The Vanilla Architecture

VERSION 1.2

MOEIN KHAZRAEE
MICHAEL TAYLOR

FOR USE IN CSE 141 AND 141L.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

University of California, San Diego

JANUARY 2014
LA JOLLA, CA

1 Introduction

Vanilla is a simple ISA that has three goals in mind:

1. **Simplicity**: an ISA implementation should be simple to understand and be expressable with a few pages of System Verilog.
2. **Energy Efficiency**: the ISA encoding should be simple but also compact in size so that it minimizes fetch energy.
3. **Configurability**: the ISA encoding should have plenty of free space so that it may be extended.

2 Vanilla Core Instruction Set

An example of a hypothetical Vanilla instruction is show in Figure 1. We explain the aspects of the instruction below.

Architectural State. The Vanilla ISA has the following programmer-visible state:

- a instruction memory,
- a data memory,
- a 32-element 32-bit register file,
- a 32-element 32-bit constant pool,
- an incoming network channel for loading the constant pool and instruction memory, and
- a barrier network for synchronizing with other cores or a host processor.

Instruction Encoding. All instructions are 16-bit. The typical instruction has:

- **a 5-bit** $opcode$ **field** that specifes the operation to perform
- **a 5-bit** rd **field** that designates both the destination and the first source of the instruction, and finally

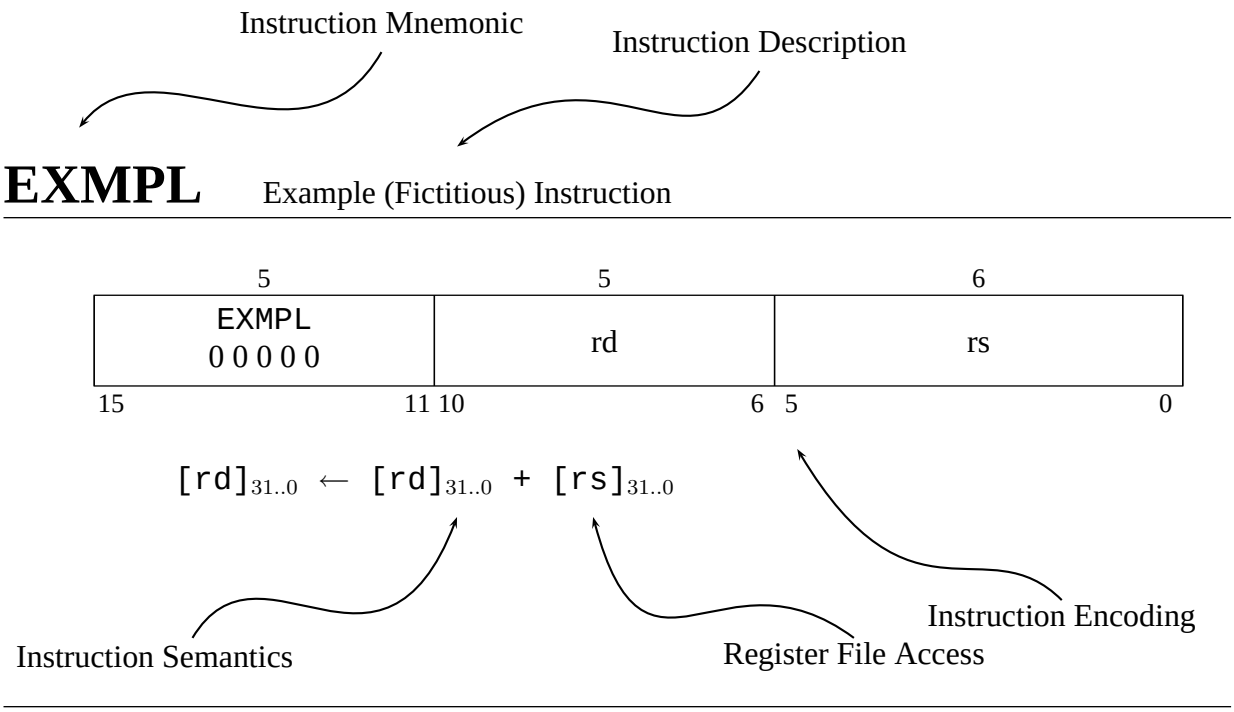


Figure 1: An example (fictitious) instruction in Vanilla.

- **a 6-bit rs field** that is typically a second source for the instruction, but can also be an offset for a PC-relative branch. When the field specifies a second source, it can be either from the register file, if the top bit of the rs field is clear, or from the constant pool, if the top bit of the rs field is set.

Data address sizes and Data value widths. The address sizes and widths for data are 32-bits; however byte-level loads and stores are also supported by the instruction set. Word accesses must be aligned. Writes to memory locations above physical memory are ignored and may be used to perform I/O.

Instruction address sizes. The PC points to words, not bytes, in instruction memory, and is 32-bits. The instruction memory may contain fewer than 2^{32} words; in this case, the top bits are ignored. The data and instruction memory addresses spaces are not shared; this is a true Harvard Architecture.

