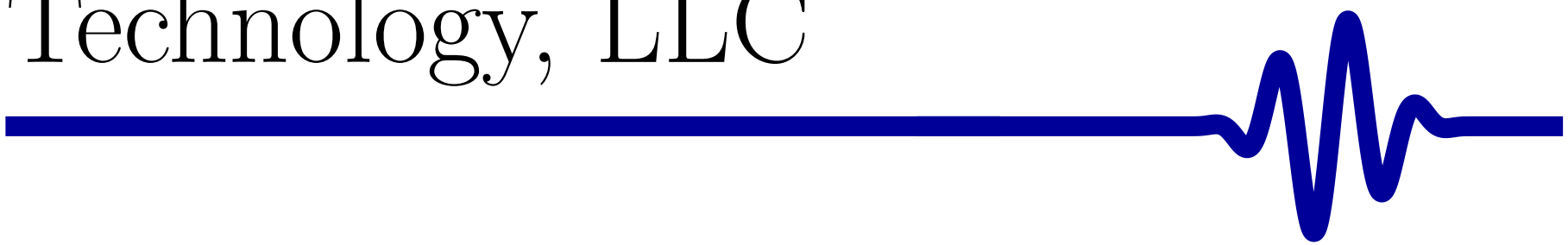
Gisselquist
Technology, LLC

The ZipCPU

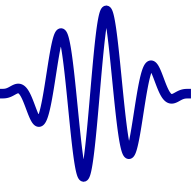
A resource efficient 32–bit SoftCore CPU

Daniel E. Gisselquist, Ph.D.

Octbober, 2016



Overview

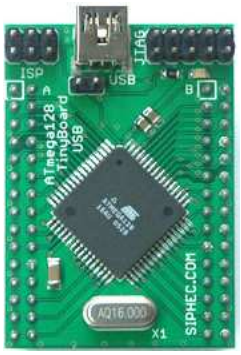


- Why do I need a ZipCPU?
- How has the ZipCPU been made resource efficient?
  - Simplified bus
  - Minimal instruction Set
  - A simpler approach to Interrupts
- Enhancements to the basic simplified ZipCPU
- What performance can be expected?

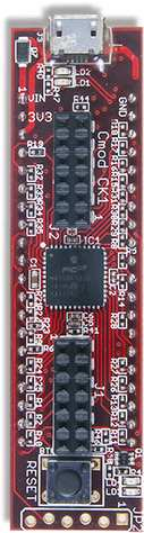
Vision: SwiC

If what you needed was a CPU, you would've bought one.

- All of the CPU's below are both *cheaper* and *faster*



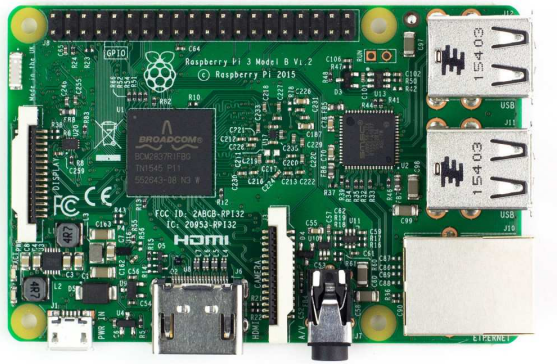
ATmega128



PIC32

MSP430

TeensyLC



RPi3/ARM

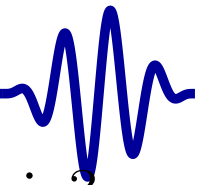
Vision: SwiC

But you bought an FPGA. Why?

- Because you had an application that needs lots of special purpose, high speed, processing to complete in time

Example: NetFPGA SUME

Vision: SwiC



Does your application have a need for any sequential logic?

- Yes, but there's never enough room for it, and ...
- Both industry solutions, MicroBlaze and NiOS-II, would make your product vendor dependent
- What you need is a System within a Chip, or a SwiC!

This is therefore our goal and vision!

- A *small core* that can be added to a special purpose application, without drawing away too many resources
- An *Open Source core* than can be adapted to any vendor's hardware

Survey of CPUs

Feature	NiOS	μBlaze	ECO–32	RISC-V	OpenRISC	LM32	ZipCPU
Open Architecture?	**No**		Yes				
Number of Instructions	**86**	**129**	**61**	**50+**	**48+**	**62**	
OpCode Bits	6–17	6–11	6	10	6–32	6	
Interrupt/Exception Vectors	1	6	2	9+	14	32	
Register Indirect plus displacement (bits)	16			12	16		
Immediate direct addressing (bits)	16, using R0=0						
Relative branching (bits)	16		26 (28)	21	26	21	
Conditional branching (bits)	16		16 (18)	13	26	16	
Register Size (bits)	32			32 (Opt. 64 Exts.)		32	
Special Purpose Registers	**6**	**25**	**6**	**66+**	**65+**	**10**	
General Purpose Registers	32 (but R0=0, others are unusable, . . . 24)						
8–bit data	Yes						
16–bit data	Yes						
32–bit data	Yes						
64–bit data	No			Yes, by extension		No	
32–bit floats	Optional		No	Yes, by extension		No	
64–bit floats	No			Yes, by extension		No	
Vector instructions	No			Not yet	64-bits, Ext	No	
MMU	Yes, but optional						
Instruction Cache	Yes, configurable						
Data Cache	Yes, configurable						

To be revealed at ORCONF 2016

Survey of CPUs

Feature	NiOS	μBlaze	ECO–32	RISC-V	OpenRISC	LM32	ZipCPU
Open Architecture?	**No**		Yes				
Number of Instructions	**86**	**129**	**61**	**50+**	**48+**	**62**	
OpCode Bits	6–17	6–11	6	10	6–32	6	
Interrupt/Exception Vectors	1	6	2	9+	14	32	
Register Indirect plus displacement (bits)	16			12	16		
Immediate direct addressing (bits)	16, using R0=0						
Relative branching (bits)	16		26 (28)	21	26	21	
Conditional branching (bits)	16		16 (18)	13	26	16	
Register Size (bits)	32			32 (Opt. 64 Exts.)		32	
Special Purpose Registers	**6**	**25**	**6**	**66+**	**65+**	**10**	
General Purpose Registers	32 (but R0=0, others are unusable, . . . 24)						
	Yes						
	Yes						
	Yes						
	No			Yes, by extension		No	
	Optional		No	Yes, by extension		No	
64–bit floats	No			Yes, by extension		No	
Vector instructions	No			Not yet	64-bits, Ext	No	
MMU	Yes, but optional						
Instruction Cache	Yes, configurable						
Data Cache	Yes, configurable						

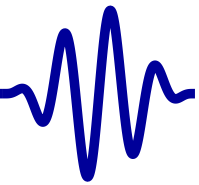
This is *way too* complex. Something simpler is needed: the ZipCPU

To be revealed at ORCONF 2016

Survey of CPUs

Feature	NiOS	μBlaze	ECO–32	RISC-V	OpenRISC	LM32	ZipCPU
Open Architecture?	**No**		Yes				Yes
Number of Instructions	**86**	**129**	**61**	**50+**	**48+**	**62**	26+
OpCode Bits	6–17	6–11	6	10	6–32	6	5+
Interrupt/Exception Vectors	1	6	2	9+	14	32	None
Register Indirect plus displacement (bits)	16			12	16		14 (16)
Immediate direct addressing (bits)	16, using R0=0						18 (20)
Relative branching (bits)	16		26 (28)	21	26	21	18 (20)
Conditional branching (bits)	16		16 (18)	13	26	16	18 (20)
Register Size (bits)	32			32 (Opt. 64 Exts.)		32	32-bits
Special Purpose Registers	**6**	**25**	**6**	**66+**	**65+**	**10**	1 (x2)
General Purpose Registers	32 (but R0=0, others are unusable, . . . 24)						14 (x2)
8–bit data	Yes						No
16–bit data	Yes						No
32–bit data	Yes						Yes
64–bit data	No			Yes, by extension		No	No
32–bit floats	Optional		No	Yes, by extension		No	Not yet
64–bit floats	No			Yes, by extension		No	No
Vector instructions	No			Not yet	64-bits, Ext	No	No
MMU	Yes, but optional						(In test)
Instruction Cache	Yes, configurable						Same
Data Cache	Yes, configurable						Not yet

