OpenSPARC and DySER Design

Jesse Benson
Ryan Cofell
Chris Frericks
Chen-Han Ho
Karthikeyan Sankaralingam

Technical Report #1685

January 2011

UNIVERSITY OF WISCONSIN MADISON

OpenSPLySER: The Integrated OpenSPARC and DySER Design

Jesse Benson Ryan Cofell Chris Frericks
Chen-Han Ho Karthikeyan Sankaralingam
University of Wisconsin – Madison
jmbenson2@wisc.edu,cofell@wisc.edu, frericks@wisc.edu,
ho9@wisc.edu, karu@cs.wisc.edu

Abstract

The Dynamically Synthesized Execution (DySE) model has been proposed to improve the energy efficiency and performance of general purpose programmable processors. We describe how a DySE Resource (DySER) block can be integrated into a processor pipeline. The block size can be adjusted based on design constraints, but we integrate an 8x8 functional unit array into a simple in-order OpenSPARC T1 pipeline. The instruction set changes and the microarchitectural interface between the DySER block and processor are described.

1. Introduction

Based on the International Technology Roadmap for Semiconductor's (ITRS) current predictions, devices will continue to double in density every 24 to 36 months over the next 15 years. However, the power efficiency of devices is expected to scale slowly from generation to generation. ITRS predicts static power consumption will remain approximately flat while dynamic power consumption will increase significantly despite power reduction techniques like low-leakage transistors, power-gating, multi-gate threshold voltages, and architectural effort to reduce switching frequency [1].

Hardware specialization has been identified as a technique to power efficiently improve performance at the cost of reduced generality and programmability. Hardware specialization examples include GPU special function units andhardware encryption, which perform significantly faster than software solutions.

The Dynamically Synthesized Execution (DySE) model has been proposed to improve the energy efficiency of general purpose processors [2]. Applications often execute in phases, and these phases can be identified at compile time. A heterogeneous array of computational units and interconnection can be configured to execute a portion of the phase's datapath. A co-designed compiler constructs application path-trees to identify phases. The compiler slices path-trees to create mappings to DySE Resource (DySER) blocks [3].

This paper describes the design of an integrated DySER and OpenSPARC T1 processor [4]. While the DySER block size can be adjusted based on design constraints, we implemented an 8x8 array of functional units. The OpenSPARC T1 instruction set was modified to add DySER specific instructions, and the simple 6-stage pipeline was modified to incorporate the 8x8 DySER block in the *execute* stage. The SPARC assembler was modified to output binaries compatible with our integrated OpenSPARC with DySER (OpenSPLySER) processor.

The remainder of this paper is organized as follows. Section 2 describes the DySER design. Section 3 describes our implementation and processor modifications. Section 4 discusses our simulation results and section 5 discusses future work. Section 6 discusses related work and section 7 concludes.

2. DySER Design

Dynamically Synthesized Execution is meant to provide the benefits of specialized hardware resources with increased generality. When an application enters a phase, the application configures the Dynamically Synthesized Execution Resource (DySER) block for that phase using instruction set extensions. Once configured, a DySER block looks like a multi-cycle specialized functional unit with addressable input and output ports to the processor and compiler. Execution proceeds with the processor sending data to the now configured DySER block. The data flows through the configured DySER block in a dataflow fashion. The processor reads the final results of the computation from the DySER block and writes the data back to the register file. Figure 1 outlines an example path-tree constructed from Blackscholes and mapped onto an example 3x2 DySER block. Additionally, normal SPARC instructions can be performed while a DySER block is performing computations.

2.1. DySER Operation

A DySER block is composed of an array of heterogeneous functional blocks connected with switches. Figure 2 breaks down the components of a DySER block (Figure 2a). The functional units (Figure 2b) compose the core of the computation array. Functional units can be multi-purpose and heterogeneous, with a configuration register determining the function to perform. Switches (Figure 2c) form the interconnection network between the functional units. Data flows into the network along the circuit-switched path in a dataflow fashion using a credit-based flow control (Figure 2d,e). The switch network and functional units are configured before use by the compiler. While the DySER block is in configuration mode, the switch network is used to send configuration data to the switches and functional units in the DySER block.

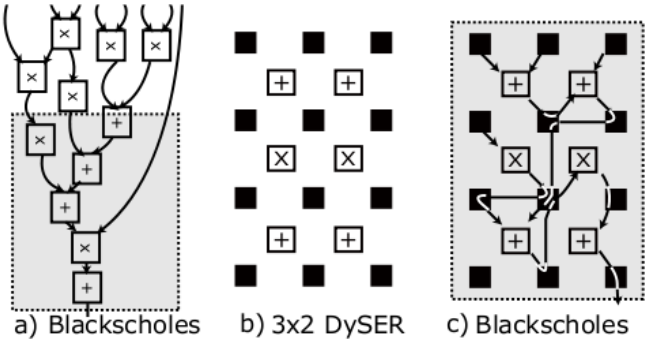


Figure 1. Mapping a phase of Blackscholes onto a DySER block.

2.2. Instruction Set Extensions

The SPARC instruction set was extended with four instructions to facilitate processor and compiler interaction with a DySER block. The instructions allow the compiler to configure the DySER block, send data from the register file to the DySER block, and send data from the DySER block to the register file. The instructions are detailed in Table 1.

3. OpenSPLySER Implementation

Before discussing the changes made to the OpenSPARC T1 architecture, we will briefly discuss the behavior of the pipeline during each of the four DySER instructions. All DySER instructions flow through the fetch and thread select stages in the same fashion as all other instructions. The only important thing to note is the register read sources are decoded but not enabled in thread select. In the decode stage, DySER instructions are decoded and DySER signals are set for pipelining [3].

Because each instruction behaves differently in execution, we describe them individually:

dyser_init: Because dyser_config_enable is high, DySER takes the configuration bits.

dyser_send: The register read enables and registers to read are piped to the register file in decode, so the data is available during execute. Both RS1 and RS2 are fed to DySER, dictated by the ports selected in the dyser_send instruction.