Opcodes. A map of instruction opcodes is shown in Table 1. Currently, there is one special extension opcode, $SPEC1$, which uses the rd field to specify the actual operation. $SPEC1$ is shown in Table 2.

	ins[13 .. 11]							
ins[15..14]	000	001	010	011	100	101	110	111
00	ADDU	SUBU	SLLV	SRAV	SRLV	AND	OR	NOR
01	SLT	SLTU	MOV		SPEC1			
10	BEQZ	BNEQZ	BGTZ	BLTZ			JAL	JALR
11	LW	LBU	SW	SB	LG			

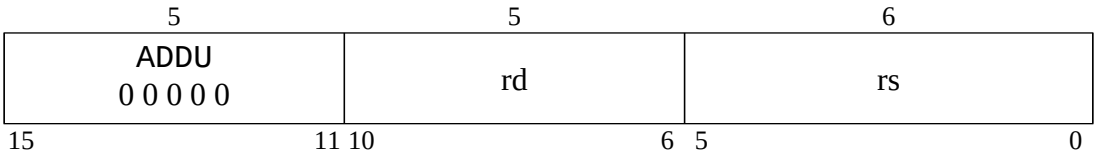
Table 1: Opcodes map

	ins[8 .. 6]							
ins[10..9]	000	001	010	011	100	101	110	111
00	WAIT							
01	SLEEP							
10	BAR							
11								

Table 2: SPEC1 opcode map

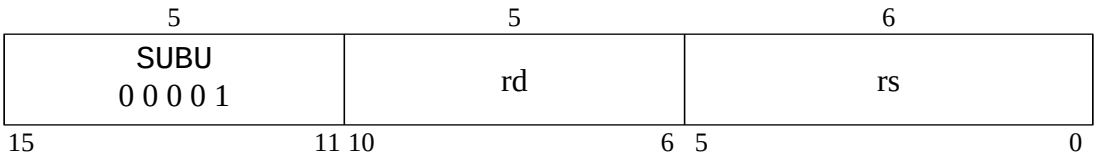
2.1 Arithmetic Operations

ADDU Add Unsigned



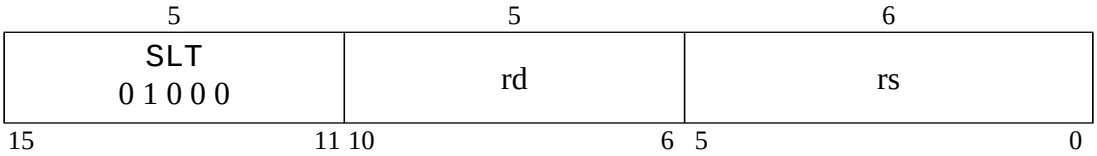
$$[rd]_{31..0} \leftarrow \{\ [rd]_{31..0} + [rs]_{31..0}\ \}_{31..0}$$

SUBU Subtract



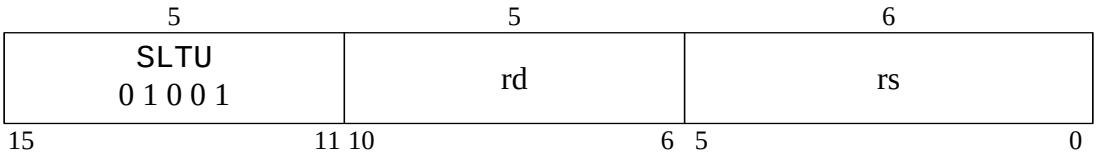
$$[rd]_{31..0} \leftarrow \{\ [rd]_{31..0} - [rs]_{31..0}\ \}_{31..0}$$

SLT Set Less Than Signed



$$[rd]_{00} \leftarrow ([rd]_{31..00} <_{signed} [rs]_{31..00})\ ?\ 1 : 0$$
$$[rd]_{31..01} \leftarrow 0$$

SLTU Set Less Than Unsigned



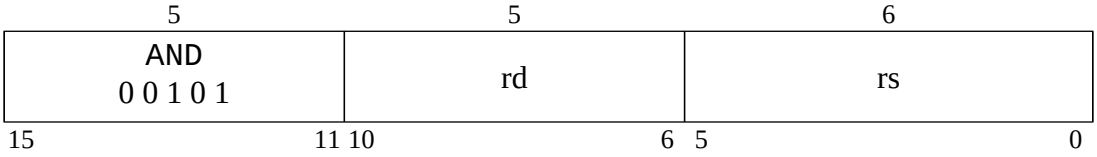
$[\texttt{rd}]_{00} \leftarrow ([\texttt{rd}]_{31..00} <_{unsigned} [\texttt{rs}]_{31..00})\ \texttt{?}\ \texttt{1}\ \texttt{:}\ \texttt{0}$