Vision



Build a simplified, open source, low–area, soft-core CPU

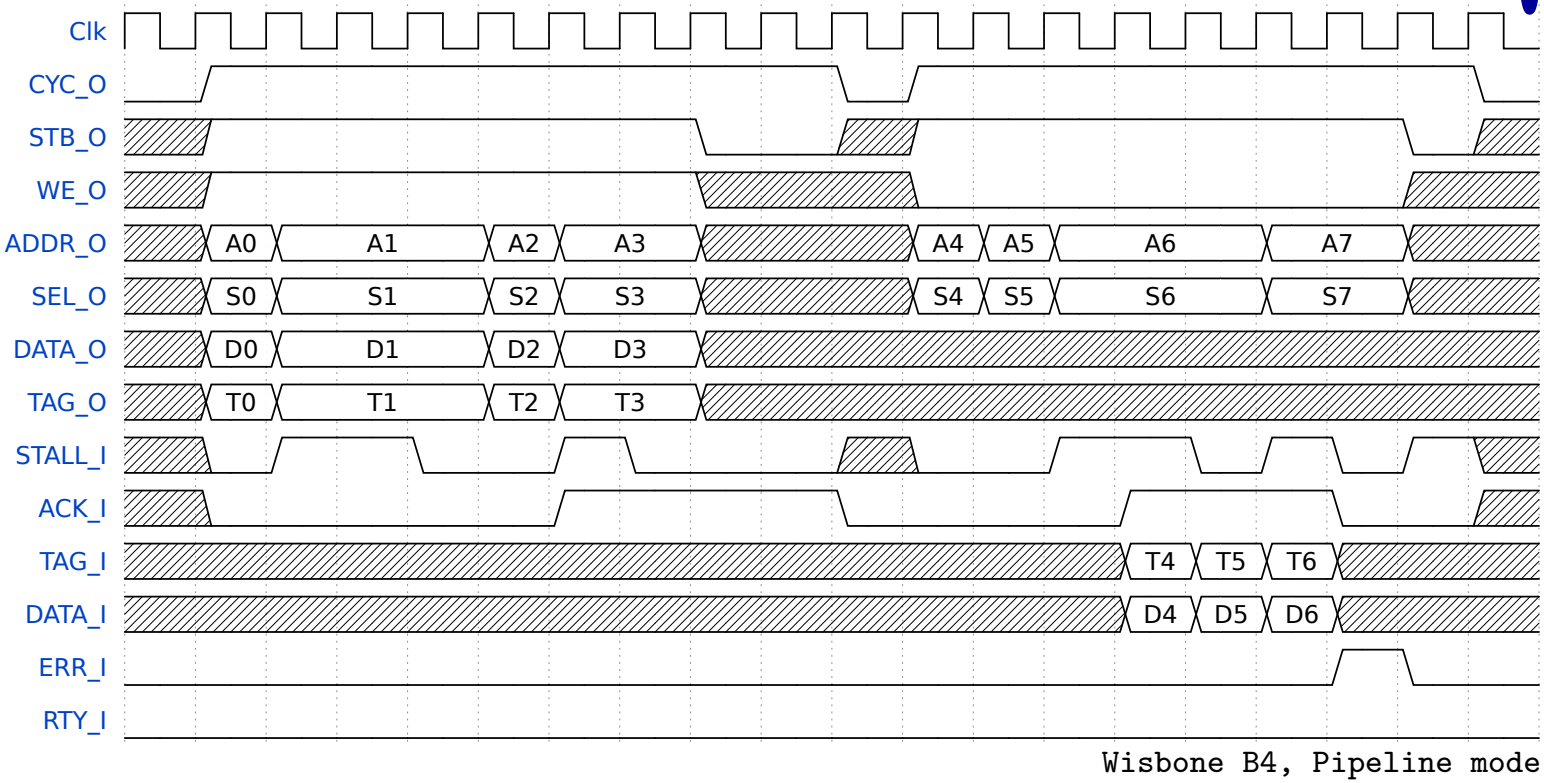
Goals

1. 32–bit
2. Pipelined
3. Wishbone
4. Threadable
   (Supervisor mode)

Choices

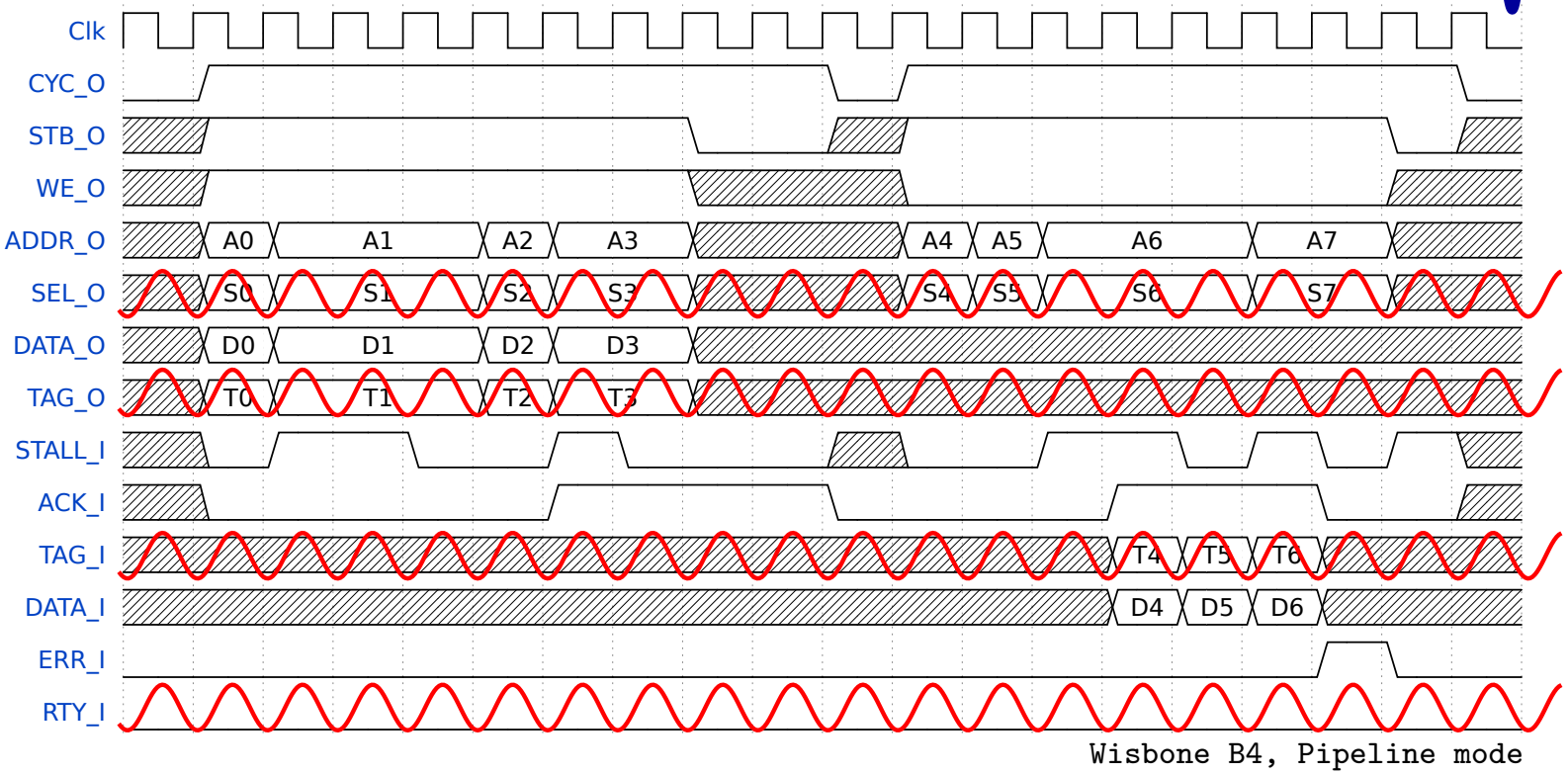
1. Simplified Wishbone
   - Single word size: 32–bits
   - Only aligned accesses
   - Only one bus for I/D
2. Simplified instruction set
3. No interrupt vectors–interrupts just switch modes

