The Harmonica Processor Core

Yanjun Chen Si Li William Song Chad Kersey

September 3, 2013

1 Implementation

Harmonica has been implemented using CHDL, a C++ digital hardware design library. This choice allows Harmonica to be instantiated and simulated using only GCC, a widely-available and free compiler.

2 Microarchitecture

Harmonica is an in-order issue, pipelined implementation of a significant subset of the Harp user mode instruction set architectures supporting out-of-order instruction completion. It is a parameterized implementation with variable width (in both SIMT lanes andbits), variable register file dimensions, and interchangeable functional units.

The Harmonica pipeline can be divided into four phases:

- Instruction Fetch
- Decode 1/Register file access
- Issue
- Execution
- Writeback

Fetch, Decode, Issue, and Writeback each require a single cycle. Execution time can vary depending on the functional unit design.

2.1 Instruction Identifiers (IIDs)

A small register, large enough to contain a number uniquely identifying each fetched instruction still within the pipeline, is contained in the fetch stage. This number, the Instruction Identifier (IID), uniquely describes each instruction in the pipeline. They are used in the writeback stage to ensure that only the most recently-issued instruction writing to a register will influence its value. This enables out-of-order completion of instructions as long as issue remains in-order.

Fetch, Register read, Issue, and Writeback each require a single cycle. Execution time can vary depending on the functional unit design.

2.2 Parameterizability

Architecture parameters, such as the number of registers and the width of the datapath, are all software configurable. At the top of `harmonica.cpp` a set of constants is defined, describing the instruction set parameters. Between these constants and the configurable set of functional units, Harmonica can be reconfigured for specific applications.

2.3 Hazard Detection and Bypass Network

No support was introduced for forwarding of values for dependent instructions. While this would improve the single-thread performance significantly, the Harmonica design is intended as the first stage in the development of a GPU-like architecture. Since fine-grained multithreading designs can benefit from the area, power, and design complexity advantages of not having a bypass network, none was provided. Instead, a basic pipeline stall insertion mechanism was used.

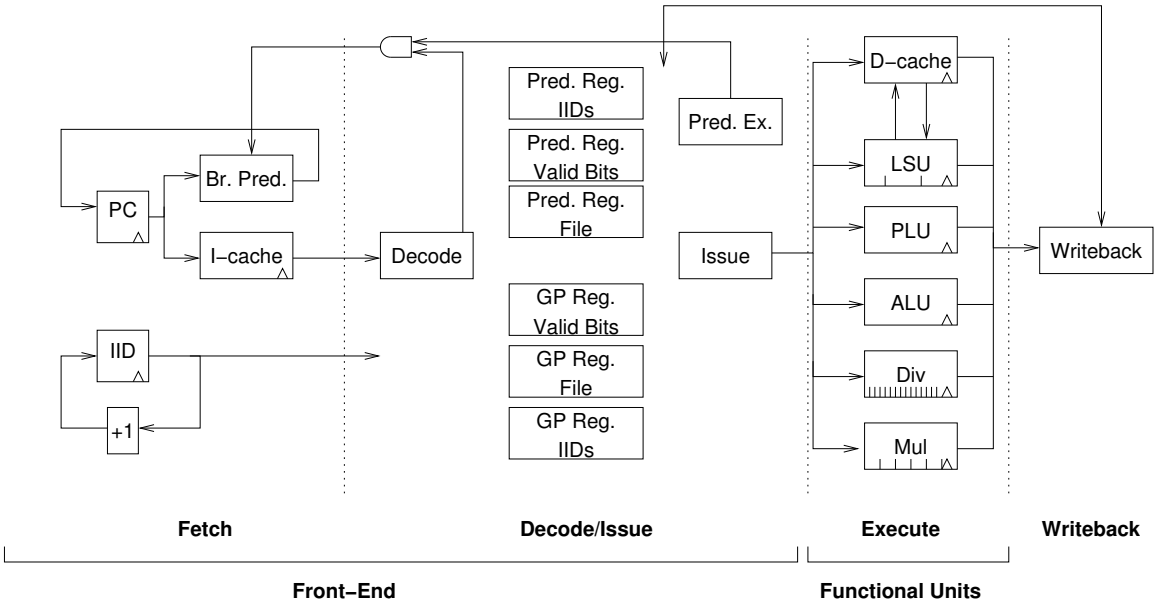

Figure 1: Block diagram of the Harmonica microarchitecture. The set of functiona l units is arbitrary.