$[\texttt{rd}]_{31..01} \leftarrow \texttt{0}$

2.2 Logical Operations

AND

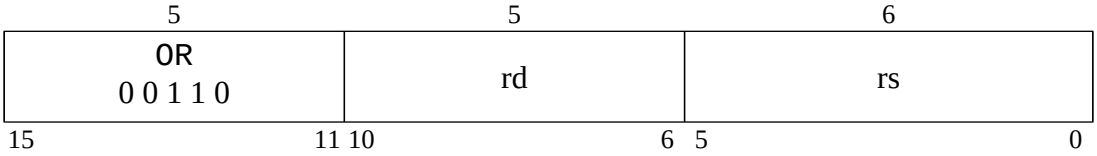
And Bitwise



$[rd]_{31..0} \leftarrow [rd]_{31..0}$ & $[rs]_{31..0}$

OR

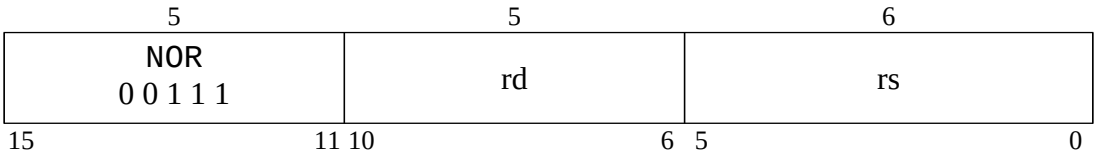
Or Bitwise



$[rd]_{31..0} \leftarrow [rd]_{31..0}$ | $[rs]_{31..0}$

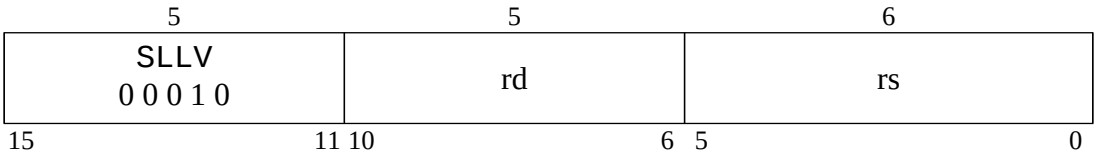
NOR

Nor Bitwise



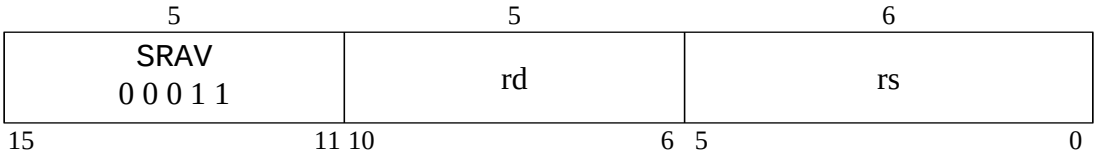
$[rd]_{31..0} \leftarrow$ ~($[rd]_{31..0}$ | $[rs]_{31..0}$)

SLLV Shift Left Logical Variable



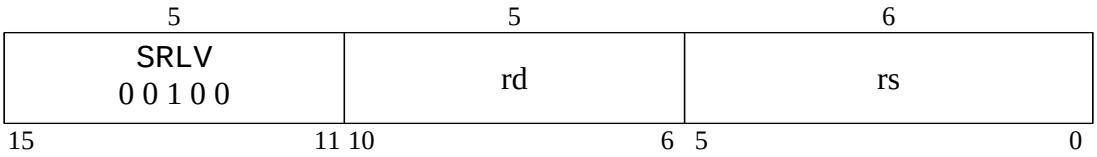

```
shift amount (sa) = [rs]4..0
[rd]31..sa ← [rd](31−sa)..0
[rd](sa−1)..0 ← 0
```

SRAV Shift Right Arithmetic Variable




```
shift amount (sa) = [rs]4..0
[rd](31−sa)..00 ← [rd]31..sa
[rd]31..(31−sa) ← [rd]31
```

SRLV Shift Right Logical Variable

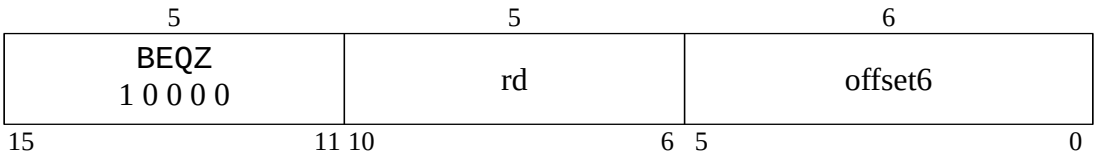



```
shift amount (sa) = [rs]4..0
[rd](31−sa)..00 ← [rd]31..sa
[rd]31..(32−sa) ← 0
```

2.3 Branch and Jump Operations

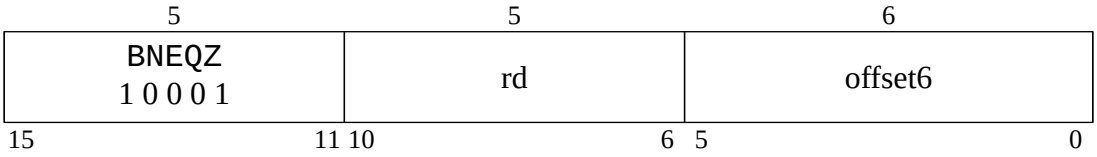
6 bit offset in the conditional branch instructions must be sign extended before being added to the current PC, which is shown as (*sign-extend* **offset6**).

