MIPS Architecture Reference - Edited for CS 141

Lukasz Strozek

December 9, 2005

Based on Hennessy and Patterson's Computer Organization and Design

1 Introduction to MIPS

MIPS is a RISC-like architecture (but MIPS is also the name of a microprocessor implementing that architecture): that is, its instructions are simple and execute fast, yet the architecture is immensely powerful.

In the final three weeks of the semester you will implement a fully functional MIPS datapath. Your datapath will implement most of the instructions from the original MIPS R2000. We will skip floating point instructions, most of the system calls and simplify exception handling, but beyond that we're implementing full MIPS.

At first glance, MIPS may seem simplistic, but it's a very powerful architecture. If you take CS 161, you will implement a Unix-like operating system for this very architecture. There exist cross-compilers building executables that can be run on MIPS, as well as a variety of other tools whose existence confirms the fact that MIPS is a real architecture that is used all over the world.

This document will guide you through the design of the MIPS datapath. It contains everything you need to know about MIPS and as such is a full specification of the system that you're asked to design. You're welcome to use other sources to help you explain the functionality of MIPS, but in case of any differences in content, treat this document as the official specification.

So what exactly is a datapath? A datapath is really a fairly complicated finite state machine. Its sole purpose is to execute **machine code** (which is a binary representation of the program to be run). For this execution, a datapath has at its disposal a bunch of **registers** (that is, a small amount of easy-to-access memory) and external **data memory** (that is, an array of data that might take some time to access). Its execution is as follows: it keeps a counter (called a **program counter**, PC) pointing to the currently executed instruction; fetches that instruction from the **code memory** (which is different from data memory), decodes the instruction (*i.e.* figures out what the instruction is supposed to do), executes it, possibly changing the values of the registers, the memory, or the program counter. If the PC doesn't change as a result of executing the instruction, it is advanced to the next instruction. The datapath may also interact with external devices, though we'll only implement a limited set of such operations.

Now that we see the "big picture," let's focus on the components of a MIPS datapath.

2 MIPS basics

MIPS is a 32-bit datapath: every instruction is 32 bits wide, and data comes in "words" which are also 32 bits wide. Memory in MIPS, however, is addressed in bytes (so address `0x00000010` refers to the sixteenth byte – *i.e.* the beginning of the fifth word). Fortunately, all code and data is **aligned** in memory, that is, every datum has an address which is a multiple of four. This simplifies the datapath design a lot, though you have to keep in mind that every address refers to the byte, not the entry of the physical memory.

MIPS is a load-store architecture: that is, the only instructions that access memory are `lw` and `sw`. This, again, simplifies our design a lot and makes efficient pipelining possible (more about this later).

MIPS comes in two flavors: big-endian and little-endian. We will be implementing big-endian flavor. This means that if a word `0x01234567` is stored in memory starting at byte `0x00000010`, the contents of memory are as follows:

Address (byte)	Value
`0x00000010`	`0x01`
`0x00000011`	`0x23`
`0x00000012`	`0x45`
`0x00000013`	`0x67`

3 Memory organization

MIPS differentiates between physical memory (that is, the memory that is actually available in hardware) and virtual memory. Our physical memory consists of two modules: the 128kB of read-only Flash memory (which suffices for 32768 instructions) and the 4kB of data memory (which suffices for 1024 data entries). MIPS abstracts from physical memory by creating a single memory space (called **virtual memory**) and mapping addresses in that memory space to actual physical memory.

In our implementation, the virtual memory is very simple. Code begins at virtual address `0x00400000`, which is mapped to address `0x00000` of the Flash memory. Subsequent addresses are mapped linearly, so that virtual address `0x00400020` corresponds to address `0x00020` of the Flash memory, *etc.*. Obviously, since we only have 128kB of Flash memory, virtual addresses `0x00420000` and up are invalid. Similarly, virtual addresses below `0x00400000` are also invalid. See the Exceptions section below for how to treat invalid addresses.

Data begins at virtual address `0x10000000` and grows in the direction of increasing virtual addresses (this data is called **dynamic data** because the machine doesn't know how much of it will be used at runtime). In MIPS, there is also a concept of **stack** – that is, data that starts just below virtual address `0x80000000` and grows in the direction of decreasing virtual addresses. We would like to support the stack, so our mapping will be as follows:

Virtual address `0x10000000` maps to address `0x000` of the on-chip RAM, and subsequent addresses are mapped linearly. Virtual address `0x7fffffff` maps to address `0xfff` of the on-chip RAM, and preceding addresses are mapped linearly. For example:

Virtual address	Physical address
`0x10000000`	`0x000`
`0x10000001`	`0x001`
⋮	⋮
`0x10000fff`	`0xfff`
`0x7ffff000`	`0x000`
`0x7ffff001`	`0x001`
⋮	⋮
`0x7fffffff`	`0xfff`

Note that one physical address has two corresponding virtual addresses – because depending on which grows more, a particular physical address may belong to either dynamic data or stack. It can't, however, belong to both – and this is why MIPS includes a `$brk` register. This register contains the upper limit of the dynamic data segment and thus limits your stack. If the stack attempts to reference memory at or exceeding this limit, your datapath will throw an exception. For example, suppose that `$brk` is equal to `0x10000120`.