Full Wishbone



Let's simplify this . . . can we remove anything we don't really need?

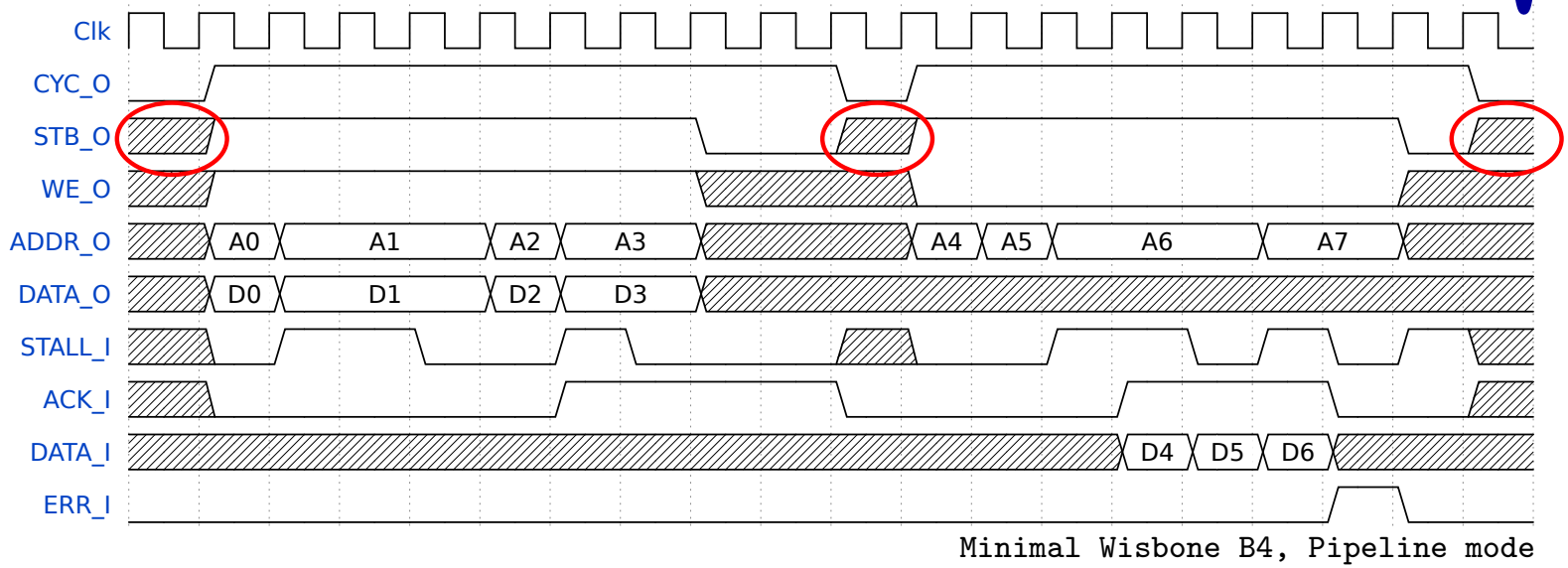
Simplified Wishbone



Let's remove the wires we don't need:

- Avoid anciliary information (TAGS, CTI_x)
- Merge the LOCK and CYC lines together
- Force all transactions to be 32–bits, so remove SEL lines
- Ignore retries (RTY), we weren't using them anyway

Simplified Wishbone



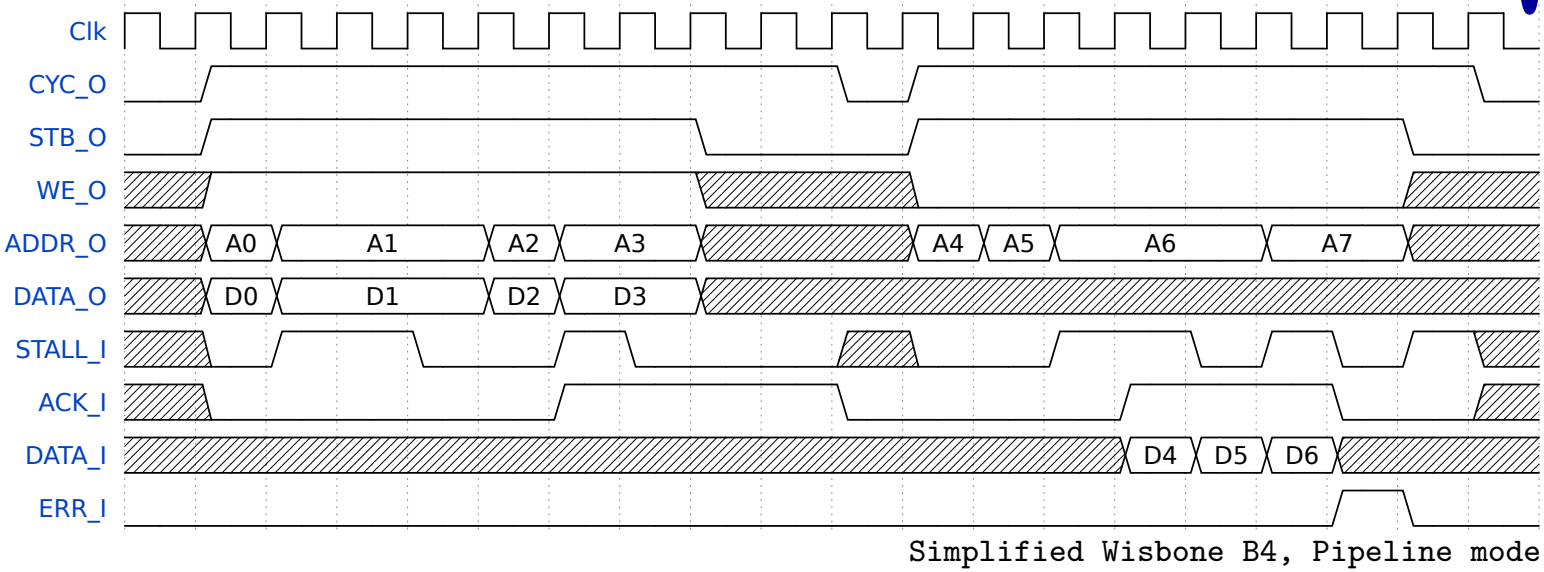
Let's simplify our remaining logic:

> Insist that `STB` be zero, rather than don't care, if `CYC` is zero

This simplifies a slave's decode logic: `if (CYC)&&(STB)` becomes `if (STB)` in any bus slave/peripheral.