BEQZ Branch if equal to zero



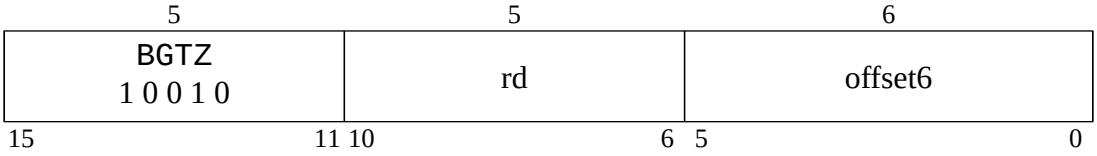

```
if ([rd]31..0==0)
    PC  ←  PC + (sign-extend offset6)
```

BNEQZ Branch if not equal to zero



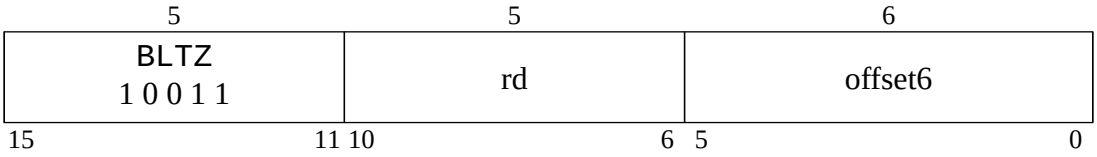

```
if ([rd]31..0!=0)
    PC  ← PC + (sign-extend offset6)
```

BGTZ Branch if greater than zero (signed)



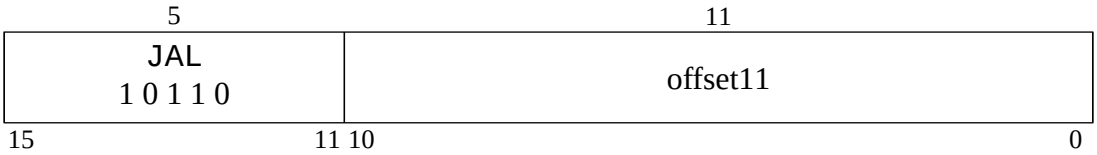

```
if ([rd]31..0>0)
    PC  ← PC + (sign-extend offset6)
```

BLTZ Branch if less than zero (signed)



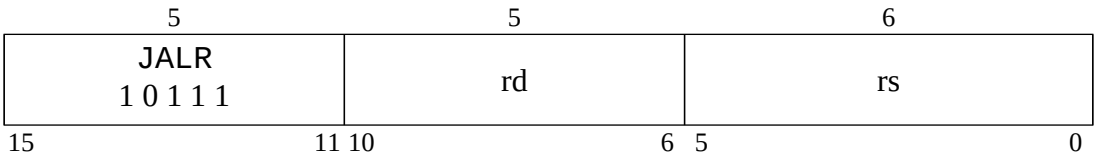

```
if ([rd]31..0<0)
    PC  ← PC + (sign-extend offset6)
```

JAL Jump and link




```
PC   ← (offset11 << 2)
r2   ← PC + 1
```

JALR Jump and link through Register

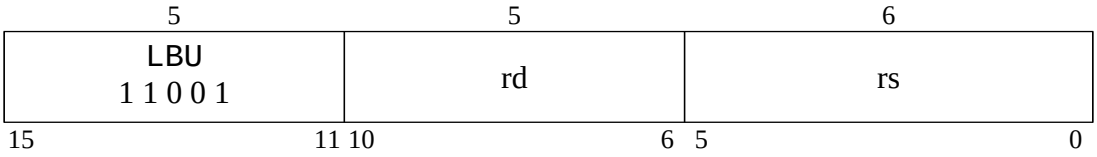



```
PC   ←  [rs]
[rd]  ← PC + 1
```

2.4 Memory Access Operations

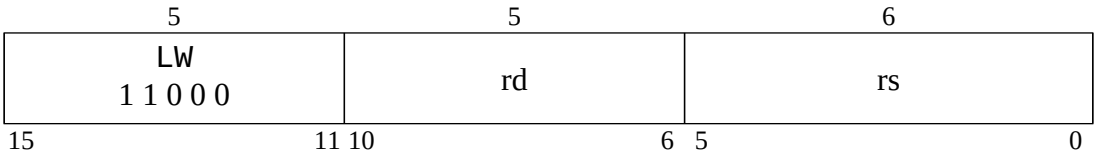
Mem8 and Mem32 refer to a byte or word request to the data memory.