This means that data between `0x10000000` and `0x1000011f` belongs to dynamic data. This means that memory references between `0x7ffff000` and `0x7ffff11f` are now invalid.

4 MIPS Register Set

Registers are a small set of fast memory that the datapath has available at its disposal for most immediate operations. All registers are 32-bit wide. MIPS contains thirty two user registers (that is, registers that the user can access/use in the assembly program) and four special-purpose registers that are hidden from the user.

The user registers are numbered 0 through 31. Their names (to be used in assembly programs) are:

Number	Name	Notes
0	`$zero`	Hard-wired to `0x00000000`
1	`$at`	Used by assembler in expanding pseudoinstructions
2–3	`$v0`, `$v1`	
4–7	`$a0` ... `$a3`	
8–15, 24, 25	`$t0` ... `$t9`	
16–23	`$s0` ... `$s7`	
26–27	`$k0`, `$k1`	
28	`$gp`	Set to `0x10008000` on startup
29	`$sp`	Set to `0x80000000` on startup
30	`$fp`	
31	`$ra`	Modified by some MIPS instructions

More detailed explanation of the conventions surrounding the registers can be found in sections below.

Register 0 is hard-wired to `0x00000000` – writing to it does nothing to it (it is important to note that attempting to write to register 0 is *not* an error and does not throw any exceptions).

Register 1 should be used with caution. The assembler may overwrite the value of register 1 when it expands pseudoinstructions – so to make sure the value of this register is preserved, the programmer should never invoke a pseudoinstruction, a directive or a pseudo-addressing scheme between uses of this register.

Registers 28 and 29 are unique in that they are set to non-zero values when the datapath is reset. They are useful in referencing memory whose virtual address is too high to reach through an offset itself. The programmer should not modify them in user programs.

Register 31 is written to by some MIPS instructions (`bgezal`, `bltzal`, `jal`).

In addition to user registers, MIPS features four special registers. They cannot be explicitly accessed in assembly programs, though there exist instructions that operate on them.

- `$lo` is a special register which is used to store the quotient of the `div` operation, and the low 32 bits of the product in the `mult` operation. `mflo` is an instruction which can access this register. It's initially set to 0
- `$hi` is a special register which is used to store the remainder of the `div` operation, and the high 32 bits of the product in the `mult` operation. `mfhi` is an instruction which can access this register. It's initially set to 0
- `$ex` is a special register which is a bitmap of exceptions which have occurred. A particular bit of this register is set to 1 when a specific exception occurs. More about the exceptions in the section below. `mfex`, `mtex`, `bex` and `bnex` are instructions which can access this register. It is initially set to `0x00000000`
- `$brk` is a special register which contains the upper limit on the current data segment. It can be altered through a special system call (see section on system calls). It is responsible for determining the range of virtual addresses that map to dynamic data. It's initially set to `0x10000000`

5 Exceptions

It is possible that an instruction fails to execute properly. There are a number of reasons that could cause an error in the execution. MIPS identifies those reasons and throws an **exception**. In our implementation, exception handling is very simplified. We introduce a special register, `$ex` , which is a bitmap of possible exceptions. The bits and their associated exceptions are listed in the table below:

Bit	Name	Terminates	Completes	Notes
0	OV	No	Yes	Addition/subtraction overflows
1	DZ	No	No	`div` divides by zero
2	IV	No	No	Invalid virtual address – thrown if load/store tries to address an invalid data virtual address
3	IP	Yes	—	Invalid PC – outside of code segment/unaligned
4	SO	No	No	Stack overflow – stack moves into data segment
5	UA	No	No	Unaligned memory – address not a multiple of 4
6	UN	No	No	Unimplemented feature

A "terminating" exception causes the datapath to enter a halt state – stop execution and enter an infinite loop until it's reset. A "completing" exception allows the datapath to complete the instruction. If an exception is not completing, the instruction at which it occurs does not complete (the datapath acts as if the instruction was a `nop`).

For example, OV is nonterminating and completing – an overflow is a rather mild error (sometimes it's desirable), and so the addition or subtraction that caused the exception should still write the result to the destination register. IP is a terminating exception, because once the PC moves outside of the code segment, the execution cannot continue.

When an exception happens, the respective bit of `$ex` is set to one and stays one. It is up to the programmer to reset `$ex` after the exception occurs. Instructions `bex` and `bnex` can be used to conditionally branch on a particular exception (or a set of exceptions). These instructions take a mask of bits. If any bit is 1, the corresponding exception becomes a condition for branching. For example,

```
add $t0, $t1, $t2
bex 0x002, exceptionHandler
mtex $zero
```

jumps to `exceptionHandler` if the add overflows.