SIMT Lanes	Bits	Registers	Total Nodes	Critical Path
2	16	2	20 404	94
8	16	2	66 836	94
32	16	2	252 612	94
1	32	8	31 869	100
2	32	8	54 247	100
4	32	8	141 603	100
8	32	8	185 561	100
16	32	8	362 001	100
32	32	8	712 288	100
1	64	8	75 638	106
4	64	8	252 733	106

Table 1: Size of netlist and critical path for different configurations (serial divider, no floating point unit).

3 Functional Units

The basic functional units are implemented in `funcunit.h`. Additional functional units have been implemented as external modules written in Verilog. Inputs to the functional units are of type `fuInput` and outputs are of type `fuOutput`. The set of opcodes provided by a functional unit are provided (as a vector of `unsigned int`s by a member function called `get_opcodes()`.

3.1 Arithmetic Logic Unit

`BasicAlu` is a simple ALU implementation, supporting all common integer arithmetic and logic functions involving registers and immediates which can be handled in a single cycle. Of the integer operations, division and modulus are the only ones not supported, although multiplication can optionally be handled by a separate functional unit.

3.2 Predicate Logic Unit

`PredLu` performs all predicate operations, including predicate logic and conversions from register vales to predicate values.

3.3 Integer Divider

Division and modulus are handled by a separate functional unit. Both a practical shift register based serial divider and an enormous, power-hungry pipelined divider design were implemented. Given the rarity of divides in modern applications, the choice is obvious.

3.3.1 Serial Divider

The serial divider performs divisions one bit at a time using a shift register. Results are made available after 32 clock cycles regardless of the size of the divisor or dividend.

3.3.2 Pipelined Divider

The pipelined divider, the result of a naive retiming algorithm that does not consider the cost of registers, is a 173 kilonode monstrosity of a circuit implementing a 7-stage pipelined divison.

3.4 Floating Point Unit

A separate FPU was implemented to handle floating point instructions. In the HARP ISA, these operate on the usual register set.

3.5 Verilog FPU

A Verilog FPU fully supporting all floating point instructions available in the HARP ISA was implemented. Unfortunately, it was not integrated with the mainline HARP codebase or FPGA tested.

3.6 Pipelined FPU

The same retiming algorithm used to create the useless pipelined divider has also been used to implement a subset of the floating point instructions: `fneg`, `fadd`, `fsub`, `fmul`, `itof`, and `ftoi`. Only the relatively difficult `fdiv` was excluded. This created a three-stage pipelined floating point unit with 79 2-input NAND gates worth of delay in each stage.

3.7 Load/Store Unit

3.8 SramLsu

`SramLsu` is a very simple unit with a single cycle latency, storing all data in a (configurable size, but usually tiny) synchronous SRAM.

3.9 Load/Store Unit With Cache Interface

The load store unit accepts memory requests in order and may issue them to the memory system out of order and write loaded memory value back to register out of order. It is parameterized and can handle variable number of lanes, queue size, address and data sizes. The address is organized as follows from the least significant to the most significant bits: byte offset within a word, lane offset.

The load store unit can be divided into four components:

- Front end
- Load queue
- Store queue
- Forwarding and Hazard Logic
- Commit Logic

The front end, load queue and store queue requires a single cycle. There also exists bypass logic for the front end and load queue in the memory bound direction.

