CIS 371
Computer Organization and Design

Unit 4: Single-Cycle Datapath

Based on slides by Profs. C.J. Taylor, Amir Roth & Milo Martin
This Unit: Single-Cycle Datapath

- Overview of ISAs
- Datapath storage elements
- MIPS Datapath
- MIPS Control
Readings

- P&H
  - Sections 4.1 – 4.4
Recall from CIS240...
Applications (Firefox, iTunes, Skype, Word, Google)

• Run on hardware ... but how?
240 Review: I/O

- Apps interact with us & each other via I/O (input/output)
  - With us: display, sound, keyboard, mouse, touch-screen, camera
  - With each other: disk, network (wired or wireless)
  - Most I/O proper is analog-digital and domain of EE
  - I/O devices present rest of computer a digital interface (1s and 0s)
240 Review: OS

- **I/O (& other services) provided by OS (operating system)**
  - A super-app with privileged access to all hardware
  - Abstracts away a lot of the nastiness of hardware
  - Virtualizes hardware to isolate programs from one another
    - Each application is oblivious to presence of others
    - Simplifies programming, makes system more robust and secure
    - Privilege is key to this
  - Commons OSes are Windows, Linux, MacOS
• App/OS are software ... execute on hardware
• HW/SW interface is **ISA (instruction set architecture)**
  • A “contract” between SW and HW
  • Encourages compatibility, allows SW/HW to evolve independently
  • **Functional definition** of HW storage locations & operations
    • Storage locations: registers, memory
    • Operations: add, multiply, branch, load, store, etc.
• **Precise description** of how to invoke & access them
  • Instructions (bit-patterns hardware interprets as commands)
LC4: a toy ISA you know
- 16-bit ISA (what does this mean?)
- 16-bit insns
- 8 registers (integer)
- ~30 different insns
- Simple OS support

Assembly language
- Human-readable ISA representation

```
.DATA
array .BLKW #100
sum .FILL #0
.CODE
.FALIGN
array_sum
    CONST R5, #0
    LEA R1, array
    LEA R2, sum
array_sum_loop
    LDR R3, R1, #0
    LDR R4, R2, #0
    ADD R4, R3, R4
    STR R4, R2, #0
    ADD R1, R1, #1
    ADD R5, R5, #1
    CMPI R5, #100
    BRn array_sum_loop
```
371 Preview: A Real ISA

- **MIPS**: example of real ISA
  - 32/64-bit operations
  - 32-bit insns
  - 64 registers
    - 32 integer, 32 floating point
  - ~100 different insns
  - Full OS support

Example code is MIPS, but all ISAs are similar at some level

```assembly
.data
array: .space 100
sum:  .word 0

.text

array_sum:
li $5, 0
la $1, array
la $2, sum
array_sum_loop:
lw $3, 0($1)
lw $4, 0($2)
add $4, $3, $4
sw $4, 0($2)
addi $1, $1, 1
addi $5, $5, 1
li $6, 100
blt $5, $6, array_sum_loop
```
240 Review: Program Compilation

- **Program** written in a “high-level” programming language
  - C, C++, Java, C#
  - Hierarchical, structured control: loops, functions, conditionals
  - Hierarchical, structured data: scalars, arrays, pointers, structures

- **Compiler**: translates program to **assembly**
  - Parsing and straight-forward translation
  - Compiler also optimizes
  - Compiler itself another application ... who compiled compiler?

```c
int array[100], sum;
void array_sum() {
    for (int i=0; i<100; i++) {
        sum += array[i];
    }
}
```
Assembly language
- Human-readable representation

Machine language
- Machine-readable representation
- 1s and 0s (often displayed in “hex”)