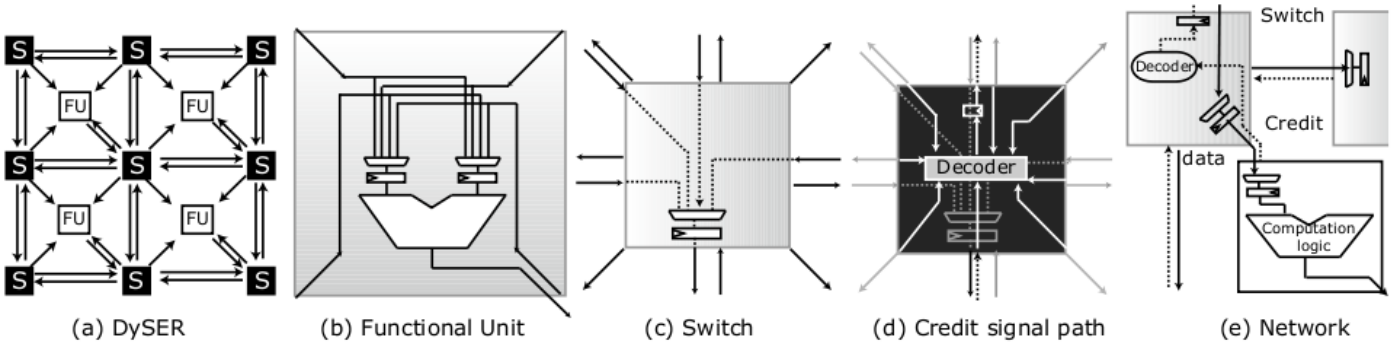


Figure 2. Major components of a Dynamically Synthesized Execution Resource.

Instruction	Description
dyser_init [config data]	The DySER block is placed in config mode, and the config data is shifted into the block. The number of dyser_init instructions to fully configure a DySER block depends on the block size.
dyser_send RS1 => DI1 dyser_send RS1 => DI1, RS2 => DI2	Reads from the register file and sends the data to a DySER block. 1 or 2 source registers are sent to the specified DySER input ports
dyser_recv DO => RD	Writes output data from DySER to the register file
dyser_commit	Signals DySER to write all ready data back to general-purpose registers and/or memory. This facilitates out-of-order execution. *Not applicable in single-issue, in-order OpenSPARC.*

Table 1. Stylized SPARC instruction set extensions to support DySER operation.

dyser_recv: Register write is enabled, the DySER register destination and DySER data out are all pipelined to memory and then writeback. In the event that the data specified by the receive port is not ready to commit, DySER will stall the processor until the data is able to come out.

dyser_commit: Commit flag going high causes DySER to writeback all of the ready results. Only required for out of order execution, and is unused in our current implementation.

Except for dyser_recv, which writes data back into register file, memory and writeback stage are the same for all DySER instructions. Register write enable is set low to prevent any writes to the register file.

3.1 OpenSPARC T1 Pipeline Changes

In order to facilitate correct DySER operation, the decode and execute stages of OpenSPARC T1 required careful modification. The DySER instructions are implemented through one of two SPARC V9 "implementation dependent" instructions already present in the pipeline. They are, however, caught as illegal instructions if not implemented and removed from the illegal instruction logic. In the decode stage, we added decode logic to decode DySER instructions and set the DySER interface signals (detailed in section 3.2) correctly. Also modified was the validation logic to enable register reads for dyser_sends and the decoding of the DySER compslice.

Pipeline latches were added for the DySER signals going to the execute stage. Here, the DySER block is added along with all of the stall logic to squash DySER from taking in any new data on a stall. A diagram of the integrated design is shown in Figure 3. Three muxes were added just after the ALU result, register write signal and register destination (as mentioned above). A DySER stall request was also added (going to the fetch control logic) on the occasion that the DySER block needs to stall the entire pipeline along with completion logic to tell the fetch stage when to resume fetching instructions again. Finally, all of the routing for DySER signals was added to the top level modules.

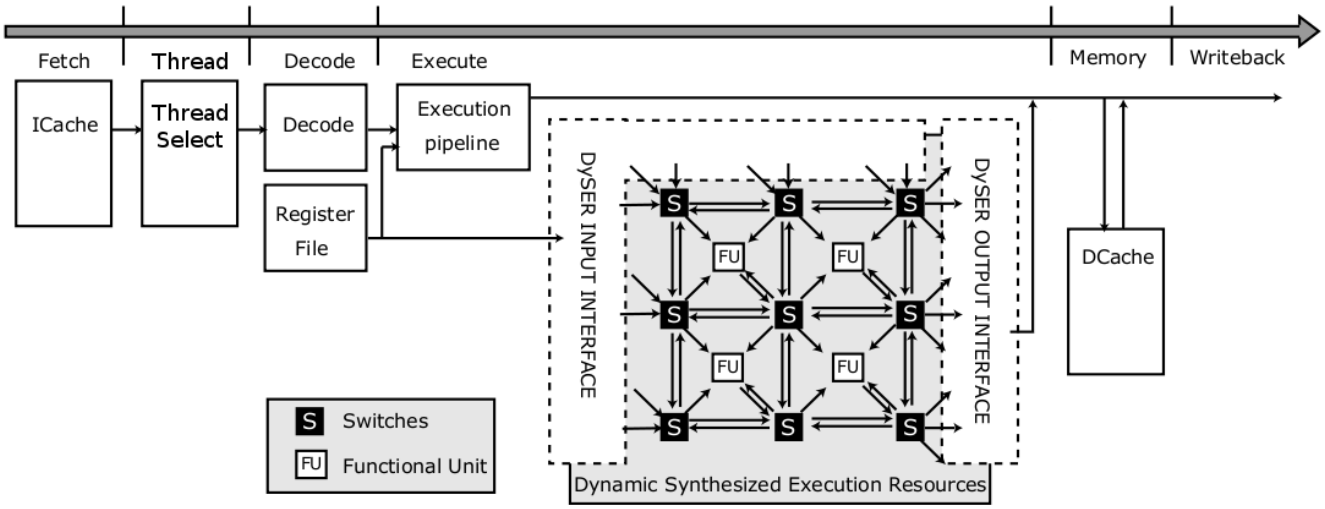


Figure 3. The OpenSPARC T1 pipeline with an integrated DySER block.