6 MIPS Instruction Set

This section describes in detail all the MIPS instructions. For each instruction, its mnemonic syntax is given, together with the encoding and notes. The following conventions were used:

- `rs` and `rt` are source registers – the datapath should fetch their values whenever they are used. Source registers are usually treated as twos-complement signed 32-bit numbers. In some special cases they are treated as unsigned numbers (the note that follows explains such circumstances)
- `rd` is the destination register – the datapath will write the result to that register number
- `imm` is a 16-bit immediate value. immediate values may either be treated as signed or unsigned values, and may either be zero-extended (in which case the padding bits are all zero), or sign extended (in which case the padding bits are all equal to the most significant bit of the immediate value). For example, an immediate value `0x54df` = 0101010011011111_2 is sign-extended to a 32-bit value `0x000054df` (because the msb of `0x54df` is 0), but an immediate value `0x94df` = 1001010011011111_2 is sign-extended to `0xffff94df` = $11111111111111111001010011011111_2$
- `amt` is an unsigned immediate value used in bitshifts, specifying the amount to shift

- `offset` is a 16-bit signed constant. See the note on branches below for more details
- `target` is an unsigned 26-bit constant. See the note on jumps below for more details
- `offset(rs)` is calculated as (the value of the register `rs`) + `offset` , where offset is a signed 16-bit constant

`add rd, rs, rt`

0	rs	rt	rd	0	0x20	
31	25	20	15	10	5	0

add `rs` to `rt` and store result in `rd`
in case of overflow, throw OV exception

`addi rd, rs, imm`

0x08	rs	rd	imm			
31	25	20	15	10	5	0

add `rs` to sign-extended immediate and store result in `rd`
in case of overflow, throw OV exception

`and rd, rs, rt`

0	rs	rt	rd	0	0x24	
31	25	20	15	10	5	0

logically AND `rs` and `rt` and store result in `rd`

`andi rd, rs, imm`

0x0c	rs	rd	imm			
31	25	20	15	10	5	0

logically AND `rs` and zero-extended immediate and store result in `rd`

`div rs, rt`

0	rs	rt	0		0x1a	
31	25	20	15	10	5	0

divide `rs` by `rt` (both registers treated as unsigned) and store quotient in `$lo` and remainder in `$hi` – see note on division below
in case of division by zero, throw DZ exception

`mult rs, rt`

0	rs	rt	0		0x18	
31	25	20	15	10	5	0

multiply `rs` by `rt` (both registers treated as unsigned) and store the 64-bit result in `$hi` and `$lo`

`nor rd, rs, rt`

0	rs	rt	rd	0	0x27	
31	25	20	15	10	5	0

logically NOR `rs` and `rt` and store result in `rd` `or rd, rs, rt`

0	rs	rt	rd	0	0x25	
31	25	20	15	10	5	0

logically OR `rs` and `rt` and store result in `rd`

`ori rd, rs, imm`

0x0d	rs	rd	imm			
31	25	20	15	10	5	0

logically OR `rs` and zero-extended immediate and store result in `rd`

`sll rd, rt, amt`

0		rt	rd	amt	0	
31	25	20	15	10	5	0

shift `rt` by `amt` (unsigned) bits to the left, shifting in zeroes, and store result in `rd`

`sllv rd, rt, rs`

0	rs	rt	rd	0	0x04	
31	25	20	15	10	5	0

shift `rt` by `rs` (unsigned) bits to the left, shifting in zeroes, and store result in `rd`

`sra rd, rt, amt`

0		rt	rd	amt	0x03	
31	25	20	15	10	5	0

shift `rt` by `amt` (unsigned) bits to the right, shifting in the sign bit, and store result in `rd` – see note on shifts below

`srav rd, rt, rs`

0	rs	rt	rd	0	0x07	
31	25	20	15	10	5	0

shift `rt` by `rs` (unsigned) bits to the right, shifting in the sign bit, and store result in `rd` – see note on shifts

`srl rd, rt, amt`

0		rt	rd	amt	0x02	
31	25	20	15	10	5	0

shift `rt` by `amt` (unsigned) bits to the right, shifting in zeroes, and store result in `rd` – see note on shifts

`srlv rd, rt, rs`

0	rs	rt	rd	0	0x06	
31	25	20	15	10	5	0

shift `rt` by `rs` (unsigned) bits to the right, shifting in zeroes, and store result in `rd` – see note on shifts `sub rd, rs, rt`

0	rs	rt	rd	0	0x22	
31	25	20	15	10	5	0

subtract `rt` from `rs` and store result in `rd` (`rd` = `rs` − `rt`)
in case of underflow, throw OV exception

`xor rd, rs, rt`

0	rs	rt	rd	0	0x26	
31	25	20	15	10	5	0

logically XOR `rs` and `rt` and store result in `rd`

`xori rd, rs, imm`

0x0e	rs	rd	imm			
31	25	20	15	10	5	0

logically XOR `rs` and zero-extended immediate and store result in `rd`

`lui rd, imm`

0x0f	0	rd	imm			
31	25	20	15	10	5	0

load immediate into upper 16 bits of `rd` ; fill lower 16 bits will zeroes

`slt rd, rs, rt`

0	rs	rt	rd	0	0x2a	
31	25	20	15	10	5	0

set `rd` = 1 if `rs` < `rt` and 0 otherwise

`slti rd, rs, imm`

0x0a	rs	rd	imm			
31	25	20	15	10	5	0

set `rd` = 1 if `rs` < signed immediate and 0 otherwise

`beq rs, rt, offset`

0x04	rs	rt	offset			
31	25	20	15	10	5	0

branch to `offset` if `rs` = `rt`

`bgez rs, offset`

0x01	rs	0x01	offset			
31	25	20	15	10	5	0

branch to `offset` if `rs` ⩾ 0

`bgezal rs, offset`

0x01	rs	0x11	offset			
31	25	20	15	10	5	0

branch to `offset` if `rs` ⩾ 0 storing address of next instruction in `$ra` `bgtz rs, offset`