Assembler
- Translates assembly to machine

<table>
<thead>
<tr>
<th>Machine code</th>
<th>Assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>x9A00</td>
<td>CONST R5, #0</td>
</tr>
<tr>
<td>x9200</td>
<td>CONST R1, array</td>
</tr>
<tr>
<td>xD320</td>
<td>HICONST R1, array</td>
</tr>
<tr>
<td>x9464</td>
<td>CONST R2, sum</td>
</tr>
<tr>
<td>xD520</td>
<td>HICONST R2, sum</td>
</tr>
<tr>
<td>x6640</td>
<td>LDR R3, R1, #0</td>
</tr>
<tr>
<td>x6880</td>
<td>LDR R4, R2, #0</td>
</tr>
<tr>
<td>x18C4</td>
<td>ADD R4, R3, R4</td>
</tr>
<tr>
<td>x7880</td>
<td>STR R4, R2, #0</td>
</tr>
<tr>
<td>x1261</td>
<td>ADD R1, R1, #1</td>
</tr>
<tr>
<td>x1BA1</td>
<td>ADD R5, R5, #1</td>
</tr>
<tr>
<td>x2B64</td>
<td>CMPI R5, #100</td>
</tr>
<tr>
<td>x03F8</td>
<td>BRn array_sum_loop</td>
</tr>
</tbody>
</table>
240 Review: Insn Execution Model

- The computer is just finite state machine
  - **Registers** (few of them, but fast)
  - **Memory** (lots of memory, but slower)
  - **Program counter** (next insn to execute)
    - Sometimes called “instruction pointer”

- A computer executes **instructions**
  - **Fetches** next instruction from memory
  - **Decodes** it (figure out what it does)
  - **Reads** its **inputs** (registers & memory)
  - **Executes** it (adds, multiply, etc.)
  - **Write** its **outputs** (registers & memory)
  - **Next insn** (adjust the program counter)

- **Program is just “data in memory”**
  - Makes computers programmable (“universal”)
Role of the Compiler
Compiler Optimizations

• Primarily goal: reduce instruction count
  • Eliminate redundant computation, keep more things in registers
    + Registers are faster, fewer loads/stores
    – An ISA can make this difficult by having too few registers

• But also...
  • Reduce branches and jumps (later)
  • Reduce cache misses (later)
  • Reduce dependences between nearby insns (later)
    – An ISA can make this difficult by having implicit dependences

• How effective are these?
  + Can give 4X performance over unoptimized code
  – Collective wisdom of 40 years (“Proebsting’s Law”): 4% per year
  + Allows higher-level languages to perform adequately (Javascript)
Compiler Optimization Example (LC4)

- **Left:** common sub-expression elimination
  - Remove calculations whose results are already in some register

- **Right:** register allocation
  - Keep temporary in register across statements, avoid stack spill/fill

```
;; temp = *first
LDR R7, R5, #2 ; R7=first
LDR R4, R7, #0
STR R4, R5, #-1

;; *first = *second
LDR R3, R5, #3 ; R3=second
LDR R2, R3, #0
LDR R7, R5, #2 ; redundant
STR R2, R7, #0

;; *second = temp
LDR R4, R5, #-1
LDR R3, R5, #3 ; redundant
STR R4, R3, #0
```

```
;; temp = *first
LDR R7, R5, #2
LDR R4, R7, #0
STR R4, R5, #-1 ; unneeded

;; *first = *second
LDR R3, R5, #3
LDR R2, R3, #0
STR R2, R7, #0 ; unneeded

;; *second = temp
LDR R4, R5, #-1 ; unneeded
STR R4, R3, #0
```
What is an ISA?
What Is An ISA?

- **ISA (instruction set architecture)**
  - A well-defined hardware/software interface
  - The “contract” between software and hardware
    - Functional definition of storage locations & operations
      - Storage locations: registers, memory
      - Operations: add, multiply, branch, load, store, etc
    - Precise description of how to invoke & access them
  - Not in the “contract”: non-functional aspects
    - How operations are implemented
    - Which operations are fast and which are slow and when
    - Which operations take more power and which take less

- Instructions
  - Bit-patterns hardware interprets as commands
  - Instruction → Insn (instruction is too long to write in slides)
A Language Analogy for ISAs

• Communication
  • Person-to-person → software-to-hardware

• Similar structure
  • Narrative → program
  • Sentence → insn
  • Verb → operation (add, multiply, load, branch)
  • Noun → data item (immediate, register value, memory value)
  • Adjective → addressing mode

• Many different languages, many different ISAs
  • Similar basic structure, details differ (sometimes greatly)

• Key differences between languages and ISAs
  • Languages evolve organically, many ambiguities, inconsistencies
  • ISAs are explicitly engineered and extended, unambiguous
LC4 vs Real ISAs

- LC4 has the basic features of a real-world ISAs
  - LC4 lacks a good bit of realism
  - Address size is only 16 bits
  - Only one data type (16-bit signed integer)
  - Little support for system software, none for multiprocessing (later)