LBU Load Byte Unsigned



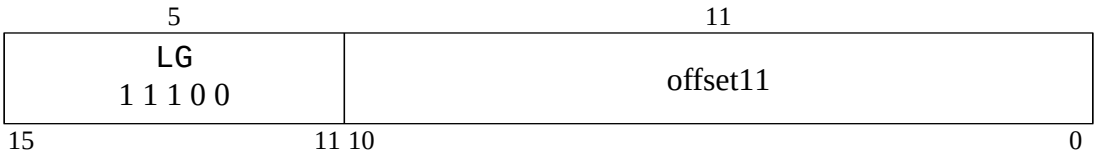
$[rd]_{7..0} \leftarrow Mem8[[rs]_{31..0}]$
$[rd]_{31..8} \leftarrow 0$

LW Load Word



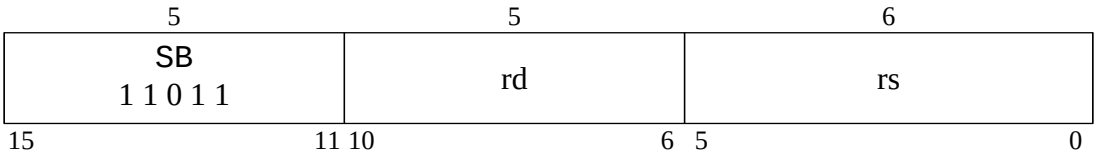
$[rd] \leftarrow Mem32[[rs]_{31..0}]$

LG Load Global



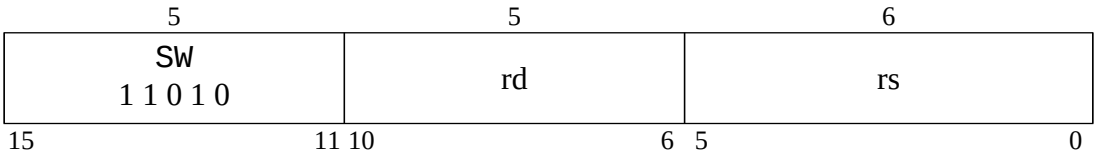
$r1 \leftarrow Mem32[offset11]$

SB Store Byte



`Mem8[[rd]]` ← `[rs]`$_{7..0}$

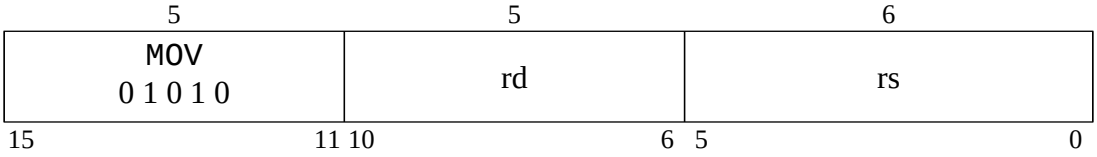
SW Store Word



`Mem32[[rd]]` ← `[rs]`

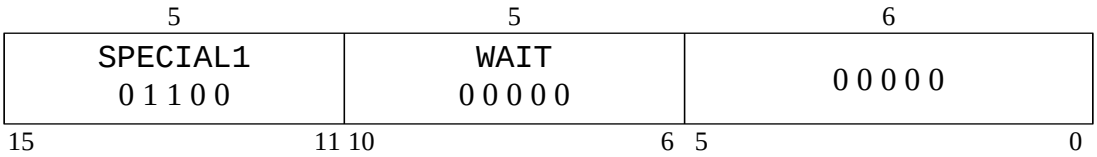
2.5 Featured Operations

MOV Move



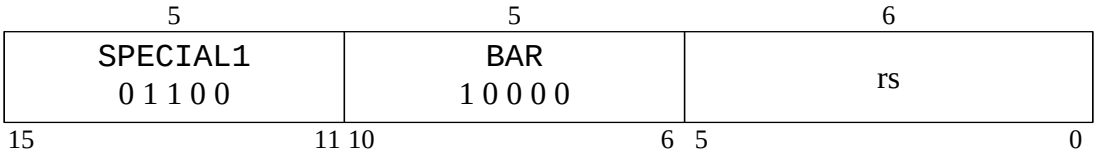
$[rd]_{31..0} \leftarrow [rs]_{31..0}$

WAIT End of Current Program



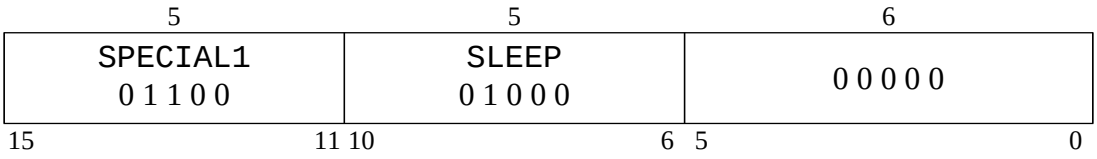
Core goes to the IDLE state and waits for new PC. (Formerly called “DONE”.)

BAR Set the barrier signal



$Barrier \leftarrow [rs]_{31..0}$

SLEEP Set the barrier signal




```
Barrier  ← all bits set to 1
In addition, core goes to the IDLE state and waits for new PC.
```

Register File. Register 0 is hardwired to the value zero.