0x07	rs	0	offset
31	25	20	15 10 5 0

branch to `offset` if `rs` > 0

`blez rs, offset`

0x06	rs	0	offset
31	25	20	15 10 5 0

branch to `offset` if `rs` $\leqslant 0$

`bltz rs, offset`

0x01	rs	0	offset
31	25	20	15 10 5 0

branch to `offset` if `rs` < 0

`bltzal rs, offset`

0x01	rs	0x10	offset
31	25	20	15 10 5 0

branch to `offset` if `rs` < 0 and store address of next instruction in `$ra`

`bne rs, rt, offset`

0x05	rs	rt	offset
31	25	20	15 10 5 0

branch to `offset` if `rs` $\neq$ `rt`

`bex mask, offset`

0x18	mask	offset
31	25 20	15 10 5 0

branch to `offset` if all exceptions specified with the mask occurred (doesn't exist in MIPS R2000)

`bnex mask, offset`

0x19	mask	offset
31	25 20	15 10 5 0

branch to `offset` if no exceptions specified with the mask occurred (doesn't exist in MIPS R2000)

`j target`

0x02	target
31	25 20 15 10 5 0

unconditional (near) jump to `target` (unsigned): set bits 27 to 2 inclusive of the next PC to `target` – see note on jumps below `jal target`

0x03	target					
31	25	20	15	10	5	0

near jump to `target` , storing address of next instruction in `$ra`
see note on jumps

`jr rs`

0	rs	0			0x08	
31	25	20	15	10	5	0

unconditionally jump to address `rs` (unsigned) – see note on jumps

`lw rd, offset(rs)`

0x23	rs	rd	offset			
31	25	20	15	10	5	0

load the 32-bit data at address given by value of `rs` + `offset` into `rd`

`sw rt, offset(rs)`

0x2b	rs	rt	offset			
31	25	20	15	10	5	0

store the value of `rt` at address given by value of `rs` + `offset`

`mfhi rd`

0			rd	0	0x10	
31	25	20	15	10	5	0

move contents of `$hi` register to `rd`

`mflo rd`

0			rd	0	0x12	
31	25	20	15	10	5	0

move contents of `$lo` register to `rd`

`mfex rd`

0			rd	0	0x14	
31	25	20	15	10	5	0

move contents of `$ex` register to `rd` (doesn't exist in MIPS R2000)

`mtex rs`

0	rs	0			0x15	
31	25	20	15	10	5	0

move contents of `rs` register to `$ex` (doesn't exist in MIPS R2000)

`syscall`

0					0x0c	
31	25	20	15	10	5	0

execute a system call

The following instructions are called **pseudoinstructions** because they don't have a binary encoding. They are processed in assembly time and replaced with sequences of MIPS instructions. Thus, you don't have to implement them, but you can use them in your assembly programs – the assembler takes care of them. One word of caution, though – some translations require an additional register. The assembler uses `$at` for this purpose. Hence, if `$at` was assigned a value before a pseudoinstruction was invoked, it may have a different value after this instruction is executed. A cautious programmer should never use `$at` in his/her programs.

`abs rd, rs`	store the absolute value of `rs` in `rd`
`div rd, rs, rt`	divide `rs` by `rt` and store the quotient in `rd`
`mul rd, rs, rt`	multiply `rs` by `rt` and store the low 32 bits of the product in `rd`
`neg rd, rs`	store the (arithmetic) negation of `rs` in `rd`
`not rd, rs`	store the logical negation of `rs` in `rd`
`rem rd, rs, rt`	divide `rs` by `rt` and store the remainder in `rd`
`rol rd, rt, rs`	rotate `rt` by `rs` (unsigned) bits to the left and store result in `rd`
`ror rd, rt, rs`	rotate `rt` by `rs` (unsigned) bits to the right and store result in `rd`
`li rd, imm`	load the 32-bit immediate into register `rd`
`seq rd, rs, rt`	set `rd` to 1 if `rs` $=$ `rt` and 0 otherwise
`sge rd, rs, rt`	set `rd` to 1 if `rs` $\geqslant$ `rt` and 0 otherwise
`sgt rd, rs, rt`	set `rd` to 1 if `rs` $>$ `rt` and 0 otherwise
`sle rd, rs, rt`	set `rd` to 1 if `rs` $\leqslant$ `rt` and 0 otherwise
`sne rd, rs, rt`	set `rd` to 1 if `rs` $\neq$ `rt` and 0 otherwise
`beqz rs, offset`	branch by `offset` if `rs` $= 0$
`bge rs, rt, offset`	branch by `offset` if `rs` $\geqslant$ `rt`
`bgt rs, rt, offset`	branch by `offset` if `rs` $>$ `rt`
`ble rs, rt, offset`	branch by `offset` if `rs` $\leqslant$ `rt`
`blt rs, rt, offset`	branch by `offset` if `rs` $<$ `rt`
`bnez rs, offset`	branch to `offset` if `rs` $\neq 0$
`la rd, target`	compute address of `target` and store in `rd`
`move rd, rs`	move contents of `rs` to `rd`
`mfpc rd`	store address of next instruction in `rd` (doesn't exist in MIPS R2000)
`nop`	do nothing – the encoding of this instruction is `0x00000000`, which is equivalent to `sll $zero, $zero, 0` (which has no side effects)