- Many real-world ISAs to choose from:
  - Intel x86 (laptops, desktop, and servers)
  - MIPS (used throughout in book)
  - ARM (in all your mobile phones)
  - PowerPC (servers & game consoles)
  - SPARC (servers)
  - Intel’s Itanium
  - Historical: IBM 370, VAX, Alpha, PA-RISC, 68k, ...
Some Key Attributes of ISAs

- **Instruction encoding**
  - Fixed length (16-bit for LC4, 32-bit for MIPS & ARM)
  - Variable length (1 byte to 16 bytes, average of ~3 bytes)

- **Number and type of registers**
  - LC-4 has 8 registers
  - MIPS has 32 “integer” registers and 32 “floating point” registers
  - ARM & x86 both have 16 “integer” regs and 16 “floating point” regs

- **Address space**
  - LC4: 16-bit addresses at 16-bit granularity (128KB total)
  - ARM: 32-bit addresses at 8-bit granularity (4GB total)
  - Modern x86 and ARM64: 64-bit addresses (16 exabytes!)

- **Memory addressing modes**
  - MIPS & LC4: address calculated by “reg+offset”
  - x86 and others have much more complicated addressing modes
ISA Code Examples
Array Sum Loop: LC4

```assembly
.DATA
array .BLKW #100
sum .FILL #0
.CODE
.FALIGN
array_sum
    CONST R5, #0
    LEA R1, array
    LEA R2, sum
L1
    LDR R3, R1, #0
    LDR R4, R2, #0
    ADD R4, R3, R4
    STR R4, R2, #0
    ADD R1, R1, #1
    ADD R5, R5, #1
    CMPI R5, #100
BRn L1
```

```c
int array[100];
int sum;
void array_sum() {
    for (int i=0; i<100; i++)
    {
        sum += array[i];
    }
}
```
Array Sum Loop: LC4 ➔ MIPS

```
.data
array .BLKW #100
sum   .FILL #0

.CODE

.FALIGN
array_sum
  CONST R5, #0
  LEA R1, array
  LEA R2, sum
L1:
  LDR R3, R1, #0
  LDR R4, R2, #0
  ADD R4, R3, R4
  STR R4, R2, #0
  ADD R1, R1, #1
  ADD R5, R5, #1
  CMPI R5, #100
  BRn L1
```

MIPS (right) similar to LC4

```
.data
array: .space 100
sum:   .word 0

.text
array_sum:
  li $5, 0
  la $1, array
  la $2, sum
L1:
  lw $3, 0($1)
  lw $4, 0($2)
  add $4, $3, $4
  sw $4, 0($2)
  addi $1, $1, 1
  addi $5, $5, 1
  li $6, 100
  blt $5, $6, L1
```

Syntactic differences:
- register names begin with $
- immediates are un-prefixed

Only simple addressing modes syntax: displacement(reg)

Left-most register is generally destination register
Array Sum Loop: LC4 → x86

.x86 (right) is different

Syntactic differences:
- register names begin with `%`
- immediates begin with `$`

%rbp is base (frame) pointer

Many addressing modes

CIS 371  |  Dr. Benedict J. Brown  |  ISAs & Single Cycle  | 25
x86 Operand Model

- x86 uses explicit accumulators
  - Both register and memory
  - Distinguished by addressing mode

Two operand insns
(right-most is typically source & destination)

Register accumulator: %eax = %eax + %edx

Memory accumulator:
Memory[%rbp-4] = Memory[%rbp-4] + 1

```
.LFE2
  .comm array, 400, 32
  .comm sum, 4, 4

.globl array_sum
array_sum:
  movl $0, -4(%rbp)

.L1:
  movl -4(%rbp), %eax
  movl array(,%eax,4), %edx
  movl sum(%rip), %eax
  addl %edx, %eax
  movl %eax, sum(%rip)
  addl $1, -4(%rbp)
  cmpl $99, -4(%rbp)
  jle .L1
```
Implementing an ISA
Implementing an ISA

- **Datapath**: performs computation (registers, ALUs, etc.)
  - ISA specific: can implement every insn (single-cycle: in one pass!)
- **Control**: determines which computation is performed
  - Routes data through datapath (which regs, which ALU op)
- **Fetch**: get insn, translate opcode into control
- **Fetch** → **Decode** → **Execute** “cycle”
Two Types of Components