Constant Pool. Instead of having immediate values baked into the instructions, instructions make use of the constant pool, which is logically a single-ported 32-bit, 32-value register file. The constant pool is typically initialized via the network before execution of a kernel (i.e. code) begins[1]. Because of the constant pool, there is no need for immediate instructions like ADDI, and the length of r_s and r_d fields are not equal, since rs may also refer to the constant pool. The use of the constant pool reduces code size.

2.6 External Interfaces

Each core contains three key interfaces to the outside world. The first is a data memory interface, which is used to perform loads and stores, as well as memory-mapped I/O. The second is a network interface, which allows instructions, data, and register values to be transmitted to a core. The third is a barrier interface, which allows the cores to synchronize with an outside controller. An example data memory interface is described

[1]Registers may also be initialized via the network, but this use is discouraged and is instead used for communication between a controller and the core.

in Section 4, which describes the single-cycle Vanilla core implementation. The barrier and network interface are described below. In the following sections, we describe linear barrier and network implementations, but other topologies or even hierarchies are easy to envision for scalability.

Barrier Hardware. Each core contains two registers; one is called the Barrier Mask and the other is the Barrier Bits. The width of these registers is the same and implementation dependent. Each bit corresponds to a logical barrier in the system. The Barrier Mask indicates which barriers a given core is currently participating in. The Barrier Bits indicate the value that the core is contributing to a barrier. The Barrier Bits and Barrier Mask registers are AND'd together before exiting the core in a single barrier bus. A linear barrier topology will OR together the individual bits of the barrier buses coming from each core. Typically a new Barrier value will be set by the network when it transmits a new PC to execute. This arrangement supports two key uses of barriers: Eureka barriers and Completion barriers.

Completion Barrier. A Completion barrier is used in codes for which cores need to synchronize with a controller. The barriers can also be used to orchestrate how cores make use of shared data memory. Execution will be divided into phases; in each phase, the address space is statically partitioned in the following way: 1) some addresses are read-only by all cores; 2) some addresses are read-write by only one core; and 3) some addresses require a lock to perform reads or writes. A barrier is necessary when execution transitions between phases that change the partitioning of memory addresses.

Execution proceeds as follows: First, the Barrier bit for the phase barrier is set by the controller, along with a PC to start executing at. Then execution of the phase begins. When a core reaches the end of the barrier, it clears its barrier bit. When the OR of all of the barrier bits for a single logical barrier coming out of the cores is zero; then we know

that all cores have reached the end of the code to execute for the phase. Typically, after a core has reached the end of its phase, it will execute a BAR instruction to clear/set a barrier bit, and then execute the WAIT instruction, which puts it in the IDLE state. When the OR'd barrier signal goes low, then all cores are in IDLE. The controller may then broadcast a new Barrier bit value and a new PC for the cores to resume at.

Eureka Barrier. A Eureka barrier is used for codes where many cores are running a search computation. When one of the cores finds the result, it asserts the Eureka signal, indicating to the outside world that the answer has been found. For Eureka barriers, the corresponding Barrier bit on all participating cores will be set to zero. When a core finds the answer, it will place an identifying value in a fixed mailbox location in data memory, and then set the Barrier bit to a one, and execute a wait instruction. Then the bit on the output barrier bus will be set, and the output of the OR will be a one, signaling to the controller that a Eureka has been found. The controller may then look at the mailbox location in data memory to learn more about the item that the core found. Cleanup for Eureka barriers can be difficult because asynchronously interrupting a core can lead memory in an unknown state. A clean way to do it is to have each core, periodically when at a "neat" place in execution, check a particular register value to see if it has been changed to zero, and perform cleanup. When it is done cleaning up, it will set a separate Completion barrier bit to a 0 and then execute a WAIT instruction. The controller then is looking for the Eureka bit. When it sees the Eureka bit, it broadcasts a register write to all of the cores. The cores will see the register value change, and then clear their Completion bit. When the Completion bit goes low, then all cores have cleaned up and a new PC can be broadcasted by the controller.

Network. All cores are connected by a network. A simple version of the network just attaches the cores in a tree or linear fashion, but a hierarchical network could also be used to save power. The network allows the controller to send a number of commands

to a Vanilla core:

These commands include:

- writing a value into the register file or constant pool,
- writing an instruction into the instruction memory,
- specifying a PC and initial barrier value for the core to begin executing at, and
- specifying a barrier mask; which helps the core coordinate with other cores.

A network is responsible for initializing Vanilla cores before it runs. A Vanilla ISA implementation is responsible for executing the commands in these packets, even if a program is currently executing. The core has some flexibility in when it executes packets relative to program instructions, but must guarantee that the instruction stream will not cause an indefinite stall of network packet execution.

If the core sees this packet while it is executing code, it should enter the exception state (see below).