I would recommend this change to the Wishbone standards body.

Simplified Wishbone



Simplified Wisbone B4, Pipeline mode

Transaction is complete when `(CYC)` returns to zero

- Bus is idle after the last `ACK`

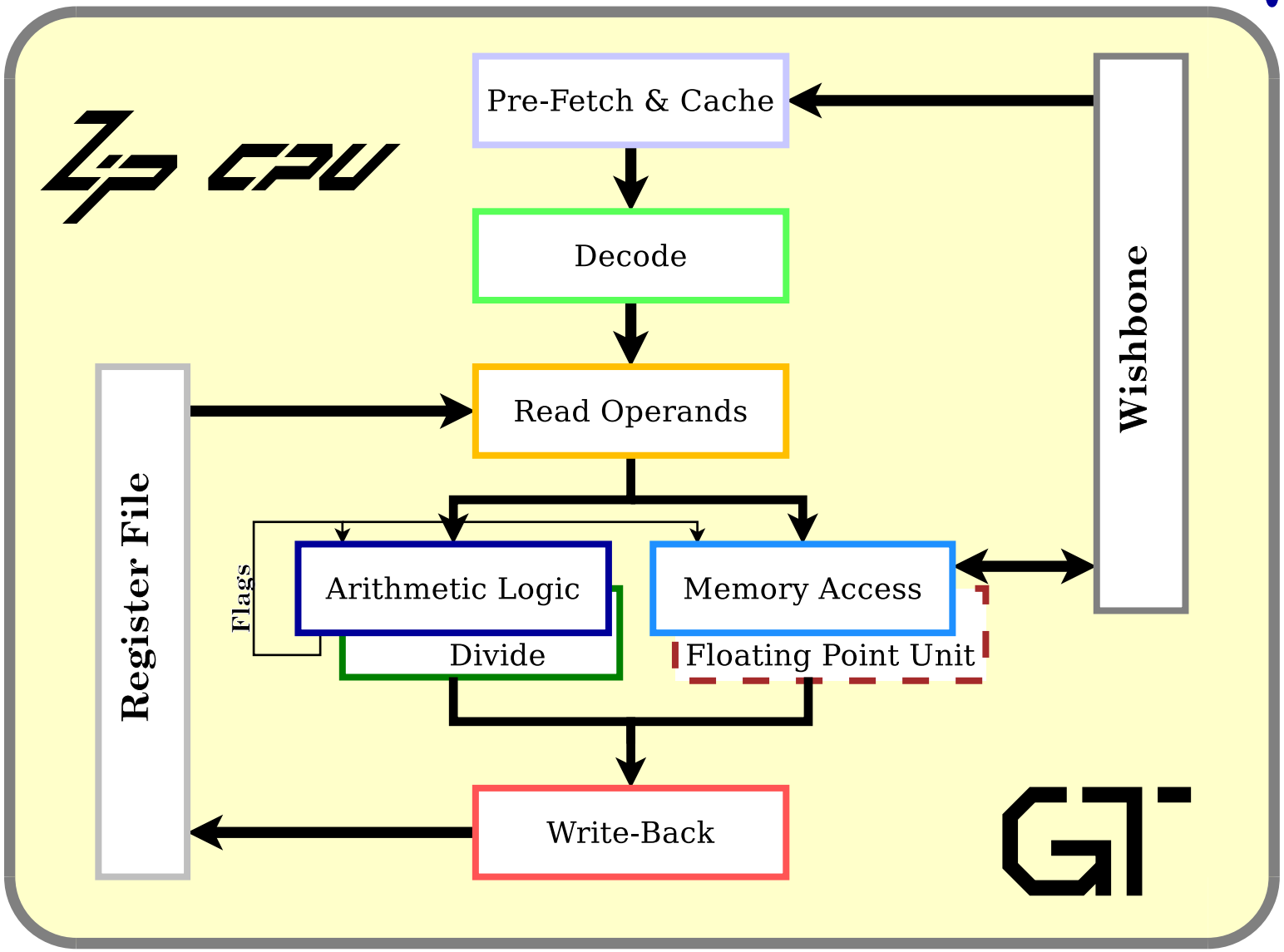
`(STB)&&(!STALL)` implies a transaction request took place

- Devices that don't stall only need to check `STB`
- Master must set check both `STB` and `STALL`

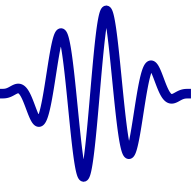
Every request expects an `ACK`

All transactions use pipeline mode

CPU Structure



Register Set

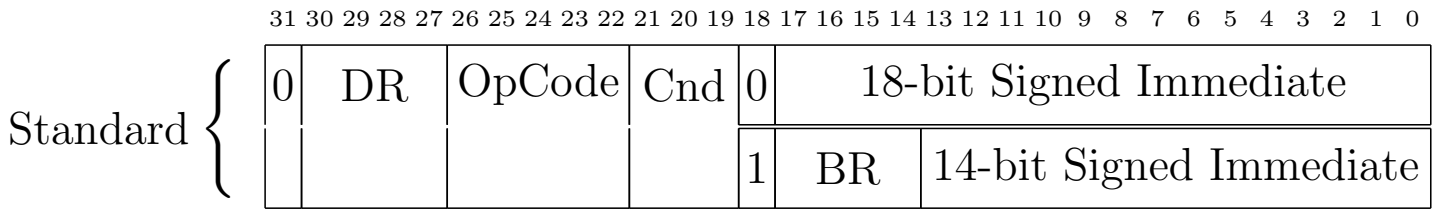


Two register sets, only one set is active at any time

Supervisor Register Set			User Register Set	
sR0(LR)	sR8		uR0(LR)	uR8
sR1	sR9		uR1	uR9
sR2	sR10		uR2	uR10
sR3	sR11		uR3	uR11
sR4	sR12(FP)		uR4	uR12(FP)
sR5	sSP		uR5	uSP
sR6	sCC		uR6	uCC
sR7	sPC		uR7	uPC
Interrupts Disabled			Interrupts Enabled	

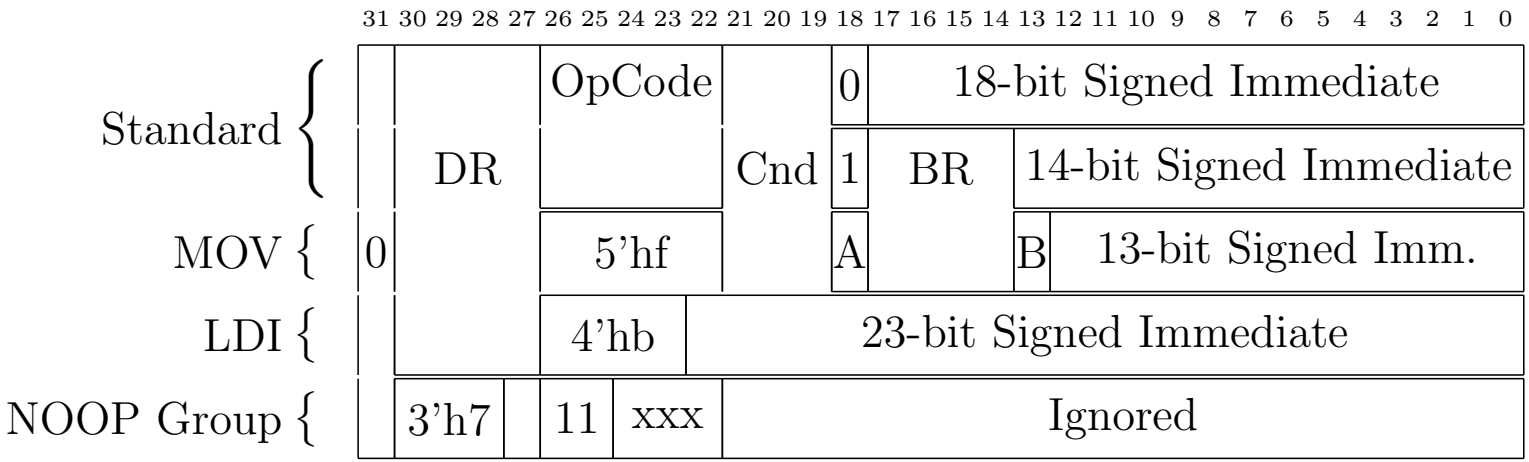
- Only the PC/CC registers have any special H/W purpose
- A special `MOV` instruction provides access to user regs