- **Purely combinational**: stateless computation
  - ALUs, muxes, control
  - Arbitrary Boolean functions
- **Combinational/sequential**: storage
  - PC, insn/data memories, register file
  - Internally contain some combinational components
Example Datapath
MIPS Datapath
Unified vs Split Memory Architecture

- **Unified architecture**: unified insn/data memory
- **“Harvard” architecture**: split insn/data memories
Datapath for MIPS ISA

• MIPS: 32-bit instructions, registers are $0, $2... $31

• Consider only the following instructions

  add $1,$2,$3      $1 = $2 + $3       (add)
  addi $1,$2,3      $1 = $2 + 3        (add immed)
  lw $1,4($3)       $1 = Memory[4+$3]  (load)
  sw $1,4($3)       Memory[4+$3] = $1  (store)
  beq $1,$2,PC_relative_target  (branch equal)
  j absolute_target         (unconditional jump)

• Why only these?
  • Most other instructions are the same from datapath viewpoint
  • The one’s that aren’t are left for you to figure out
### MIPS machine language

<table>
<thead>
<tr>
<th>Name</th>
<th>Format</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>R</td>
<td>0 18</td>
<td>19 17 0 32</td>
</tr>
<tr>
<td>sub</td>
<td>R</td>
<td>0 18</td>
<td>19 17 0 34</td>
</tr>
<tr>
<td>addi</td>
<td>I</td>
<td>8 18</td>
<td>17 100</td>
</tr>
<tr>
<td>lw</td>
<td>I</td>
<td>35 18</td>
<td>17 100</td>
</tr>
<tr>
<td>sw</td>
<td>I</td>
<td>43 18</td>
<td>17 100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field size</th>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>R</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
</tr>
<tr>
<td>I-format</td>
<td>I</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>address</td>
</tr>
</tbody>
</table>

**FIGURE 2.6** MIPS architecture revealed through Section 2.5. The two MIPS instruction formats so far are R and I. The first 16 bits are the same: both contain an op field, giving the base operation; an rs field, giving one of the sources; and the rt field, which specifies the other source operand, except for load word, where it specifies the destination register. R-format divides the last 16 bits into an rd field, specifying the destination register; the shamt field, which Section 2.6 explains; and the funct field, which specifies the specific operation of R-format instructions. I-format combines the last 16 bits into a single address field.
Start With Fetch

- PC and instruction memory (split insn/data architecture, for now)
- A +4 incrementer computes default next instruction PC
- How would Verilog for this look given insn memory as interface?
First Instruction: **add**

- Add register file
- Add arithmetic/logical unit (ALU)
Wire Select in Verilog

• How to rip out individual fields of an insn? **Wire select**

```verilog
wire [31:0] insn;
wire [5:0] op = insn[31:26];
wire [4:0] rs = insn[25:21];
wire [4:0] rt = insn[20:16];
wire [4:0] rd = insn[15:11];
wire [4:0] sh = insn[10:6];
wire [5:0] func = insn[5:0];
```

R-type  | Op(6) | Rs(5) | Rt(5) | Rd(5) | Sh(5) | Func(6)
Second Instruction: \texttt{addi}

- Destination register can now be either Rd or Rt
- Add sign extension unit and mux into second ALU input
Verilog Wire Concatenation

- Recall two Verilog constructs
  - **Wire concatenation**: \{bus0, bus1, \ldots, busn\}
  - **Wire repeat**: \{repeat_x_times\{w0\}\}

- How do you specify sign extension? **Wire concatenation**

```verilog
wire [31:0] insn;
wire [15:0] imm16 = insn[15:0];
wire [31:0] sximm16 = {{16{imm16[15]}}, imm16};
```

| I-type  | Op(6) | Rs(5) | Rt(5) | Immed(16) |
Third Instruction: \textit{lw}

- Add data memory, address is ALU output
- Add register write data mux to select memory output or ALU output
Fourth Instruction: **sw**

- Add path from second input register to data memory data input
Fifth Instruction: **beq**