3.9.1 Front end

The front end intercepts multiple streams of memory requests per memory instruction. It attempts to coalesce into fewer memory requests to the memory system. It first loads the incoming memory requests into temporary registers called feAddress. A non-predicated memory address is selected for insertion in to a load or store queue, based on the request type. This memory address is then compared against the upper bits of all other valid feAddresses for coalescing. Lane offsets are masked off for this comparison. Matching addresses will have their lane offset inserted into the same entry of the load or store queue. Additionally, matching addresses will have their valid flags cleared. The queue entry that receives the first address insertion of a memory instruction has its leader flag set, to be used for receiving data from the memory system. All non-leader queue entries will store the queue id of the leader. In the case of a fully coalesced access, all feAddresses will be cleared after one cycle and only one queue entry is used, it will be designated as a leader.

To avoid the one cycle latency in storing incoming addresses into a register, the selection and coalescing logic is replicated for the input addresses. This way at least one address would be issued to the load or store queue without delay. In a fully coalesced access, the feAddress registers are bypassed entirely.

3.9.2 Load Queue

The load queue does not guarantee in-order memory issue or in-order write-back. It finds a free entry and stores the next available memory address and its offsets in that entry. This is acheived by using a priority encoder on a bit vector of free flags. The result is then decoded to drive write enable on the selected free entry. Selecting an entry to send to the memory interface uses a similar mechanism with the pending flag, which is set on insertion and cleared when selected. A unique loaded flag indicates the status of the data register particular to a lane.

When the data returns from the memory system, it is sent to the leader entry of the load queue. If the returning queue id is to a non-leader queue entry, it will find the leader and store the data to the leader queue entry. Each non-leader queue entry stores a pointer to its leader. When all valid lanes are loaded, the entry can be written back to architectural registers.

To avoid the one cycle latency to the memory interface, a bypass logic is used to route the insertion address to the memory interface. No bypass mechanism for the write-back route is implemented

3.9.3 Store Queue

The store queue must commit stores in order, so a head and tail pointer is used to keep track of insertion and ejection positions. Each store queue entry also keeps track of one dependent load entry. Multiple dependent load entries are not supported at the moment. All offsets within a store queue are assumed to be in consecutive order. All lanes of the same instruction are assumed to store to consecutive locations within a l*word boundary. No bypass mechanism is used with this structure.

3.9.4 Fowarding and Hazard Logic

This logic examines incoming requests and checks for matching addresses in both load and store queues. If the request is a store, matching stores will be invalidated and matching loads are recorded so the store must wait on it. The WAR detection is conservative, where only the address is used in the comparison, and the offsets are ignored. If the request is a load, no RAR detection is made, but RAW detection is implemented, and the appropriate lanes will forward

data from the store queue to the load queue.

There are three possible outputs to the memory interface: load queue, store queue, and load queue bypass. If the load queue is empty, with no pending requests, the load queue bypass is selected. The by pass is disabled if load queue has pending requests. The store queue is selected if there are no pending load requests or the store queue is full.

3.9.5 Commit Logic

The commit logic selects a load queue entry with all data registers loaded. It is up to the write back logic outside of the load store unit to decide whether the result will be written to its destination register based on the instruction identifier.

4 Next Program Counter Logic

While multithreading obviates branch prediction for the same reasons as the bypass network, a branch predictor was implemented.

The branch prediction unit resolves the target of jump instructions based on the program counter (PC) before fetched instructions are decoded. A correct prediction removes stall cycles to resolve predicated instructions or the target addresses of jump instructions. The branch prediction unit includes a conventional gshare predictor and branch target buffer (BTB). The branch predictor unit is parameterizable with gshare and BTB table sizes and counter length.

4.1 Branch Prediction

The gshare predictor is comprised of a global history register (GHR) and branch pattern table (BPT). The GHR is a shift register that marks a pattern of jump and predicated instructions. The GHR pattern and PC are used to index BPT and BTB entries. A typical XOR function is used to hash GHR and PC. For instance, the lowest two bits are discarded for a 32-bit PC, and upper segments are multi-folded (i.e., XOR) to the length of GHR. Thus, the hashing function is a series of XOR operations between PC segments and GHR pattern.