3.2 DySER-OpenSPARC T1 Interface

An input interface and output interface support the interaction between the OpenSPARC T1 pipeline and a DySER block. A DySER block has separate addressable input ports and output ports. Each input or output port contains a FIFO queue that buffers data at that port. Table 2 describes the detailed signal interface between the OpenSPARC processor and a DySER block. The OpenSPARC register file contains three read ports, and two write ports. Due to the 32-bit instruction encoding limitation, only two source registers can be read per cycle for a DySER instruction. The second write port is used to allow long running memory accesses to write results to the register file immediately when non-memory instructions are also writing to the register file. Thus, one result can be read from DySER to be written into the register file per cycle.

Input Interface: The processor sends data from the register file to an input port. The data is buffered in the queue at that port until the DySER block is ready to consume the data. If the processor attempts to send data to a full port, the pipeline is stalled until the DySER block frees up space at the port.

Output Interface: Output values from DySER are held at the output ports in the queues until a dyser_recv instruction is executed to read the values out. The credit-based flow control guarantees that results will be generated in the order inserted into the DySER block, and results will not be overwritten by more recent invocations before being read out. If the result data has not yet reached the output port, the pipeline is stalled.

4. Results

The first step in determining OpenSPLySER's success is to verify that the changes to the pipeline do not interfere with normal processor operation. Because our main goal was to integrate DySER into an existing pipeline, functionality first takes precedence. After integrating OpenSPARC with the DySER modules, we ran all of the regression tests that came with it to ensure they all complete successfully. To verify DySER's correctness, we wrote a DySER assembler to construct binaries containing OpenSPARC and DySER instructions compatible with our OpenSPLySER processor. This enabled us to able to verify that the correct data was sent into the DySER input queues, the configuration bits routed data correctly and the correct data was read from the DySER output queues. Pipeline stalls and interrupts were also verified to execute correctly. Context switches were not investigated, as DySER's internal configuration state does not currently support context saves. Once functionality was verified, we then considered DySER's potential impact on performance.

A DySER block contains WxH functional units, allowing up to WxH operations to be performed

Signal Name	Description
Inputs	
dyser_config_enable	Signals DySER to take incoming configuration data. Enabled on a dyser_init
dyser_send_enable0 *dyser_send_enable1*	Signals DySER to receive data incoming from register file (RF) specified by RS1(2). High on dyser_send. dyser_send_enable1 is only used when sending two source registers
dyser_receive_enable0 *dyser_receive_enable1*	Signals DySER to send results from DySER ports specified by dyser_receiveport0(1) to register specified by dyser_reg_dest. High on dyser_recv. *OpenSPARC only allows 1 write per cycle, so dyser_receive_enable1 is always low*
dyser_sendport0[4:0] *dyser_sendport1[4:0]*	Address that specifies DySER input ports where data from register file goes. dyser_sendport1 is only used when two register sources are specified by dyser_send instruction
dyser_receiveport0[4:0] *dyser_receiveport1[4:0]*	Address that specifies port on DySER that writes to the register file. *Currently, OpenSPARC allows 1 write per cycle, so dyser_receiveport1 is always low*
dyser_commit	Signals DySER block to commit. Enabled on dyser_commit instruction
dyser_write	Register file write enable for dyser_recv instructions. Enabled on dyser_recv instruction.
dyser_mux_select	Selects DySER output signals from muxes in execute stage, which include dyser_write, dyser_data_out and dyser_reg_dest.
dyser_reg_dest[4:0]	Register destination for dyser_data_out on a dyser_recv
send_data_r0[63:0] *send_data_r1[63:0]*	Data from the register file sent to DySER on a dyser_send.
dyser_clk	DySER's clock based on global clock
dyser_rst	DySER's reset based on global reset
Outputs	
dyser_data_out[63:0]	Output data from DySER block to be written into a register
dyser_stall	Request from DySER to stall pipeline. dyser_stall is routed to the fetch control logic unit where it is handled by stall request logic

Table2. The microarchitectural interface between DySER and OpenSPARC.

concurrently. A DySER block kernel f computes Y[1..M] = f(X[1..N]), where f has N inputs and M outputs. We use an 8x8 DySER block which supports either 2 inputs or 1 output per cycle (dyser_send and dyser_recv, respectively). Thus, a particular kernel that requires N inputs and M outputs will require N/2 + M cycles to send in a set of inputs and receive the set of outputs. A kernel performing K operations (K <= WxH) can obtain speedup at most S = K / (N/2+M), if the kernel is executed repeatedly in a pipelined fashion. To test this, we performed a sum-reduction kernel over an array of values. This kernel takes 16 inputs and produces 1 output performing 15 additions, allowing theoretical speedup of S = 15 / (16/2 + 1) = 1.67. Using simulation, summing an array of size 4096 resulted in a speedup of 1.57 (95% of optimal) compared to usingasequenceof4095 SPARC additions. Larger data sets can better amortize the overhead of configuring DySER.

5. Future Work

A number of future directions have been identified to improve the design. One source of overhead is the configuration of DySER blocks. The configuration time is amortized over the duration of the phase, but opportunities to reduce configuration time exist. One solution is to implement two DySER blocks and perform double buffering, where one DySER block is in use while a second is being configured. An alternate solution is

to create copies of the configuration register inside a DySER block. With this design, the DySER block could be pre-configured multiple times, and the stored configurations swapped between very quickly.