- Add left shift unit and adder to compute PC-relative branch target
- Add PC input mux to select PC+4 or branch target

\[ \text{PC} \rightarrow \text{Insn Mem} \rightarrow \text{Register File} \rightarrow \text{Data Mem} \]

\[ \text{Op(6)} \quad \text{Rs(5)} \quad \text{Rt(5)} \quad \text{Immed(16)} \]
Another Use of Wire Concatenation

- How do you do $\ll 2$? **Wire concatenation**

  ```
  wire [31:0] insn;
  wire [25:0] imm26 = insn[25:0]
  wire [31:0] imm26_shifted_by_2 = {4'b0000, imm26, 2'b00};
  ```
Sixth Instruction: \( j \)

- Add shifter to compute left shift of 26-bit immediate
- Add additional PC input mux for jump target
MIPS Control
What Is Control?

- 8 signals control flow of data through this datapath
  - MUX selectors, or register/memory write enable signals
  - A real datapath has 300-500 control signals
Example: Control for add
Example: Control for *sw*

- Difference between *sw* and *add* is 5 signals
- 3 if you don’t count the X (don’t care) signals
Example: Control for \texttt{beq}

- Difference between \texttt{sw} and \texttt{beq} is only 4 signals
How Is Control Implemented?
Implementing Control

• Each instruction has a unique set of control signals
  • Most are function of opcode
  • Some may be encoded in the instruction itself
    • E.g., the ALUop signal is some portion of the MIPS Func field
      + Simplifies controller implementation
  • Requires careful ISA design
Control Implementation: ROM

- **ROM (read only memory):** like a RAM but unwritable
  - Bits in data words are control signals
  - Lines indexed by opcode
  - Example: ROM control for 6-instruction MIPS datapath
  - X is “don’t care”

<table>
<thead>
<tr>
<th>opcode</th>
<th>BR</th>
<th>JP</th>
<th>ALUinB</th>
<th>ALUop</th>
<th>DMwe</th>
<th>Rwe</th>
<th>Rdst</th>
<th>Rwd</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>addi</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>beq</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>j</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Control Implementation: Logic

- Real machines have 100+ insns 300+ control signals
  - 30,000+ control bits (~4KB)
    - Not huge, but hard to make faster than datapath (important!)
- Alternative: **logic gates** or “random logic” (unstructured)
  - Exploits the observation: many signals have few 1s or few 0s
  - Example: random logic control for 6-in insn MIPS datapath

![Diagram showing control implementation logic]

- `add`, `addi`, `lw`, `sw`, `beq`, `j`
- `BR`, `JP`, `DMwe`, `Rwe`, `Rwd`, `Rdst`, `ALUop`, `ALUinB`
Control Logic in Verilog

wire [31:0] insn;
wire [5:0] func = insn[5:0]
wire [5:0] opcode = insn[31:26];
wire is_add = ((opcode == 6’h00) & (func == 6’h20));
wire is_addi = (opcode == 6’h0F);
wire is_lw = (opcode == 6’h23);
wire is_sw = (opcode == 6’h2A);
wire ALUinB = is_addi | is_lw | is_sw;
wire Rwe = is_add | is_addi | is_lw;
wire Rwd = is_lw;
wire Rdst = ~is_add;
wire DMwe = is_sw;
Datapath Storage Elements
### Register File

- **Register file**: M N-bit storage words
  - Multiplexed input/output: data buses write/read “random” word
- **“Port”**: set of buses for accessing a random word in array
  - Data bus (N-bits) + address bus ($\log_2 M$-bits) + optional WE bit
  - P ports = P parallel and independent accesses
- **MIPS integer register file**
  - 32 32-bit words, two read ports + one write port (why?)
Decoder

• **Decoder**: converts binary integer to “1-hot” representation
  - Binary representation of $0...2^N-1$: $N$ bits
  - 1 hot representation of $0...2^N-1$: $2^N$ bits
    - J represented as $J^{th}$ bit 1, all other bits zero
  - Example below: 2-to-4 decoder
module decoder_2_to_4 (binary_in, onehot_out);
    input [1:0] binary_in;
    output [3:0] onehot_out;
    assign onehot_out[0] = (~binary_in[0] & ~binary_in[1]);
    assign onehot_out[1] = (~binary_in[0] & binary_in[1]);
    assign onehot_out[2] = (binary_in[0] & ~binary_in[1]);
    assign onehot_out[3] = (binary_in[0] & binary_in[1]);
endmodule