Simplified Insns



- 5–bit OpCode allows for 32 instructions
- 3–bit Condition allows every instruction to be conditional
- 4–bit Register code allows for up to 16 registers
- All instructions take either one or two registers
  - `OP.C #X+Rb,Ra`
  - `OP.C #X,Ra`
- This works until
  - the supervisor needs access to the user registers, or
  - you want to load a large number into a register

Instruction Set



The entire instruction set was designed to keep decoding simple

32 OpCodes

ALU Instructions

SUB
AND
ADD
OR
XOR
LSR
LSL
ASR
MPY
LDILO
MPYUHI
MPYSHI
BREV
POPC
ROL
MOV

CMP
TEST
LOD
STO

Memory operations

DIVU
DIVS

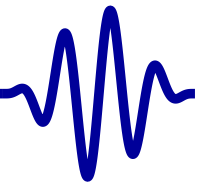
Divide unit

LDI

FPADD /NOOP
FPSUB /BREAK
FPMPY /LOCK
FPDIV
FPCVT
FPINT
Reserved
Reserved

Reserved for floating point unit

Notable



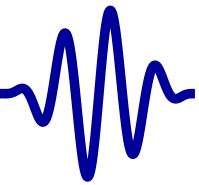
Unusual Instructions

1. `BREV` (bit reverse)
2. `TEST` (`AND` sets cond)
3. `CMP.`x (sets cond if X)
4. `ROL` (rotate left)
5. `POPC` (pop count)
6. `LDILO` (Load imm, lo)
7. `LOCK` (for atomic access)
8. `BRA` (`ADD #x,PC`)

"Missing" Instructions

1. `LB`, `SB`, `LH`, `SH`
2. `PUSH`, `POP`
3. `CALL`, `JSR`, `JAL`, `JALR`
4. `RETurn`
5. `ADDC`, `SUBC`, `SUBR`
6. Compare and branch
7. Shift w/ carry
8. Compare and set
9. Set if zero

8 Conditions

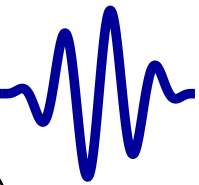


ZipCPU supports eight conditions:

Code	Meaning	CC Bits	Usage
3'b000	(Always)		
3'b001	.LT	N	$A < B$ (signed)
3'b010	.Z	Z	$A = B$
3'b011	.NZ	!Z	$A \neq B$
3'b100	.GT	(!N)&(!Z)	$A > B$ (signed)
3'b101	.GE	(!N)	$A \geq B$ (signed)
3'b110	.C	C	$A < B$ (unsigned)
3'b111	.V	V	On overflow

Any instruction can be executed conditionally

Function Calls



There are no subroutine OpCodes (`JSR`, `JAL`, `JALR`, etc.)

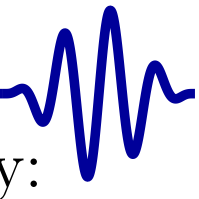
- Such instructions require two writes to the register set: one to store the PC, one to set the PC
- Solution: Function calls just take an extra instruction

```
MOV return_lbl(PC),R0  ; This is the link instruction
BRA subroutine         ; Implemented as an ADD #x,PC
return_lbl:
```

- Returns are simply indirect jumps

```
JMP R0 ; Indirect branches cost 6-cycles
       ; Implemented as a MOV R0,PC
```

Interrupts

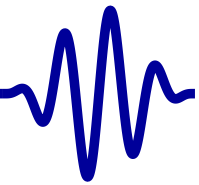


Traditionally, on an interrupt, most CPUs automatically:

1. `PUSH CC`
2. `PUSH PC`
3. `LOD ITBL[INT],PC`

```
__attribute((interrupt,N))__
void ISRN(void) {
   // Save user context/stack
   //   Special purpose interrupt processing
   // Restore user context/stack
   // GCC ends this with an IRET instruction
}
void (*ITBL)[] = { ..., ISRN, ... };
```

Interrupts



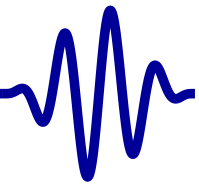
1. `PUSH CC`
2. `PUSH PC`
3. `LOD ITBL[INT],PC`

This requires extra CPU logic to support: special purpose instructions and registers may be required.

```
__attribute((interrupt,N))__
void ISRN(void) {
  //
  //
  //
  //
}
void (*ITBL)[] = { ..., ISRN, ... };
```

ISR coding can be a real challenge to program for, and traditionally requires hand-optimized assembly.

Interrupts

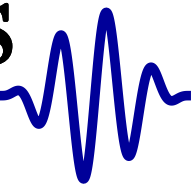


ZipCPU's approach to interrupts is . . . different:

- Only one interrupt line to the CPU
- No interrupt vectors, tables or "handler" functions
- ZipCPU just switches from user to supervisor mode

```
void entry(void) { // Supervisor entry function, on CPU reset
    // Setup up user tasks
    while(1) {
        zip_rtu(); // Return to userspace instruction
                   // Equivalent to OR #0x20,CC, sets GIE bit
        // Handle interrupts, traps and exceptions
        // Run scheduler, swap contexts?
    }
}
```