Allowing*dyser_init*s to load configuration information directly from memory, instead of being instruction encoded, is expected to be a better approach as well. This would provide the opportunity for a more flexible approach to configuration, such as designating configuration information to be sent to only a subset of DySER blocks, as well as providing larger effective bandwidth. Considering research is already underway for scheduling multiple smaller comp-slices at once, this strategy would complement it well.

In our current implementation, at most two values can be sent to or one value read from a DySER block each cycle. The DySER interface scales well, so this is a limitation with OpenSPARC T1, not DySER. Integrating a DySER block into a multi-issue and/or out-of-order processor can improve the use efficiency of a DySER block by improving raw operand bandwidth and effective operand bandwidth limit through dependency reduction and memory disambiguation. A full characterization of the performance in the OpenSPARC pipeline would still be helpful in evaluation, with an FPGA implementation looking to be the most feasible option for this. This would allow us to run larger programs utilizing DySER in the native compiler-dictated environment, something lacking from current RTL simulations, as well as accounting for cache-miss impact on large run-times.

6. Related work

Similar architectures, such as VEAL[5], attempt to address the common problem of hardware accelerators by focusing on overcoming the need for binary compatibility. Due to DySER's requirement of special compilation, it ends up less portable than would otherwise be. Instead, VEAL executes the processor's baseline instruction set, without the need for extension. VEAL's advantage lies in its ability to accelerate inner loops, while DySER has similar capabilities in addition to the ability of extracting inherent parallelism out of block structures that VEAL cannot.

On the other end of the spectrum, architectures like Wavescalar[6] are almost entirely composed of circuit-switched ALU "tiles" forming a grid, similar to DySER. Between the ALUs are sets of small data caches, store buffers and switch controls for sending data around the network. Wavescalar is an example of a dataflow architecture, an approach that greatly differs from the typical realization of linear control-flow structured operation, an example being MIPS as described in Patterson and Hennessy [7]. In Wavescalar, results are immediately sent through a network to become an operand to any ALU that needs it, whereas typical pipelines use centralized register-files to organize operand-fetch & write for each operation. As a result, Wavescalar differs greatly on a high-level from DySER. DySER is a hardware accelerator meant to complement the existing (super-)scalar pipeline, whereas Wavescalar itself replaces the entire pipeline and memory structure, eschewing the linear Von Neumann PC model.

7. Conclusions

The DySER model of execution has been outlined, with details of the DySER hardware block constituting a grid of circuit-switched functional units provided. An overview of the ISA extensions and changes made for integration into the OpenSPARC T1 pipeline has been detailed as well. Application of the DySER model to an in-order pipeline has been proven possible, with DySER producing functionally correct results during chip-level RTL simulations. The peak-performance results for a sum-reduce kernel in these simulations corroborate with the theoretical expected for our particular DySER configuration. As a whole, these results show promise for a more complete implementation of the original ideas outline in [2], in addition to those mentioned above in future work.

Integrating DySER into the OpenSPARC pipeline presented some unique challenges, considering the sheer scale of the existing infrastructure. Luckily, the acts of understanding and modifying the simulation environment, making changes to the

pipeline RTL and additions to the DySER RTL were able to be done in a relatively parallel fashion. In retrospect, the maturity of the OpenSPARC offerings is a blessing and a bane. Regression environments are well defined, allowing minor micro-architectural changes to proceed relatively easy, however ISA-level changes are rather complex, as assembler provisions are not directly provided and workarounds are necessary. The multi-threaded philosophy of the core presented additional complications, as single-threaded operation was a late addition, leaving many ramifications in the pipeline's approach to flushes and stalling. Actively making design decisions early in the process favoring a simpler implementation approach proved well, considering this.

Acknowledgments

Many thanks to the great team at Vertical Research Group including Chen-han Ho, Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. A special thanks to Hennessey and Patterson for setting the foundation for computer architecture [8].

References

[1] K. Jeong, A. Kahng, A power-constrained MPU roadmap for the International Technology Roadmap for Semiconductors (ITRS), *SoC Design Conference (ISOCC), 2009 International*, pp. 49-52, Nov 2009.

[2] C. Ho, V. Govindaraju. Energy Efficient Computing With Dynamically Synthesized Datapaths. Vertical Research Group. To appear.

[3] C. Ho. Dynamically Synthesized Execution Resources (DySER) Design Specification. Vertical Research Group. To appear.

[4] OpenSPARC T1 Microarchitecture Specification. Sun Microsystems, Inc. 2006.

[5] N. Clark, A. Hormati, and S. Mahlke. Veal: Virtualized execution accelerator for loops. In ISCA '08, pages 389–400, 2008.

[6] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In ISCA '03, pages 291–302.

[7] D. Patterson and J. Hennessy. Computer Organization and Design. Morgan Kaufman Publisher 4th Edition, 2008

[8] J. Hennessy and D. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, Inc, 1996.

A. Appendix

A.1 Changes to OpenSPARC pipeline

In order to facilitate correct DySER operation, the following changes to the OpenSPARC architecture were applied:

sparc_ifu.v:

- Routed dyser_stall from execution stage to fetch control logic
- Routed all DySER signals from the output of decode to execute

sparc_ifu_dec.v:

- Used "Implementation Dependant Instruction 2" to implement all DySER instructions
    - Removed "impdep2" from illegal opcode logic
- Added decode logic to set DySER signals correctly which are pipelined to execute stage
    - DySER_Init, Store, Receive and Commit signals are set (detailed below in 3.1.2)
- Compslice extraction from DySER_Init
- Added validation logic to enable RS1 and RS2 reads for DySER_Sends

sparc_ifu_flc.v:

- Added "DySER stall request" to the rest of the stall request logic

sparc_ifu_thrcmpl.v:

- Added dyser_complete signal to the other long latency instruction complete signals

sparc_exu.v:

- Pipelined all DySER signals coming from decode
- Added instance of DySER module to execute stage
- Stall logic to disable DySER inputs on a global Stall

dyser_block.v:

- Dyser_block module created as a wrapper between SPARC execution stage and DySER

sparc_exu_ecl.v:

- Added mux for choosing either a DySER_Receive's register destination or the normal destination register
- Routed DySER register write to writeback logic

sparc_exu_ecl_wb.v:

- Added mux for choosing either DySER register write enable or normal write enable

sparc_exu_byp.v:

- Added mux for choosing DySER data out or ALU output

sparc.v:

- Routed all DySER signals from 'fetch-thread select-decode' to 'execute'

A.2 Changes to DySER

Flip-Flop Stage (ff_stage.v):

- Improved the resiliency of the stage machine
- Added support for multi-cycle functional units

Functional Unit (functional_unit.v):

- Functional unit was not properly capturing both inputs when they arrived at different times
- Added a Flip-Flop stage specific to functional units to manage this (fu_stage.v)

Switch (switch.v):

- Added configuration registers to each switch
- Configuration data is pipelined through the switches using a static "configure mode" path along the existing switch data network

Core (core.v):

- Fixed incorrect connections between the Edge fabric and Tile fabric
- Added a "configure mode" to the Core which uses the existing switch network for streaming configuration data to all Tiles in the DySER block

Input Bridge and Output Bridge (input_bridge.v and output_bridge.v):

- Full redesign. Added 4 element FIFO's at every port
- Control logic to interface between the processor and DySER Core (core.v)

DySER (dyser.v):

- Designed the interface between OpenSPARC and DySER based on various limitations
- Combined the components of DySER to interface properly with the OpenSPARC processor.

Testbenches:

- A variety of testbenches have been created to validate the functionality of the components of DySER. The most important being dyser_tb, which simulates a processor interfacing with a DySER block.
- All testbenches reside in the $DYS_SVN_ROOT/dyser/testbenches/ directory.

A.3 Environment Setup

In order to run conduct OpenSPLySER Assembly & Regressions/Simulation, the following needs to done to correctly configure the environment. Environment variables to the following need to be configured in the main script sourced from (found in the ./opensparc/OpenSPARCT1.bash):

- $DYS_SVN_ROOT := The absolute path to the OpenSPLySER SVN check-out (where ./dyser & ./opensparc are located).

- OR - (Normally set by the above)

- T1_ROOT := Path to the OpenSPARC base directory, used to hold anything required for the full implementation.
- DYS_ROOT := Path to the DySER verilog files.

Doing so points $PATH to the correct binaries & scripts used for simulation, as well other important directories/vars, including:

- $DV_ROOT := Directory holding all of the RTL, scripts/binaries, regression tests, etc.
    - ./design/ := All OpenSPARC RTL
        - ./sys/iop/ := Core-level verilog, anything relating to the pipeline, broken down into sub-modules/stages mostly
        - ./sys/iop/SPARC_Changes := Any new RTL that gets included in the design, or changed verilog to get grafted (used instead of) the original
    - ./tools/ := Simulation/regression tools
        - ./src/sims/sims,1.26 := Main front-end script called to handle building RTL into run-time models with VCS, calling Midas to assemble diagnostic assembly code into memory images, and simulating said models
- $MODEL_DIR := Directory to place/search for compiled VCS RTL models
- $REGR_DIR := Directory to place regressions/simulations in
- $DRMJOBSCRATCHSPACE := Temporary scratch space to use when running simulations

At the current moment, VCS is configured (for use in the CS department's environment) as the verilog compiler/simulator. Changing to another version may require debugging, as the shared libraries tend to clash; OpenSPARC provides 32-bit PLI libraries for interfacing with the simulator by default, so re-compiling them will be necessary if the sims' run-time argument -vcs_build_args="-mode32" doesn't suffice to fix compilation.

A.4 Running Regressions

Since it is not currently possible to run a single assembler that includes OpenSPARC T1 plus the DySER extensions, running a normal regression diagnostic is required before integrating DySER instructions. An example of this procedure is as following:

1. Change to the $REGR_DIR

1. Run "sims -group=thread1_mini -graft_flist=$T1_ROOT/impl/design/sys/iop/SPARC_Changes/graft.flist -flist=$T1_ROOT/impl/design/sys/iop/SPARC_Changes/include.flist -sim_type=vcs -vcs_build_args="-mode32" -novera_build -novera_run -sys=core1 -regress_id=somedir -vcs_build_args=-PP -vcs_build_args=+v2k -config_rtl=VPD

2. The above will compile the verilog model and run the full mini-regression test suite for the single-thread version of the T1 core, with DySER integrated (remove the graft/flist options to do so for the original pipeline). In order to run only ONE diagnostic it may be possible to add "-alias=exu_alu:model_core1:thread1_mini:0" for running a simple alu diag, although this has not been tested, so some variation on the "-alias" argument may be required.

3. After the above is run, the results for each diagnostic run will appear in $REGR_DIR/somedir, including the diagnostic assembly file (diag.s) and file to re-run only that diagnostic (sim_command). These provide everything necessary to create a new diagnostic/simulation test that will work for our purposes.

A.5 Diag Assembly for DySER Instructions and Simulations

With the environment created from the above, it is now possible to create an assembly file containing DySER operations to be performed, which will be spliced into the diagnostic assembly file after being processed through the DySER assembler (dyser_asm.awk). The DySERassembler will convert any DySERoperations into nops and pass all other instructions through, so as to allows Midas, the OpenSPARC assembler/linker front-end, to produce a properly aligned binary memory image. This is required, since the actual binutils used (GNU AS, LD, and OBJDUMP) do not support DySER instructions, and adding support has not been possible yet. Using this procedure allows a perfectly valid memory image (aligned with our DySER instructions!) and MMU programming to be created with the existing tools, then some extra effort is done to lay the DySER ops over the shadow nops in the image.