• Is there a simpler way?
module decoder_2_to_4 (binary_in, onehot_out);
  input [1:0] binary_in;
  output [3:0] onehot_out;
  assign onehot_out[0] = (binary_in == 2’d0);
  assign onehot_out[1] = (binary_in == 2’d1);
  assign onehot_out[2] = (binary_in == 2’d2);
  assign onehot_out[3] = (binary_in == 2’d3);
endmodule

• How is “a == b” implemented for vectors?
  • \(~|(a ^ b)\) (this is a “nor” reduction of bitwise “a xor b”)
  • When one of the inputs to “==” is a constant
    • Simplifies to simpler inverter on bits with “one” in constant
    • Exactly what was on previous slide! (apply DeMorgan’s law)
Register File Interface

- **Inputs:**
  - RS1, RS2 (reg. sources to read), RD (reg. destination to write)
  - WE (write enable), RDestVal (value to write)
- **Outputs:** RSrc1Val, RSrc2Val (value of RS1 & RS2 registers)
Register File: Four Registers

- Register file with four registers
Add a Read Port

- Output of each register into 4to1 mux (RSrc1Val)
  - RS1 is select input of RSrc1Val mux
Add Another Read Port

- Output of each register into another 4to1 mux (RSrc2Val)
  - RS2 is select input of RSrc2Val mux
Add a Write Port

- Input RegDestVal into each register
  - Enable only one register’s WE: (Decoded RD) & (WE)
- What if we needed two write ports?
Register File Interface (Verilog)

module regfile4(rs1, rs1val, rs2, rs2val, rd, rdval, we, rst, clk);
    parameter n = 1;
    input [1:0] rs1, rs2, rd;
    input we, rst, clk;
    input [n-1:0] rdval;
    output [n-1:0] rs1val, rs2val;
 ...
endmodule

• Building block modules:
  • module register (out, in, wen, rst, clk);
  • module decoder_2_to_4 (binary_in, onehot_out)
  • module Nbit_mux4to1 (sel, a, b, c, d, out);
module regfile4(rs1, rs1val, rs2, rs2val, rd, rdval, we, rst, clk);
    input [1:0] rs1, rs2, rd;
    input we, rst, clk;
    input [15:0] rdval;
    output [15:0] rs1val, rs2val;
endmodule

• Warning: this code not tested, may contain typos, do not blindly trust!
module regfile4(rs1, rs1val, rs2, rs2val, rd, rdval, we, rst, clk);
    parameter n = 1;
    input [1:0] rs1, rs2, rd;
    input we, rst, clk;
    input [n-1:0] rdval;
    output [n-1:0] rs1val, rs2val;
endmodule

• Warning: this code not tested, may contain typos, do not blindly trust!
module regfile4(rs1, rs1val, rs2, rs2val, rd, rdval, we, rst, clk);
    parameter n = 1;
    input [1:0] rs1, rs2, rd;
    input we, rst, clk;
    input [n-1:0] rdval;
    output [n-1:0] rs1val, rs2val;
    wire [n-1:0] r0v, r1v, r2v, r3v;

Nbit_reg #(n) r0 (r0v, , , rst, clk);
Nbit_reg #(n) r1 (r1v, , , rst, clk);
Nbit_reg #(n) r2 (r2v, , , rst, clk);
Nbit_reg #(n) r3 (r3v, , , rst, clk);

endmodule

• Warning: this code not tested, may contain typos, do not blindly trust!
module regfile4(rs1, rs1val, rs2, rs2val, rd, rdval, we, rst, clk);
  parameter n = 1;
  input [1:0] rs1, rs2, rd;
  input we, rst, clk;
  input [n-1:0] rdval;
  output [n-1:0] rs1val, rs2val;
  wire [n-1:0] r0v, r1v, r2v, r3v;

  Nbit_reg #(n) r0 (r0v, , , rst, clk);
  Nbit_reg #(n) r1 (r1v, , , rst, clk);
  Nbit_reg #(n) r2 (r2v, , , rst, clk);
  Nbit_reg #(n) r3 (r3v, , , rst, clk);
  Nbit_mux4to1 #(n) mux1 (rs1, r0v, r1v, r2v, r3v, rs1val);

endmodule

• Warning: this code not tested, may contain typos, do not blindly trust!
module regfile4(rs1, rs1val, rs2, rs2val, rd, rdval, we, rst, clk);
    parameter n = 1;
    input [1:0] rs1, rs2, rd;
    input we, rst, clk;
    input [n-1:0] rdval;
    output [n-1:0] rs1val, rs2val;
    wire [n-1:0] r0v, r1v, r2v, r3v;