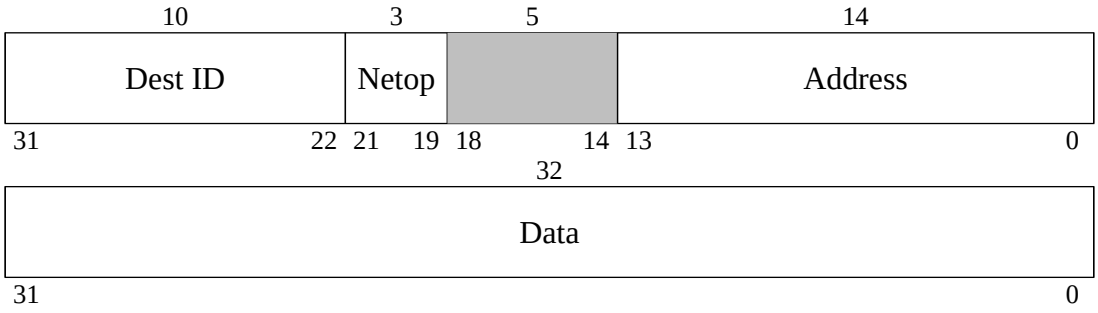


Figure 2: **Network Packet.** The network packet is 64-bits wide, and may be transmitted as two 32-bit words or a single 64-bit wide word. If it is a single word, then Data occupies the low 32-bits.

Network Packets. The network packet is shown in Figure 2, with the start and end bit of each field under it. A network packet consists of two 32-bit words, which depending on the implementation, may be sent serially, or may be sent as a single 64-bit wide word. If it is sent as a 64-bit wide word, the data word occupies bits 31..0, and the other "header" word occupies bits 63..32.

Netop	Function
000	noop
001	`IMEM[Addr] = Data;`
010	`RegFile[Addr] = Data;` or `Constant_Pool[Addr] = Data;`
011	`PC = Addr; Barrier = Data;`
100	`Barrier_Mask = Data;`
101	unused
110	unused
111	unused

Table 3: **Network operations.** The network packet specifies a command for the Vanilla core to execute.

The first field is the destination ID field. A given core only executes packets that have that core's ID, or a broadcast id of all 1's. The second field, the 3-bit network opcode ($Netop$), specifies the operation that the core has to perform. These operations are shown in Table 3.

The address field contains the address of the instruction for instruction memory or the destination register for the value, or the PC for execution. The data field has the required data or the barrier mask.

Exception State. The ISA supports an exception state; which generally means something bad has happened. Upon entering the exception state, execute stops. Currently, the only such exception is that if the network sends a new PC while the core is executing a program. A core may receive a PC packet only at reset or after it has executed a $WAIT$ instruction.

3 Vanilla ISA Assembler

This section describes the assembler which translates Vanilla ISA instructions into Vanilla machine code. The assembler supports a few features unique to the Vanilla ISA, as well.

3.1 Keywords

The assembler framework supports the following keywords and reserved words:

.text

Indicates the start of the text section, consisting of instructions. *.text* can appear many times in an assembly file, but they will be merged into a single section in the output of the assembler.

.data

Indicates the start of the data section. Similar to *.text*, there can be multiple *.data* in an assembly file.

.word

Specifies the word data at a memory location. It can be followed by multiple one or more words to describe data for multiple words.

.fillword

Duplicates a word data many times. For example, *.fillword 10 0x0* duplicates 0x0 ten times.

.byte

Specifies the byte data at a memory location. It can be followed by multiple one or more words to describe data for multiple bytes.

.fillbyte

Duplicates a byte data many times. For example, *.fill 10 0x0* duplicates 0x0 ten times.

.reg, .constreg, .const

The most important extra feature is giving values to the registers, using these three keywords. Since there is no instruction containing immediate value in the vanilla cores (except offset of jumps), the required values must be stored in the constant registers (the last 32 registers which are read-only for the core), using *.constreg* keyword. The registers 1 to 31 can be valued by the *.reg* keyword as well, but is prohibited, since they can be written via the core. The *.const* keyword, makes the work easier by setting some labels starting with *%* as the constants, for example *.const %example, 0xEEF*. The assembler will assign a constant register to it which were not used by the other two keywords. Moreover, you can set the desired constant register for the label as well. You can use the *%example* as operand in your instructions. But be careful of limited amount of constants, you can get the error running out of constants. Examples:

- *.reg $r21,0xFFEEFFEE*
- *.constreg $c21,0xFFEEFFEE*
- *.const %example, 0x66FF56EE*
- *.const %example, 0x66FF56EE, 23*

.kernel

Another feature of the assembler is supporting several pieces of code sharing the same data memory. Each part begins with a *.kernel* keyword and each code section must be in a kernel and there is only one data section for all kernels, which is shared among them.

3.2 Labels

Represents an instruction or data memory address, similar to labels in other languages, when you use labels, you can minimize the amount of manual modifications in your

assembly code upon insertions or deletions of instructions and/or data. Labels make the code maintenance cost more manageable by keeping the effect of a code modification be local to a label.

Moreover, Since there could be several kernels in one assembly file, you can specify that the label is in which kernel, by using a dot, for example *kernel-name.label*. If there is the same label both in data and code section, the assembler will first look through the data section. It is obvious that using the same labels is prohibited.