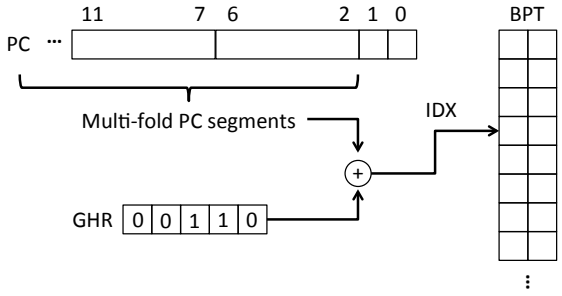

Figure 2: A gshare predictor example with 32-bit PC, 5-entry GHR, and 32-entry 2-bit saturating counters.

The BPT consists of saturating counters. The BPT entries are initially reset to 01 that means weakly not-taken. The hashed GHR pattern and PC, noted index (IDX) in Figure **??**, is used to read a BPT entry. The highest bit of a counter indicates the jump result; 1 is taken, and 0 is not-taken. When predicated or jumps instructions are resolved, the BPT counters are updated. Since the last prediction was made by previous GHR pattern and PC, the branch prediction unit maintains the last index and use it to update the right entry that was used to make a prediction.

4.2 Branch Target Buffer

The same index from the GHR and PC is used to access a BTB entry. The BTB has tag and target address fields. If the tag matches with current PC, then the BTB treats the target address is valid. A taken prediction is made only when a BPT counter indicates taken jump and BTB target address is valid. Coalescing the BPT and BTB results removes all false predictions made for non-jump or non-predicated instructions. BTB entries are updated with jump and predicated-taken instructions.

5 Example Programs

Five simple test programs were implemented:

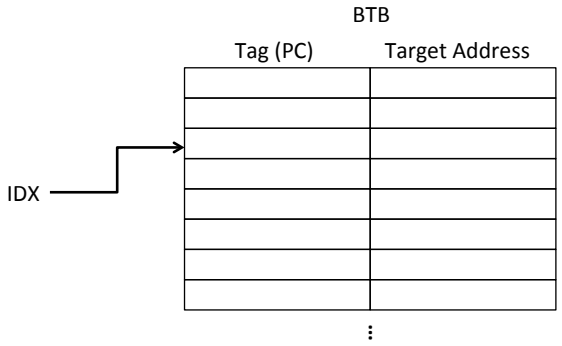

Figure 3: A direct-mapped branch target buffer composed of tag (PC) and target address.

5.1 `sum_regs.s`

The initial test program sums sequential numbers in registers, without using memory. The results can be seen by examining register contents.

5.2 `sum_mem.s`

The second program sums the same set of numbers, but first stores them in RAM. This exercises the control flow and ALU like the first program, but also brings the (for a single lane) load/store unit into play.

5.3 `sum_simd.s`

The third program sums numbers stored in memory, but sums a different set of numbers in each SIMD lane. This program exercises the SIMT load/store unit as well as the ALU.

5.4 `hello.s`

No set of test programs would be complete without its own, "Hello, world." This is the HARP equivalent, the first of these programs to use the I/O subsystem and thus be useable on the FPGA board.

5.5 `sieve.s`

The Sieve of Eratosthanes is an ancient algorithm for finding prime numbers by process of elimination.

Program	Dyn. Inst.	Cycles	IPC
`sum_reg.s`	502	1509	0.333
`sum_mem.s`	1203	3612	0.333
`sum_simd.s`	27	84	0.321
`hello.s`	322	972	0.331
`sieve.s`	3866	12248	0.316

Table 2: Performance of applications as determined from RTL simulation.

This is the most complete benchmark. It exercises most the arithmetic operations, control flow and I/O.

6 Performance

The prototype scheduling logic has a critical path length of 100 NAND gates. Without technology mapping or placement/routing, it is difficult to say how this translates to delays in terms of FO4, but assuming each NAND gate has a delay on the order of ten picoseconds leads to a clock frequency on the order of one gigahertz.

Table **??** gives the core's performance for the test applications. As it happens, every one of these applications consists almost entirely of chains of dependent instructions, leading to quite poor performance. In an implementation with fine-grained multithreading, the idle cycles could be replaced with the execution of other threads, improving performance.