Note: division and negative arguments

You will be asked to implement signed division in assembly. MIPS doesn't specify what the remainder should be when either operand is negative. For the sake of our implementation, the quotient when a is divided by b is $\lfloor a/b \rfloor$. Similarly, the remainder satisfies $0 \leqslant r < b$ when b is positive and $b < r \leqslant 0$ when b is negative. In all cases, $db + r = a$. For example,

$$\begin{aligned} 13/3 &= 4\ r1 & \because\ & 4 \cdot 3 + 1 = 13 \\ -13/3 &= -5\ r2 & \because\ & -5 \cdot 3 + 2 = -13 \\ 13/(-3) &= -5\ r(-2) & \because\ & -5 \cdot (-3) - 2 = 13 \\ -13/(-3) &= 4\ r(-1) & \because\ & 4 \cdot (-3) - 1 = -13 \end{aligned}$$

Note: shifts

Shifts are also somewhat complicated. While shifting left is easy – the least significant bits (bits shifted in) are zeroes, shifting right has two "flavors." An arithmetic shift shifts in the sign bit (*i.e.* repeats what was previously the most significant bit). For example, $-6 >> 1 = -3$.

Logical shift shifts in zeroes, not the sign bit. This logically makes sense (since the same happens when shifting left), but causes apparently wrong results. For instance, $-6 >> 1 = 125$.

Either type of shift does what is expected for positive numbers.

Also, note that some left shifts will be "mathematically" incorrect due to the bit width limitation: for example `0x40000010<<1 = 0x80000020` where `0x40000010 = 1073741840` yet `0x80000020 = -2147483616`.

Note: branches

Branches take offsets in instructions – the number of instructions between the *next* executed instruction and the target instruction (so that branching offset of 0 means no jump at all). Each offset is a signed immediate value. The MIPS assembler allows the use of labels as offsets, in which case the assembler computes the offset and replaces the label with an immediate value. It is considered bad style to use hard-wired numbers as offsets, since the assembler, in expanding pseudoinstructions, may change the number of instructions between any two instructions. You should always use labels when branching.

Any branch or a jump may throw the IP exception.

Note: jumps

Jumps take targets (absolute memory locations). The assembler allows the use of labels as targets, in which case it computes the address of the label and replaces the label with the appropriate numerical value. Target is a 26-bit value, which means only near jumps are allowed: the upper four bits of the next PC are unchanged in a jump. Fortunately, in our implementation, all code resides between `0x00400000` and `0x0fffffff`. If executing code in higher addresses were possible, for example, to jump from a location `0x00100000` to a location `0x1a000000`, you would have to jump twice – once, to `0x0ffffffc` (the boundary of the first segment), then to the right location within the second segment.

Similarly, you should use labels when jumping, unless you are absolutely certain that the program is located at a specific location in memory (by using the `.text addr` attribute, for example) – remember that the address of a particular instruction is very difficult to "guess" since the assembler expands pseudoinstructions.

Note: loads and stores

Loads and stores take offset/register pairs. Offset must be a signed immediate number.

A load or a store may throw the IV, or SO exception, depending on whether addressed memory is simply unmapped, or mapped to a different segment (more specifically, when a stack tries to access dynamic data segment).

7 System Calls

MIPS defines a number of high-level specialized functions which can be called through the `syscall` instruction.

System calls are identified by their number (which should be loaded in register `$v0`). Optional arguments should be passed through registers `$a0` through `$a3`. System calls that return a value may do so through register `$v0`.

System calls (and some complex instructions) put the datapath in a "blocking" state – they may take an indefinite amount of time to execute. You might need a flag that tells you that the datapath is currently executing a system call and should not proceed with its regular execution cycle.

The following table shows all system calls that we will implement:

Syscall number	Name	Arguments	Result
0x01	`print_int`	`$a0` = integer	
0x05	`read_int`		integer in `$v0`
0x09	`sbrk`	`$a0` = amount	starting address in `$v0`
0x0a	`exit`		
0x12	`transmit`	`$a0` = integer	
0x13	`receive`		integer in `$v0`

Here's a detailed explanation of each of the system calls:

- `print_int` displays the value of the register `$a0` on the LED. The number is displayed as eight consecutive nibbles (four-bit chunks), from the most significant one to the least significant one. In between any two nibbles, the datapath is waiting until the user presses and depresses the `strobe` input (this way the user can see the entire number regardless of the speed of the datapath clock
- `read_int` collects eight four-bit values from input `in[3:0]` and concatenates them making them into one 32-bit value which is then sent to `$v0`
- `sbrk` increases the value of `$brk` by the amount specified in `$a0` and returns the old value of `$brk` in `$v0`. If no more memory is available (`$brk-0x10000000` is more than size of data memory), `$brk` would move below `0x10000000` or at or above `0x80000000`, or the amount is not a multiple of 4, the value of -1 should be returned
- `exit` puts the datapath in a halt state (the same state that a terminating exception puts the datapath in)
- `transmit` sends a 32-bit value together with a clocking signal through the serial port of the Xilinx. The "serial port" is simply two bidirectional pins of the Xilinx board. On transmit, one of the pins becomes driven by the clock for 32 cycles, during which all bits of the value to transmit are send through the other pin (on the rising clock edge). After those 32 cycles the datapath continues operation
- `receive` blocks the datapath until the clock pin on the first pin goes up, upon which the datapath collects 32 bits from the other pin and assembles a 32-bit value from them, storing it in `$v0`

8 MIPS Instruction Execution

First, a few notes. The datapath's state consists of: data memory (initially zeroed), user registers and special registers (all set to initial values), and the program counter (initially set to `0x00400000`). Upon asserting `reset`, the datapath should return to this PC. We will not require that the datapath reset its memory, but we will require it reset the registers to their initial value. All of the datapath's internal latches should be reset as well.

The datapath's toplevel signals are: `clock` and `reset`; and `strobe`, `led[6:0]`, `in[3:0]`, `serial_data` and `serial_clock` for syscalls. `led` is an output, the two serial signals are bidirectional, and the other signals are inputs. In addition, all signals wiring the FPGA with the Flash memory need to be declared as toplevel signals.

Remember that all addresses are byte addresses, so every address refers to the byte, not the entry of the physical memory. Since your physical memory implementation accepts entry address, you will have to do some padding to convert byte address to an entry address. It is recommended that you have your virtual memory module perform this conversion.

Since the code memory and the data memory are separate, the virtual memory module should be connected to both memories, and a multiplexer should choose which memory to write to, and which `dataout_s2` should be output. Moreover, the virtual memory module should be passed a flag that specifies whether a code word or a data word is requested. This will allow the module to throw an appropriate exception in case of an error. To reiterate:

- If a code word is requested, the incoming address `addr` should be between `0x00400000` and `0x00420000` exclusive. The resulting physical memory has address

  ```
  addr & 0x000fffff
  ```

  (you might want to make some of these constants parameters in verilog for portability). If `addr` falls outside of this range (or if `addr` is not a multiple of 4), an IP exception should be thrown and the datapath should terminate

- If a data word is requested, `addr` should be between `0x10000000` and the current value of `$brk` exclusive, or between `$sp` and `0x80000000` exclusive. If it's not, an IV exception is be thrown. However, if for any memory reference, `$brk` is greater than

  ```
  $sp & 0x10000fff
  ```

  an SO exception should be thrown (this means that the stack overlaps with the dynamic data)

- With any kind of request, if the address is not a multiple of four, the datapath should not perform the read/write and throw the UA exception

Below is a brief explanation of how MIPS actually executes instructions. You should ensure that your datapath's execution is similar to the one outlined here.

- The first step is to fetch the instruction at the current PC. That means the datapath should convert the PC into a physical address using the virtual memory module described above and issue a request to read code memory at the physical address
- Once the instruction is ready, the datapath uses the opcode fields (see the formats above) to decode the instruction. If an opcode is not supported, the datapath throws the UN exception
- If an opcode is supported, the datapath then fetches the values of registers `rs` and `rt`, if they are used. This can be determined by looking at the opcodes of the decoded instruction
- After all the source registers are fetched, the datapath executes the instruction. `lw` takes an extra few cycles to fetch data from memory (again, passing the address through a virtual memory module), more complicated instructions such as `mult` or `div` might take several clock cycles, and `syscall` blocks the datapath for indefinite time, but all the other instructions should execute instantaneously (since they are driven by combinational logic)
- When the instruction is executed, the result is written either to a register, or to `$ra` (`bgezal`, `bltzal`, `jal`), or to a special register (`div`, `mult`, `mtex`) or to memory (`sw` – with address converted to physical address)
- If any exception occurred, `$ex` might also be written to
- the PC is then incremented (or incremented by an offset from a branch, or set to a target from a jump) – again, this will happen instantaneously since the new PC will be known at the end of execution

9 MIPS Assembler, MIPS Simulator and Executable Files[†]

An **assembly program** is a text file containing a sequence of MIPS instructions. MIPS expects at most one label and instruction per line of the assembly code (either a label or an instruction may be omitted). Any contiguous whitespace is equivalent to a single space. A **label** is any identifier followed with a colon. An **identifier** is any sequence of alphanumeric characters, underscores, and dots that doesn't begin with a digit. Instruction opcodes and assembler directives cannot be used as identifiers.