Basic Enhancements



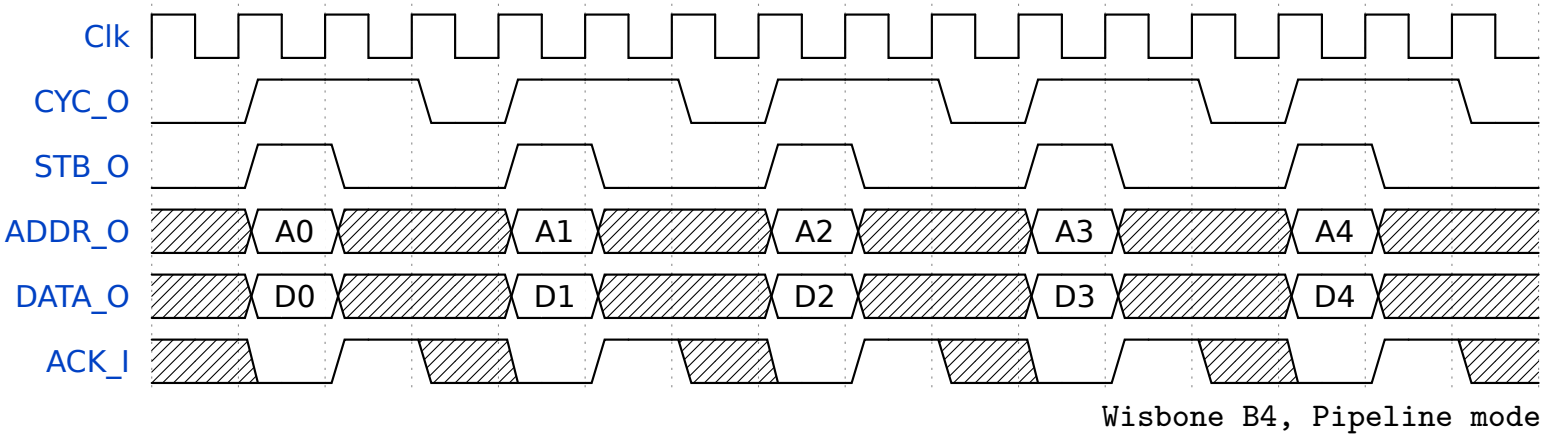
- Pipelined memory access

  Recovers some of the missing data cache performance

- Early Branching
- 14'bit packed OpCodes (VLIW)
- *Future* VLIW triple operand decoder, one cycle instruction
  - `ADD Ra,Rb,Rc` becomes `MOV Ra,Rc` | `ADD Rb,Rc`
  - `SUB Ra,Rb,Rc` becomes `MOV Rb,Rc` | `SUB Ra,Rc`
  - `SUBR Ra,Rb,Rc` becomes `MOV Ra,Rc` | `SUB Rb,Rc`
- *Future* MMU and data–cache
- *Future* FPU

Pipelined Memory

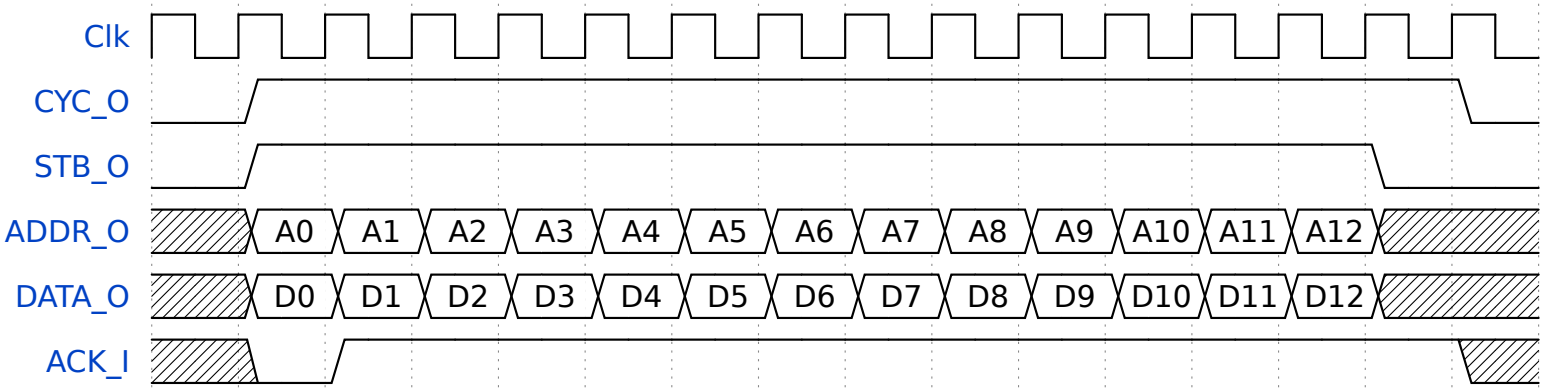
Without pipelining:



Best case transfer time: $3N$ clocks

Pipelined Memory

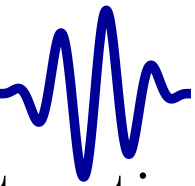
With pipelining:



Best case transfer time: $N + 2$ clocks

ZipCPU supports pipelined `LOD` and `STO` instructions

Pipelined Memory

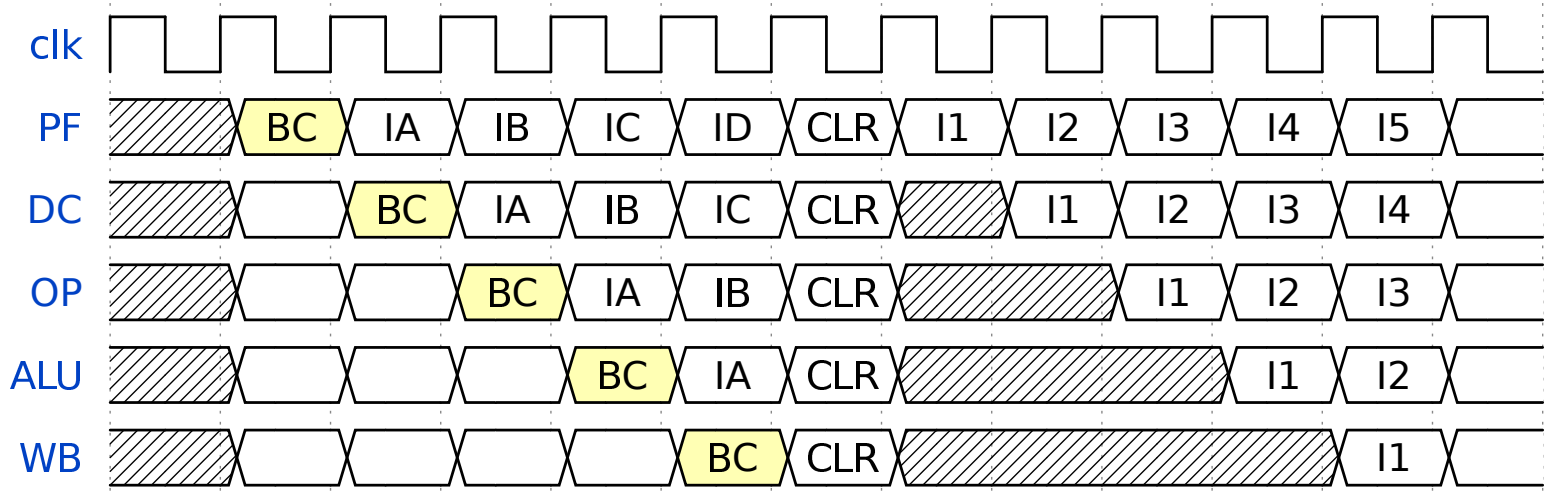


To use the pipelined wishbone mode from a ZipCPU instruction:

- Requires adjacent instructions
  - All must be either LODs or STOs
  - All must use the same base address register
  - All must have the same, or a more specific, condition
  - Have incrementing (or identical) immediate offsets
- Example: Stack frame set up

```
SUB 3,SP       ; Allocate stack space
STO R0,(SP)    ; First offset is zero, avoids a stall
STO R1,1(SP)   ; Second STO costs only one more clock
STO R2,2(SP)   ; One more clock here
```