A DySER assembly file is written as such:

- Normal GAS (GNU AS assembly) syntax is used for SPARC V9 (T1 implementation) instructions, which is to say, that anything non-DySER and normal should be handled correctly.
- DySER instructions attempt to follow the same syntax in the following format

dyser_init <Configuration>	Configuration is a 21-bit immediate, typically numbers should be expressed in hex, with '0x' as a prefix
dyser_send <reg>, <port>[, <reg>, <port>]	One or two DySER input ports can be sent to in an instruction. The ports are 5-bit immediates, expressed in either decimal or hex ('0x')
dyser_recv <port>, <reg>	Only one port can be received from, where port is 5-bits immediate, either decimal or hex ('0x')
dyser_comm	No operands required
<reg>	General purpose registers are specified in typical GAS SPARC syntax: • %g[0-7] • %l[0-7] • %i[0-7] • %o[0-7]

The procedure for generating the proper memory image is as follows:

2. Change directory to the regression output
   a. e.g. $REGR_DIR/somedir/exuexu_alu:model_core1:thread1_mini:0
   3. Create the DySER assembly file
4. Run dyser_asm.awk <dys_asm_file>
5. From the output, splice the adjusted assembly code into diag.s
6. Modify sim_command with the following arguements:
. -nobuild
   a. -asm_diag_path=$REGR_DIR/somedir/model_core1:thread1_mini:0
   b. -asm_diag_name=diag.s #Note: Remove the normal *.s reference in here
7. Run sim_command, which will create a new mem.image, diag.ev, and symbols.tbl
8. Run dyser_asm.awk <dys_asm_file> again

9. Splice the differences in mem.image with the memory locations produced
   . This is rather tedious at the moment, this procedure will be simplified sometime in the near future
      a. For the moment being, it's best to have the DySER code go into the beginning of the "main" section in the diagnostic assembly file, as that can be found at the "MAIN" section tag seen in mem.image.
10. Copy mem.image, diag.s, diag.ev, and symbols.tbl to something like dys_test.*, keeping the same extension names.
11. Run sim_command with the changes
   . Remove the (6.c) addition, and add the following
      a. -image_diag_path=$REGR_DIR/dummy_dyser_1/exu_alu:model_core1:thread1_mini:0/
      b. -image_diag_name=dys_test.image
12. View the results
   . Use "dve" to load the vcdplus.vpd file for waveform viewing
a. "procvlog ./sim.log" gives a log of architectural changes

A.6 Presentation Slides

Agenda

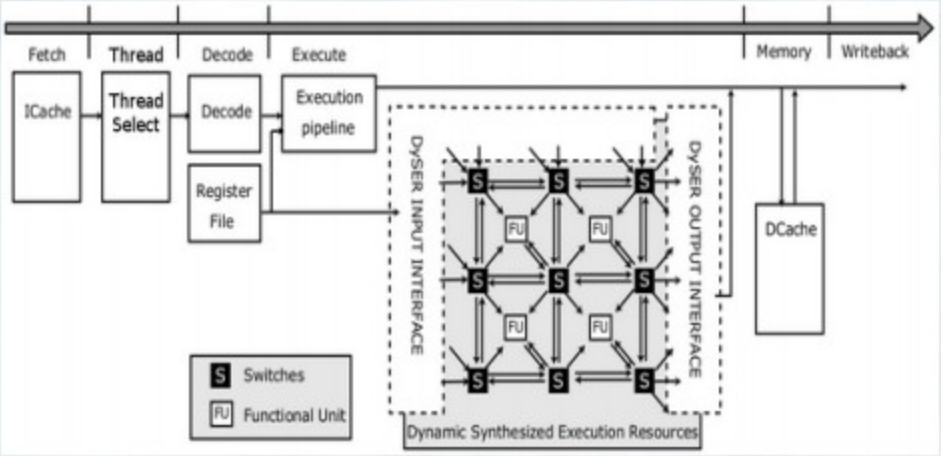
1. OpenSPLySER Design Overview
2. DySER Review
3. DySER Interface Design
4. OpenSPLySER Integration

5. DySER Verilog Modifications
6. OpenSPARC Verilog Modifications
7. Verilog Simulation and Verification

8. FPGA Synthesis
9. OpenSPLySER Going Forward

1. OpenSPLySER Design Overview

- Changes to Thread Select, Decode, and Execute stages



1. OpenSPLySER Design Overview

dyser_init <21 bits config>

10	config	110111	config	000	config

dyser_send RS1, DI1 [, RS2, DI2]