The syntax for all instructions has already been presented. The only exception is that MIPS assembler allows for a more general load/store addressing mode. The datapath expects a

[†]This section describes the tools you will use to write assembly code. It's not directly relevant to the implementation of your datapath.

pair of offset and the register, but the assembler allows users to specify the address in the following way:

`label` $\pm$ `offset(rs)`

where either `label`, `(rs)`, or `offset(rs)` can be omitted. For example, if `table` is a label, the following are valid:

```
lw $t0, table + 32($t1)
lw $t0, 32
lw $t0, table + 32
lw $t0, table
```

MIPS translates these into sets of instructions (in a similar fashion to how pseudoinstructions are translated), possibly using `$at` .

Anything beginning with a sharp sign until the end of the line is a comment and thus is ignored by the assembler.

Numbers in MIPS are in base 10 by default. If preceded by `0x`, they are interpreted as hexadecimal. All numbers are sign-extended to their required width.

The assembler supports a small subset of **assembler directives**. Directives are instructions which are translated into regular MIPS instructions, so in a way they are like pseudoinstructions, with the exception of irregular syntax. The supported directives are:

- `.data` – stores the instructions/words that follow the directive in the data segment at the current pointer (starting at `0x10000000`). The assembler calls the sbrk syscall first to ensure that the data specified is in a valid memory space
- `.text` [*addr*] – stores the instructions/words that follow in code memory. If *addr* is supplied, the data is put starting at that (word-aligned) address; otherwise, the code is put in the next available address (starting at `0x00400000`)
- `.word` $w_1, \ldots, w_n$ – stores the specified sequence of words in the current segment. w_1 is stored at current address, w_2 is stored after it, and so on up to w_n
- `.space` n – allocate n bytes in the current segment. Nothing is certain about what the initial value of the allocated data will be

These directives should be combined with labels to enable accessing the data (or instructions) stored in such way. For example, the following program initializes four words of data at `0x10000000`, eight words of extra space at `0x10000010`, and some instructions at `0x00400000` and `0x00401000`:

```
    .data
someData:
    .word      0xdeadbeef, 0x01234567, 0xfaceface, 0x89abcdef
someSpace:
    .space     8

    .text
main:
    lw $t0, someData
    lw $t1, someData + 4
    sw $t1, someSpace
    sw $t0, someSpace + 4
    j otherCode

    .text 0x00401000
otherCode:
    li $v0, 0x0a
    syscall
```

Notice that it is also possible to store `.word`s in the code segment. This is not advisable given that this is the code that will be executed (and cannot be read using `lw` since loads/stores can only access data memory).

Using `.data` and `.space` is a preferred way to allocate memory whose size is known at compile time. Bare stores and loads will likely cause an IV exception since `$brk` won't allow unallocated addresses to be accessed. Setting `$brk` to the size of the physical memory will allow you to use all the data without worrying about further calling `$brk` , but then you can't use stack.

CS141 MIPS has a dedicated assembler, `cs141-as`, which takes the assembly program as an argument and produces machine code suitable for use in your MIPS implementation. If a flag `-x` is specified, the code produced is of the same format as the XESS simplified data that can be uploaded to Flash, for easy of uploading.

We also have a dedicated simulator, which executes MIPS machine code. The simulator is called `cs141-sim`. It takes one argument – the machine code to execute, and executes it. The simulator dumps any output that MIPS may have produced (through the `print_int` system call) on `stdout`, and the values of all registers and the data memory on standard error after the program exits or throws a terminating exception. `transmit` and `receive` are not supported in simulation.

10 Advanced MIPS Concepts – Writing Good Assembly[†]

You may wonder what the actual role of those three registers is. By now you should understand that they are useful in performing loads and stores. More precisely, `$gp` (called a "global pointer") is a pointer to the beginning of the data segment. This way, loads and stores can easily access dynamic data through the offset + register convention. Since offset is a 16-bit value, if we didn't have `$gp`, we would have to load 32-bit values to temporary registers and so every load and store would take three instructions. Having a dedicated register helps us speed up this process.

Similarly, `$sp` points to the location of the top of the stack. Stack is useful in function calls, when data needs to be preserved across calls to multiple (nested) functions, or even multiple (recursive) calls to one function. It is possible to do the same using just dynamic data, but we would like the functions themselves to have expandable memory (so they can dynamically allocate memory) and this is how stack comes in handy.

To reiterate, `$brk` is a special register (accessivle only through the sbrk system call) which contains the limit of dynamic memory. It helps determine when stack begins overlapping with dynamic data, and which addresses within the data segment are valid.

Even if you're not writing high-level code, your assembly code must obey a few rules:

- Avoid using `$at` – this register is used by the assembler itself to expand pseudoinstructions
- Avoid assuming anything about the registers' initial values (with the exception of `$zero`, `$gp` and `$sp`) and the initial value of uninitialized memory
- Use labels whenever you can: to access dynamic memory, and branch to instructions. Remember that memory references including labels are transformed into memory references including `$gp`. Never use constants for instruction offsets/locations as the expanded pseudoinstructions may shift instructions around
- Remember that `$ra` is modified by some MIPS instructions
- Remember to clear `$ex` when you are done checking for exceptions
- Pushing items on stack is very easy:

```
sub $sp, $sp, 4
sw $t0, 0($sp)
```

[†]This section describes how to write good assembly code. It's not directly relevant to the implementation of your datapath.