    Nbit_reg #(n) r0 (r0v, , , , , rst, clk);
    Nbit_reg #(n) r1 (r1v, , , , , rst, clk);
    Nbit_reg #(n) r2 (r2v, , , , , rst, clk);
    Nbit_reg #(n) r3 (r3v, , , , , rst, clk);
    Nbit_mux4to1 #(n) mux1 (rs1, r0v, r1v, r2v, r3v, r3v, rs1val);
    Nbit_mux4to1 #(n) mux2 (rs2, r0v, r1v, r2v, r3v, rs2val);
endmodule

• Warning: this code not tested, may contain typos, do not blindly trust!
module regfile4(rs1, rs1val, rs2, rs2val, rd, rdval, we, rst, clk);
  parameter n = 1;
  input [1:0] rs1, rs2, rd;
  input we, rst, clk;
  input [n-1:0] rdval;
  output [n-1:0] rs1val, rs2val;
  wire [n-1:0] r0v, r1v, r2v, r3v;
  wire [3:0] rd_select;
  decoder_2_to_4 dec (rd, rd_select);
  Nbit_reg #(n) r0 (r0v, rdval, rd_select[0] & we, rst, clk);
  Nbit_reg #(n) r1 (r1v, rdval, rd_select[1] & we, rst, clk);
  Nbit_reg #(n) r2 (r2v, rdval, rd_select[2] & we, rst, clk);
  Nbit_reg #(n) r3 (r3v, rdval, rd_select[3] & we, rst, clk);
  Nbit_mux4to1 #(n) mux1 (rs1, r0v, r1v, r2v, r3v, rs1val);
  Nbit_mux4to1 #(n) mux2 (rs2, r0v, r1v, r2v, r3v, rs2val);
endmodule

- Warning: this code not tested, may contain typos, do not blindly trust!
module regfile4(rs1, rs1val, rs2, rs2val, rd, rdval, we, rst, clk);
  parameter n = 1;
  input [1:0] rs1, rs2, rd;
  input we, rst, clk;
  input [n-1:0] rdval;
  output [n-1:0] rs1val, rs2val;
  wire [n-1:0] r0v, r1v, r2v, r3v;

  Nbit_reg #(n) r0 (r0v, rdval, (rd == 2`d0) & we, rst, clk);
  Nbit_reg #(n) r1 (r1v, rdval, (rd == 2`d1) & we, rst, clk);
  Nbit_reg #(n) r2 (r2v, rdval, (rd == 2`d2) & we, rst, clk);
  Nbit_reg #(n) r3 (r3v, rdval, (rd == 2`d3) & we, rst, clk);
  Nbit_mux4to1 #(n) mux1 (rs1, r0v, r1v, r2v, r3v, rs1val);
  Nbit_mux4to1 #(n) mux2 (rs2, r0v, r1v, r2v, r3v, rs2val);
endmodule

• Warning: this code not tested, may contain typos, do not blindly trust!
Another Useful Component: Memory

- Register file: $M \times N$-bit storage words
  - Few words ($< 256$), many ports, dedicated read and write ports
- **Memory**: $M \times N$-bit storage words, yet not a register file
  - Many words ($> 1024$), few ports ($1, 2$), shared read/write ports
- Leads to different implementation choices
  - Lots of circuit tricks and such
  - Larger memories typically only 6 transistors per bit
- In Verilog? We’ll give you the code for large memories
Single-Cycle Performance
Single-Cycle Datapath Performance

- One cycle per instruction (CPI)
- **Clock cycle time proportional to worst-case logic delay**
  - In this datapath: insn fetch, decode, register read, ALU, data memory access, write register
  - Can we do better?
Foreshadowing: Pipelined Datapath

- Split datapath into multiple stages
  - Assembly line analogy
  - 5 stages results in up to 5x clock & performance improvement
Summary

- Overview of ISAs
- Datapath storage elements
- MIPS Datapath
- MIPS Control