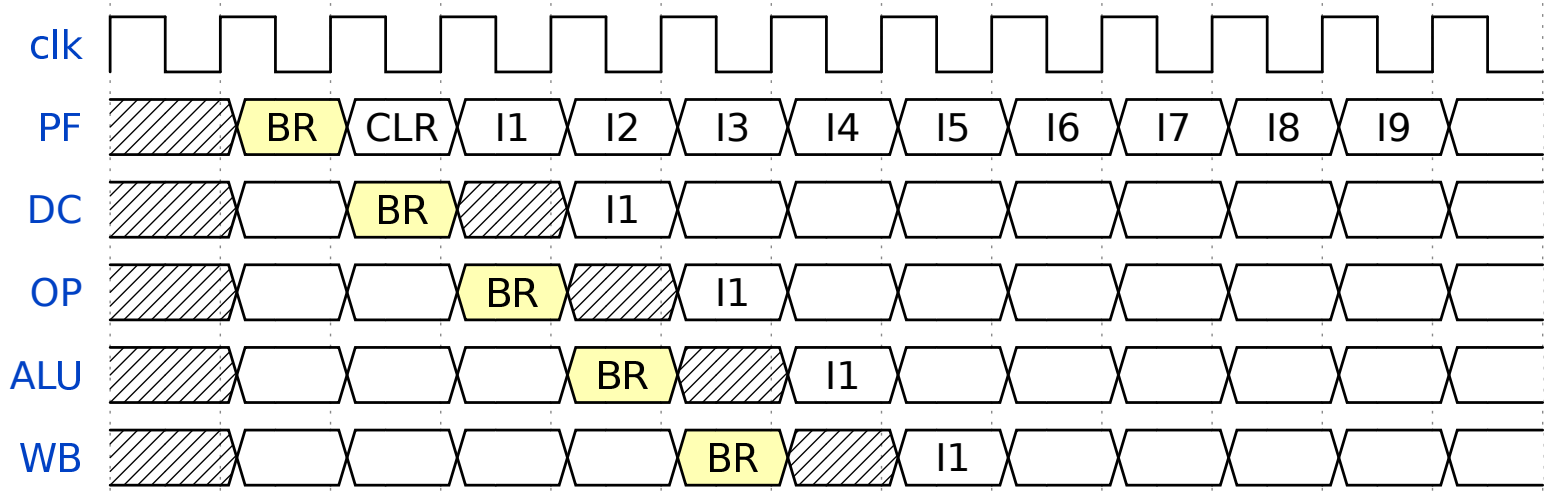
Early Branching



Instruction cost: 6 clocks

Without special logic, branches cost a full pipeline stall

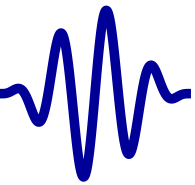
Early Branching



Instruction cost: 2 clocks

Early branching allows early detection of branch instructions, before the whole pipeline needs to be cleared.

Early Branching



- Three early branching instructions supported

  `LDI #x,PC` 2–cycles

  `ADD #x,PC` (`BRA`) 2–cycles

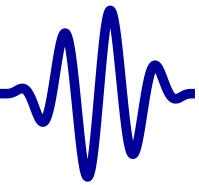
  `LOD (PC),PC` (`LJMP`) 3–cycles

  `LJMP` *was an afterthought to support linking*

- `LDI #x,PC` is hardly ever used
  - The address must fit inside 23–bits (signed)
  - The decision of which instruction (`LDI` vs `LJMP`) is made before the absolute address is known

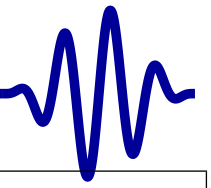
    Perhaps I should recover this unused logic . . .

Benchmark



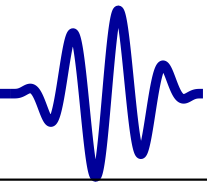
- We'll use the Dhrystone benchmark
  - It's public domain, and doesn't require floating point
- GCC compiled code
- Hand optimized assembly for the library routines:
  - strcmp(), strcpy(), memcpy()
- Memory structure
  - All memory placed in block RAM (8kW on S6LX25)
  - 1kW I-Cache for CPU (when pipelined)
  - No data cache (that's gonna hurt)
- Accomplished via a XuLA2-LX25 SoC Verilator simulation
- Formula: DMIPS/MHz $= 10^6 \cdot N/\text{cycles}/1757$

Performance



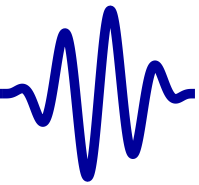
Configuration	6-LUTs	ΔLUTs	CPU LUTs	DMIPS/MHz
S6SoC (w/CPU, no pipelining)	2345			
XuLA2 SoC, No CPU, Base System	2446			
CPU (no-pipeline, w/ debug)	3725	1286	[1286]	
Multiply	3965	233	[1519]	0.128
CPU (Pipelined, 1kW I-Cache)	4383	418	[1937]	0.465
Pipe Memory	4543	160	[2097]	0.613
Early Branching	4541	-2	[2095]	0.687
Divide *(Optimized out of Dhrystone Benchmark)*	4905	364	[2459]	
VLIW	5030	125	[2584]	
ZipSystem on XuLA2 board				
Basic (2x PIC, 3x timers, 2x watchdogs, jiffies)	5615	585	[3169]	0.683
8x Performance Counters	6232	617	[3786]	
DMA	6882	650	[4436]	**0.744**
XuLA2 Full up SoC (includes SD)	7372	490	[4926]	

vs OpenRISC



DMIPS/MHz	CPU
0.74	ZipCPU
0.97	OpenRISC Why does OpenRISC score higher? Because Dhrystone requires byte-wise string operations, and newlib optimizes the strcpy, strcmp, and memmove functions to operate on 4-bytes (32-bits) at a time when possible/aligned. If we could pack the characters strings in the test, the ZipCPU score would improve:
0.95	ZipCPU *(Modified Dhrystone for packed strings)*

How'd we do?



Did we build a simplified,

- Simplified wishbone, instruction set, interrupt processing

open source,

- GNU General Public License (GPL), v3.0

low-area

- 1300-2600 LUTs

soft-core CPU?

Yes! The Zip CPU

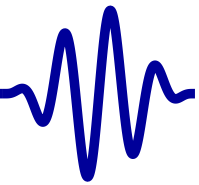
Try it at: `https://github.com/ZipCPU/zipcpu` or
`http://opencores.org/project,zipcpu`

Gisselquist
Technology, LLC

In all labour there is profit . . .