and popping items off the stack can be done with

```
    lw $t0, 0($sp)
    add $sp, $sp, 4
```

Notice that it's necessary to change the value of `$sp` as its value will be used to check for validity of virtual addresses

- Allocating data can be done either in the beginning of your code with

```
    .data
chunk:
    .space 0x10
```

  after which the label `chunk` will point to a chunk of 16 words in the data memory (so calling, say, `sw $t0, chunk` will store the contents of `$t0` in the first word of the allocated space); or during the execution of your program with

```
    li $v0, 0x09
    li $a0, 0x10
    syscall
```

  after which the register `$v0` will contain the address of the first of the sixteen words of newly allocated memory

If you want to implement function calls, it's often useful to create a so-called "frame." A frame is an area of memory that contains the function's current context (saved registers and local variables). To call a function, the calling routine needs to do the following:

- Pass arguments. By convention, the first four arguments are passed in `$a0` – `$a3`. Any extra arguments are pushed on the stack
- Save some registers. By convention, registers `$a0` – `$a3` and `$t0` – `$t9` may be modified by the called function, so if the caller wants to preserve them, it must save their values before the call
- Execute a `jal` instruction. This will jump to the called function, saving the return address in register `$ra`

The function that was called must do the following:

- Allocate memory for the frame (the frame is part of stack). The frame needs to be big enough to fit all registers that it's changing, which (by convention) ought to be preserved across function calls. Those are: `$s0` through `$s7`, `$fp` and `$ra`
- The called function must then set up a frame pointer (register `$fp`). A frame pointer should point to the first entry in the frame
- When the called function returns, it should place the returned value in `$v0`, restore all registers that were saved in the frame, pop the frame and return by jumping to the address in `$ra` . Note that `$ra` is one of the registers that was saved in the frame – so even if its value was overwritten since it was first set up, restoring the frame also restores the value of this register

11 A Note on Pipelining

It is possible to pipeline our implementation of MIPS. As a recap, the execution of the datapath proceeds as follows:

- First, the instruction is fetched
- The fetched instruction is decoded
- After the instruction is decoded, the source registers are read
- In case of a load, data from memory is read; in case of a syscall, the datapath enters the syscall mode (blocks execution until the syscall is done)
- Result is written to memory (or the destination register)

Notice that some components of the datapath are idle on some of those stages. We can **pipeline** the datapath, *i.e.* compress the execution (resulting in a smaller number of clock cycles required to execute an instruction) by shifting the usage of components in time when they are not used.

First, notice that, as we said already, the MIPS instruction set is very regular. In fact, if you look at all the instructions defined above (just core instructions, not pseudoinstructions), you will notice that their encoding follows one of the three formats. These formats are:

`add, and, div, mult, nor, or, sll, sllv, sra, srav, srl,`
`srlv, sub, xor, slt, jr, mfhi, mflo, mfex, mtex, syscall`

0	rs (0)	rt (0)	rd (0)	const	opcode

`addi, andi, ori, xori, lui, slti, beq, bgez,`
`bgezal, bgtz,blez, bltz, bltzal, bne, lw, sw`

opcode	rs (0)	rt (rd, op)	const

`bex, bnex, j, jal`

opcode	const

Notice that since we fetch the high order bits of the instruction first, we know the source register (`rs`, `rt`) numbers even before we fetch the entire instruction! (this is because `rs` and `rt`, if at all, appear in the high 16 bits). Hence, it's a good idea to assume that `rs` and `rt` do appear in the instruction, and start fetching them while the low 16 bits of the instruction are fetched.

With that in mind, we notice that we can compress some execution in time (while the instruction is being fetched, nothing else was happening in the datapath). The list below briefly summarizes how MIPS can be pipelined:

- Once the 16 high bits of the instruction are ready, the register `rs` and `rt` numbers are ready. Even though the registers might actually not appear in the instruction, we assume they do and begin fetching their values. You should think about which register value to fetch first – `rs` or `rt`. While in our datapath, both register values will be fetched before the entire instruction is fetched, it is conceivable that fetching register values might actually take more time. In such case, fetching a register that's likely to be used more first is a good idea
- While the instruction is fetched, the result of the previous instruction could be written to the register (or data memory). Since data and code memory are separate, this process will not interfere with the fetching of the next instruction
- We can go even further by *prefetching* instructions. If we fetch an instruction one word in advance, we will be able to determine whether the current instruction is a load, and if it is, read from data memory without spending extra cycles. Hence, at any stage, the

next instruction will be fetched, the results of the *previous* instruction will be written to registers or memory, and the data will be fetched from memory. There are a few hazards associated with this solution:

- If the current instruction is a branch or a jump, the prefetched instruction is no longer valid (the PC could have changed). In that case you must stall the pipeline – *i.e.* wait with fetching new instruction until the current instruction completes
- Similarly, if the program contains a store immediately followed by a load, the datapath will attempt to execute both the writing to memory and the reading from memory at the same time. Since we can only perform one operation on data memory at once, you will need to stall the datapath in this case as well

12 Parting Words

Now you know everything there is to know about MIPS. This may seem like a daunting task, but it gives you an excellent feeling of fulfillment when you're finally done.

Good luck!