10	DI1[4:0]	110111	RS1[4:0]	DI2[4:0]	V	001	RS2[4:0]

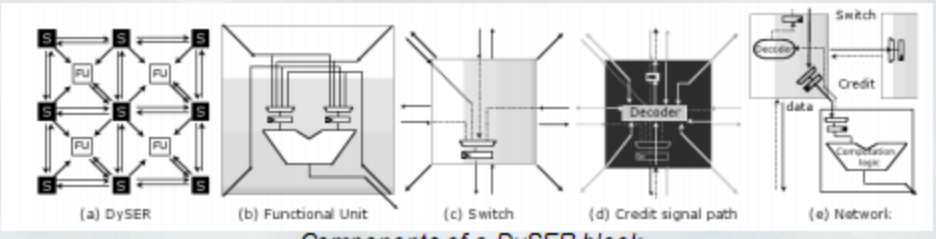
dyser_recv DO, RD

10	DO[4:0]	110111	RD[4:0]	unused	010	unused

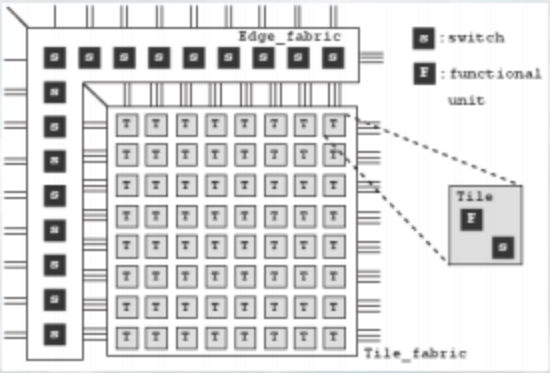
dyser_commit

10	unused	110111	unused	011	unused

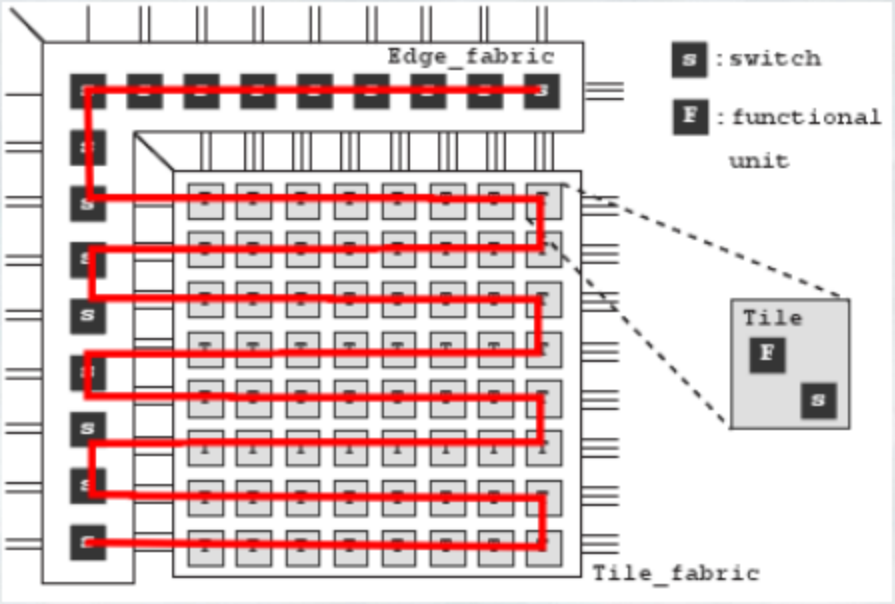
2. DySER Review



Components of a DySER block

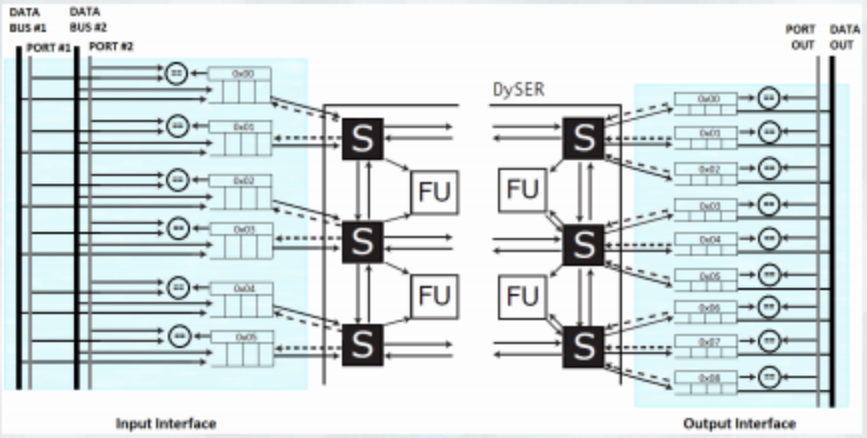


2. DySER Review



3. DySER Interface Design

- Ports: 2 input, 1 output; FIFO at each Port
- Uses DySER's "credits-flow" between DySER and FIFOs



3. DySER Interface Design

Signal	Description
INPUTS	
dyser_send	
send_data_r1[64]	Data from RS1
send_port_r1[5]	DySER input port for RS1
send_en1	Send enable for input Port 1
send_data_r2[64]	Data from RS2
send_port_r2[5]	DySER input port for RS2
send_en2	Send enable for input Port 2
dyser_recv	
recv_port_r1[5]	Output Port 1 to read from
recv_en1	Read enable for output Port 1
recv_port_r2[5]	Output Port 2 to read from
recv_en2	Read enable for output Port 2
dyser_init	
config_bits[21 or 30]	Configuration bits
config_en	Config mode enable
commit	dyser_commit enable
OUTPUTS	
send_stall	dyser_send must stall (input ports full)
recv_data_r1[64]	Data read from output Port 1
recv_data_r2[64]	Data read from output Port 2
recv_stall	dyser_recv must stall (data is not ready)

4. OpenSPLySER Integration

- DySER_inits are encoded into the instructions, comp slices decoded for DySER's config bits
- DySER_sends supply up to two operands to DySER ports specified in the instructions
- DySER_recvs allow one data result from a DySER port to be written back to RF at a time. DySER may stall pipeline until result is retrieved
- DySER_commit is for out of order execution which signals DySER to commit all of it's ready results back to memory/RF, it is unused in our in-order implementation

5. DySER Verilog Modifications

ff_stage.v: *credit-flow FSM with FF, mostly for switches*
multi-cycle FUs, done bit, different for EDGE switches

functional_unit.v: *heterogeneous functional unit block*
multi-cycle ability, better "done" notification, fu_stage.v

switch.v: *circuit switched*
local configuration registers, "config mode"

core.v: *edge_fabric and tile_fabric*
fixed misconnections between edge_fabric and tile_fabric
added "config mode" support

input_bridge.v and output_bridge.v: *interface between DySER and processor*
full redesign, 2 inputs, 2 outputs, 4-elt. FIFO at each port, processor stall

dyser.v: *top level design*
combined components

6. OpenSPARC Verilog Modifications

sparc_ifu.v: High level header file for fetch-thread select-decode stages
- Routed dyser_stall from execution stage to fetch control logic
- Routed all DySER signals from the output of decode to execute

sparc_ifu_dec.v: Decode stage
- Used "Implementation Dependant Instruction 2" to implement all DySER instructions
  - Removed "impdep2" from illegal opcode logic
- Added decode logic to set DySER signals correctly which are pipelined to execute stage
  - DySER_Init, Store, Receive and Commit signals are set (detailed below in 3.1.2)
- Compslice extraction from DySER_Init
- Added validation logic to enable RS1 and RS2 reads for DySER_Sends

sparc_ifu_fcl.v: Fetch Control Logic, controls stalls
- Added "DySER stall request" to the rest of the stall request logic

sparc_ifu_thrcmpl.v: Manages each thread's instruction completion
- Added dyser_complete signal to the other long latency instruction complete signals

sparc_exu.v: High level header file for execute stage
- Pipelined all DySER signals coming from decode
- Added instance of DySER module to execute stage
- Stall logic to disable DySER inputs on a global Stall