Prov 14:23a

Missing Conditions

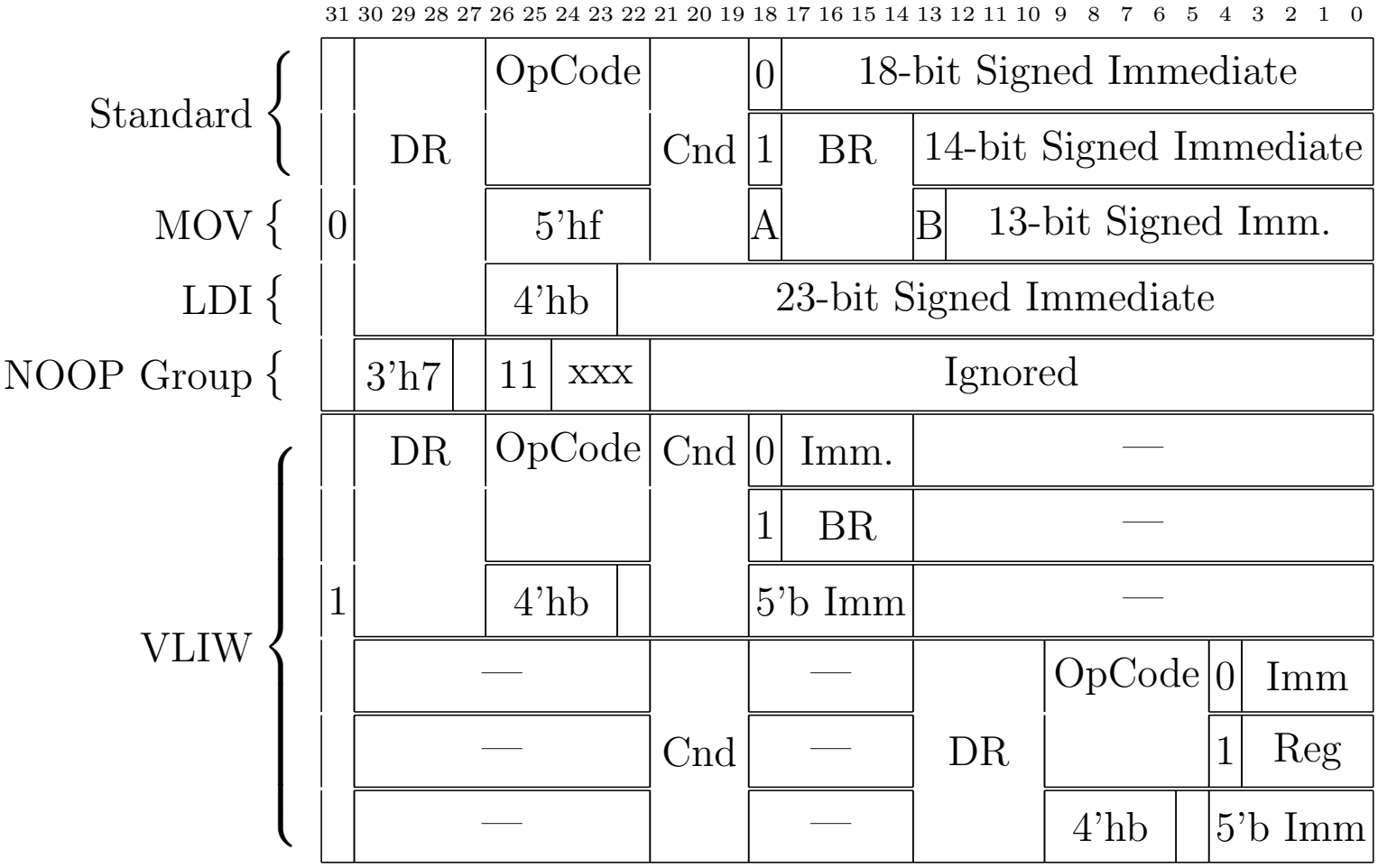


Four common conditions are missing

Missing Condition			Replacement	
LTE	$A \leq N + B$	`CMP` N+`Rb,Ra`/OP.LTE	`CMP` -N+`Ra,Rb`	/OP.GE
LEU	$A \leq N + B$ (U)	`CMP` N+`Rb,Ra`/OP.LEU	`CMP` 1+N+`Rb,Ra`	/OP.C
GEU	$A \geq N + B$ (U)	`CMP` N+`Rb,Ra`/OP.GEU	`CMP` 1-N+`Ra,Rb`	/OP.C
GTU	$A > N + B$ (U)	`CMP` N+`Rb,Ra`/OP.GTU	`CMP` -N+`Ra,Rb`	/OP.C

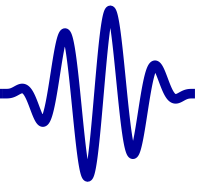
Only one problem: what if the replacements overflow?

VLIW Instructions



VLIW instructions pack two instructions into one word, at the cost of a smaller immediate range.

Function Calls



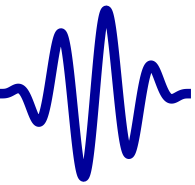
GCC doesn't optimize function calls very well:

```
    MOV return_lbl(PC),R0 ; Not somewhere_else?
    LJMP subroutine
return_lbl:
    BRA somewhere_else
```

Neither does it optimize the returns well:

```
    BRA.x subroutine_complete ; Not JMP.x R0?
    ...
subroutine_complete:
    JMP R0
```

Branch Prediction



The ZipCPU has no branch prediction logic.

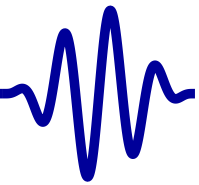
- Branch prediction can be done statically by the compiler
- Normal: conditional branch costs 6 cycles, exit costs one

```
loop:
    ; Work
    BZ loop      ; Suffers a full pipeline stall if taken
```

- Optimized: branch costs 3 cycles, exit costs 6

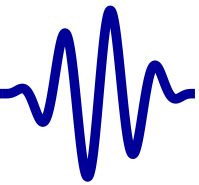
```
loop:
    ; Work
    BNZ skip     ; Full pipeline stall if taken
    BRA loop     ; Exploits early branching
skip:
```

Peripherals



- QSPI Flash controller
- Nearly internal
  - Interrupt controller
  - Timers, Counters, Jiffies, Watchdog timer, bus watchdog
  - Direct Memory Access (DMA) Controller
- SDRAM controller (DDR3 work in progress)
- UART, GPIO, PWM, FMTX Hack, Real-Time Clock
- GPS NMEA processor, internal timestamp generator
- SD card interface (SPI only, SDIO work pending)
- OLEDrgb interface
- Ethernet (MII interface)

Challenges



- Lack of byte and halfword instructions
  - This is slowing down the newlib port
- Lack of data cache
  - This may be slowing down our overall performance
  - The pipelined memory accesses are mitigating this nicely, though
- Lack of an MMU (*MMU is now built, and in test*)
  - This prevents the implementation of a proper O/S
  - It also permits rogue programs access to system memory and peripherals

Performance

Digilent Inc.
CMod S6 Board

FPGA:	Spartan 6, LX4
Clock:	80 MHz
CPU:	Not pipelined
LUTs:	2,345/2,400 (98%)
RAM:	4 kW
Flash:	4 MW

Other Peripherals: Serial port, PWM audio controller, GPIO, keypad, 2-line display, and more.

Operating System: Supports a small, preemptive multi-tasking, pipe based Operating System, the ZipOS.

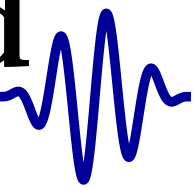
Performance

Xess
XuLA2-LX25

FPGA:	Spartan 6, LX25
Clock:	80 MHz
CPU:	All options on
LUTs:	7,372/15,032 (49%)
RAM:	8 kW
Flash:	256 kW
SDRAM:	8 MW

Other Peripherals: Serial port, PWM audio controller, Flash, SDRAM, SD Card controller, RTC clock, ICAPE, GPIO, and more.

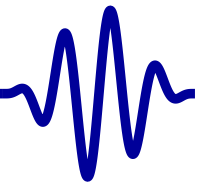
ISA Lessons Learned



- ISAs should be designed with compiling/linking in mind
  - Instructions that the compiler doesn't understand or expect won't get used. (`TEST`, `CMP.`x, `ROL`, `POPC`)

    `ROL` *and* `POPC` *are ripe for repurposing.*
  - The linker needs relocatable JMP and LDI instructions
    1. `LOD (PC),PC`; `Addr`, also known as long jump (`LJMP`)
    2. `BREV #x,Ra`; `LDILO #x,Ra`, created from `LDI #x,Ra`
- GCC reverses branch conditions arbitrarily and at will
- Many programs depend upon 8-bit bytes

Lessons Learned

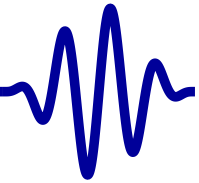


- Fast $\neq$ small LUT count

  LUTs can be used to purchase speed.

- Von Neumann turns a single memory interface a bottle neck
  - The Instruction cache mitigates this problem
- Interrupt handling, though different, is actually quite simple
- BUSERR and ILLegal instruction detection are *necessary*
  - These are marked as optional within the code base
- S/W pipeline scheduling (delayed branching) gets in the way of interrupt handling and debugger step execution
- `MOV Rx,Rx` is not a NOOP, as it might stall waiting for `Rx`

Future



- Memory Mangement Unit, integrating caches
  - Integrated Data cache
  - Integrated (somehow) with the Instruction cache
- Floating Point Unit

  Will handle 32–bit, single precision floats only
- More peripherals
  - Arty's Ethernet
  - DDR3 SDRAM controller, 128–bits/5ns
  - SDIO SD card controller
  - OLEDrgb interface
- Continued work on the ZipOS