3.3 Comments

The assembly code is very hard to be understood. To prevent the programmers from forgetting why they write these code, the assembler framework also allows the programmers to append some comments. All comment must be started with "//". In other words, all the contents after "//" will not be processed by the assembler framework.

3.4 Instruction Format

The assembler framework can recognize instructions with the form of operator + operands. There are for types of instruction formats:

- *LW $rd, $rs*
- *BEQ $rd, offset*
- *BAR $rs*
- *DONE*

3.5 Output files

To run the assembler, you have to give two inputs. One is the name of the assembly file and the other is the name for the file containing the data section:
Java vanillaAssembler test.asm dataMemory Since there is only one data section, the assembler will make a file named *dataMemory.hex* for this part which is shared among every kernel.

Moreover, for each kernel, there are three output files. One is the *kernel-name_i.hex* which contains the instructions in binary, *kernel-name_r.hex* which has the values for the registers before running the code, and *kernel-name_info.txt* which shows which constant is mapped to which register for further debugging.

3.6 Auto adjust in Data Section

The assembler will auto adjust the data section by words. For example if you put one word, 7 bytes, and another word consecutively, the assembler will add a zero byte before the second word. Using the labels would prevent such problems.

4 Vanilla Core:
A single-cycle Vanilla ISA Implementation

4.1 Overview

Vanilla Core is a single-cycle implementation of the Vanilla ISA. Some aspects of the core are implementation-specific; we record them here. Of interest are the memory handshake protocol and a barrier output which is described later in this section.

The core implements the 32-element register file and the 32-element constant pool together in a unified 64-element 2-read port, 1-write port register file. The core has a 1024-word instruction memory, and employs a 32-bit data address space. Each instruction is 16 bits and registers and data bus between memory and core are 32 bits. Only 32 registers can be written by the core, and the rest of the registers and the instruction memory are initialized through the network.

4.2 State machine

There are four states in Vanilla cores, IDLE, WORK, WAIT and ERROR. After resetting the core, it goes to the IDLE state and waits for the network to get the PC and start execution. Since all instructions, except data memory related ones, take only one cycle, the core goes to WORK state. If there is a Memory related instruction and the core has to wait for the memory, it goes to WAIT state. Afterwards, when memory is ready, it goes back to the WORK state. In the end, when it reaches the DONE instruction, the program is finished and it goes to IDLE state. If a problematic situation happens, the core sets the error signal from the next clock cycle. However, it will finish the execution of current instruction before going to the ERROR state. It will remain there until the

reset signal makes it to go to the IDLE state.

4.3 Booting from the Network

The network protocol allows code to be transferred into the instruction memory, and for the constant pool and registers to be initialized. After initializing this state, the network protocol will transmit the PC for the core to start execution. If an incorrect network packet is arrived, the core sets the error signal and goes to error state. Upon error signal, reset may be asserted to reinitialize the core. At this point, the standard network boot process may be performed.

The core can receive correct packets during execution, like a new instruction at end of the program while it is executed from start. These network packets have higher priority than execution itself, since they must not be dropped. The core will focus on execution of instructions when there is no network packet. Because the core does not buffer network packets, it is illegal to send a new PC while the core is executing. This is because a data memory access may take arbitrarily long, but the new PC cannot be loaded until the current instruction finishes. An arbitrary number of network packets could also arrive in the meantime. An alternative implementation could keep track of the network-write PC until after the data memory access completes, but the semantics of the network asynchronously interrupting a core are ill-defined and do not lend themselves to a clean programming model anyways.

4.4 Data Memory Handshake

The Vanilla core module is implemented with an external data memory. This decouples the core from the data memory design and also makes it easy to implement a multicore design with a shared memory. Since different data memory implementations could take

different amounts of time, and since a multicore system might have contention, we employ a flexible hand-shake protocol in each direction between core and data memory. We call this the Valid/Yumi protocol.

Here is a full description of the protocol:

The data memory communicates with the core using “valid/yumi” synchronization in both directions. It works like this:

1. The core indicates that it would like to make a request to the data memory by asserting the valid signal. The assertion of this signal will coincide with data being transmitted and an indiciation of whether it is a read or write. It will hold this signal high until the request is acknowledged (yumi’d) by the data memory.
2. When the data memory is read to process the request from the core, it acknowledges the request. (On the next cycle, the core must drop the valid signal low unless it intends to send a second request.)
3. When the data memory has completed the load or store, it asserts its valid signal until the core acknowledges (yumi’s) the request. The assertion of this signal will coincide with data being transmitted.
4. When the core is ready to accept the response from the data memory, it will assert the yummi signal. On the next cycle, the data memory must drop its valid line unless it intends to send another response.

A few notes about valid/yumi:

- upon receipt of a yumi, valid should not be dropped until the next cycle, otherwise a combinational loop will be formed.

- this protocol is flexible and a single request/reply pair could happen in a single cycle, or across many cycles.
- the protocol can support pipelined requests where successive requests are sent before any replies are received.