6. OpenSPARC Verilog Modifications

dyser_block.v: Dyser Wrapper File
- Dyser_block module created as a wrapper between SPARC execution stage and DySER

sparc_exu_ecl.v: Execute Control Logic
- Added mux for choosing either a DySER_Receive's register destination or the normal destination register
- Routed DySER register write to writeback logic

sparc_exu_ecl_wb.v: Writeback to RF
- Added mux for choosing either DySER register write enable or normal write enable

sparc_exu_byp.v: Bypass Logic for forwarding data to ALU or DySER
- Added mux for choosing DySER data out or ALU output

sparc.v: Highest Level Verilog file for SPARC core
- Routed all DySER signals from 'fetch-thread select-decode' to 'execute'

7. Verilog Simulation and Verification

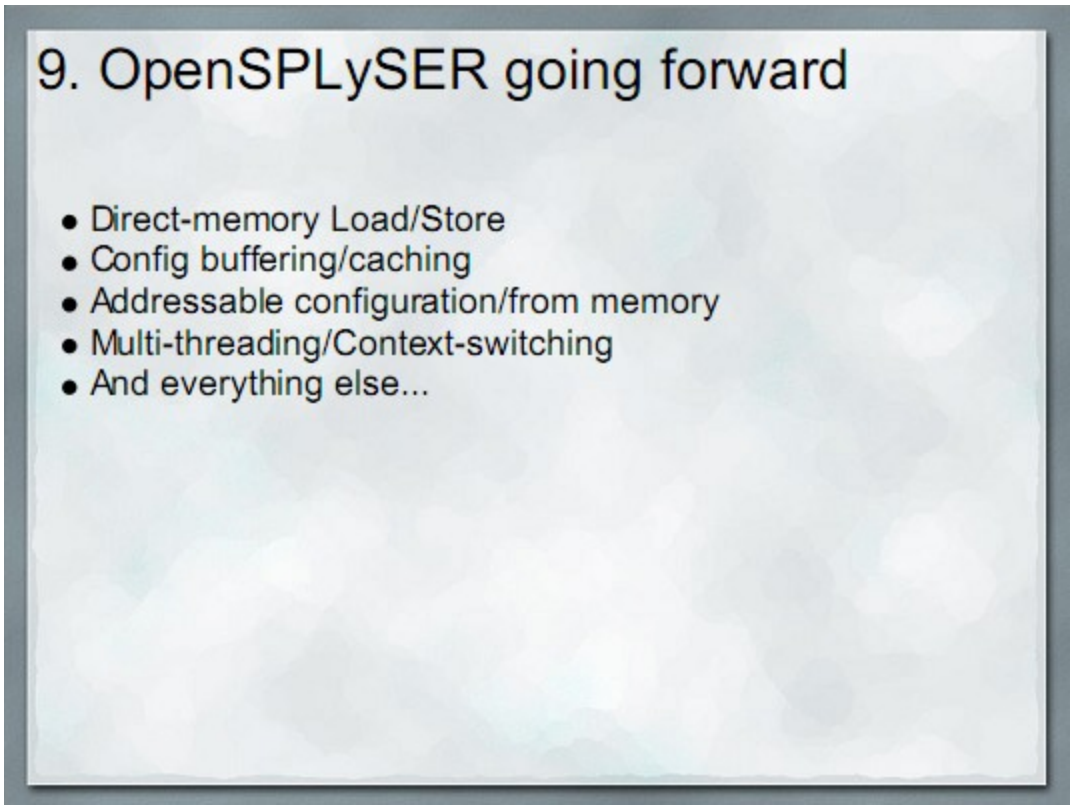
- All OpenSPARC regression tests pass
  - i.e. DySER doesn't break SPARCv9
- DySER instruction assembly
  - Procedure to get from DySER extended assembly to memory image is somewhat convoluted
  - For the time being, use AWK script to translate DySER portion of code into nops so normal assembler (GNU AS) can generate an aligned base memory image
  - Same script produces a memory image to overlay the base, manually spliced
  - Ideally this wouldn't be an issue if GAS can be extended, work in progress
    - Collaborate on this?

7. Verilog Simulation and Verification

- DySER instruction simulation
  - Uses the prior mentioned memory image as program, plus auto-generated MMU/eFuse programming
    - includes POR & basic MMU setup in code
  - simulations run from POR through to user-code
    - Cache, TLB, DRAM timing, etc. details modeled
  - No instruction-by-instruction verification
    - Instead rely on waveform viewing, or else ability to trap to "bad_trap" on unexpected register values
    - Simulation trace file produced, which can be parsed for changes to architected state
- Basic DySER tests pass and work in the pipeline
  - Sum-Reduce

8. FPGA Synthesis

- Little analysis
  - How does stand-alone DySER synthesize?
- Suspect synthesis will require 2 FPGA's (or Virtex-6?)
  - Wire Mayhem


9. OpenSPLySER going forward
• Direct-memory Load/Store
• Config buffering/caching
• Addressable configuration/from memory
• Multi-threading/Context-switching
• And everything else...