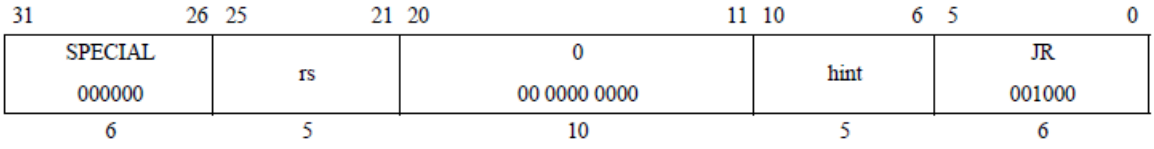
Additional Notes for the ISA

Jump Register (JR) Instruction:

Jump Register JR



Format: `JR rs` MIPS32 (MIPS I)

Purpose:

To execute a branch to an instruction address in a register

Description: `PC ← rs`

Jump to the effective target address in GPR *rs*. Execute the instruction following the jump, in the branch delay slot, before jumping.

For processors that implement the MIPS16 ASE, set the *ISA Mode* bit to the value in GPR *rs* bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one

Restrictions:

The effective target address in GPR *rs* must be naturally-aligned. For processors that do not implement the MIPS16 ASE, if either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched as an instruction. For processors that do implement the MIPS16 ASE, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

At this time the only defined hint field value is 0, which sets default handling of JR. Future versions of the architecture may define additional hint values.

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

```
I:  temp ← GPR[rs]
I+1:if Config1CA = 0 then
        PC ← temp
    else
        PC ← tempGPRLEN-1..1 || 0
        ISAMode ← temp0
    endif
```

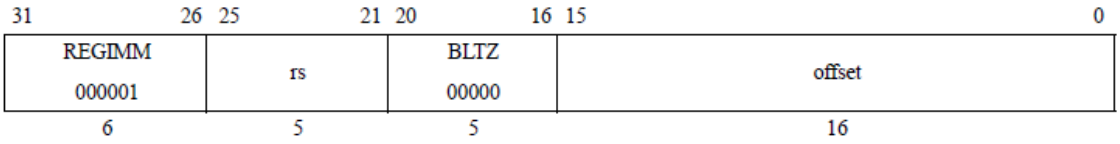
Exceptions:

None

This instruction is an R-Type instruction based on the opcode (000000). However, this instruction has a different flow than all the other R-Type instructions we ask you to implement. Because of this, the Controller needs to know when an R-Type instruction is the JR instruction. To handle this, IR(5 downto 0) should also be sent to the controller. This detail was excluded from the datapath.

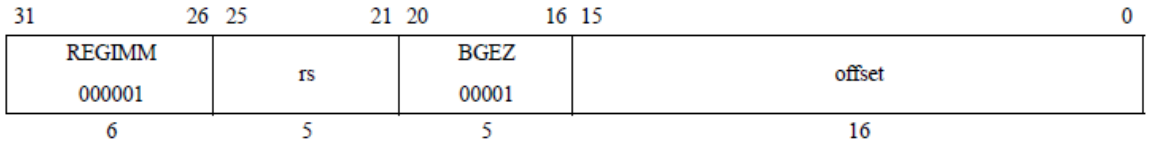
Branch Less Than Zero (BLTZ) / Branch Greater Than or Equal to Zero (BGEZ) Instructions:

Branch on Less Than Zero BLTZ



Format: `BLTZ rs, offset` MIPS32 (MIPS I)

Branch on Greater Than or Equal to Zero BGEZ



Format: `BGEZ rs, offset` MIPS32 (MIPS I)

These two branch instructions have the same opcodes (000001). The only way to differentiate them is using IR(20 downto 16). That information needs to be given to the Controller or the ALU Control so that the correct branch condition is evaluated. This detail was excluded from the datapath.

Fall 2021 Extension of the MIPS project

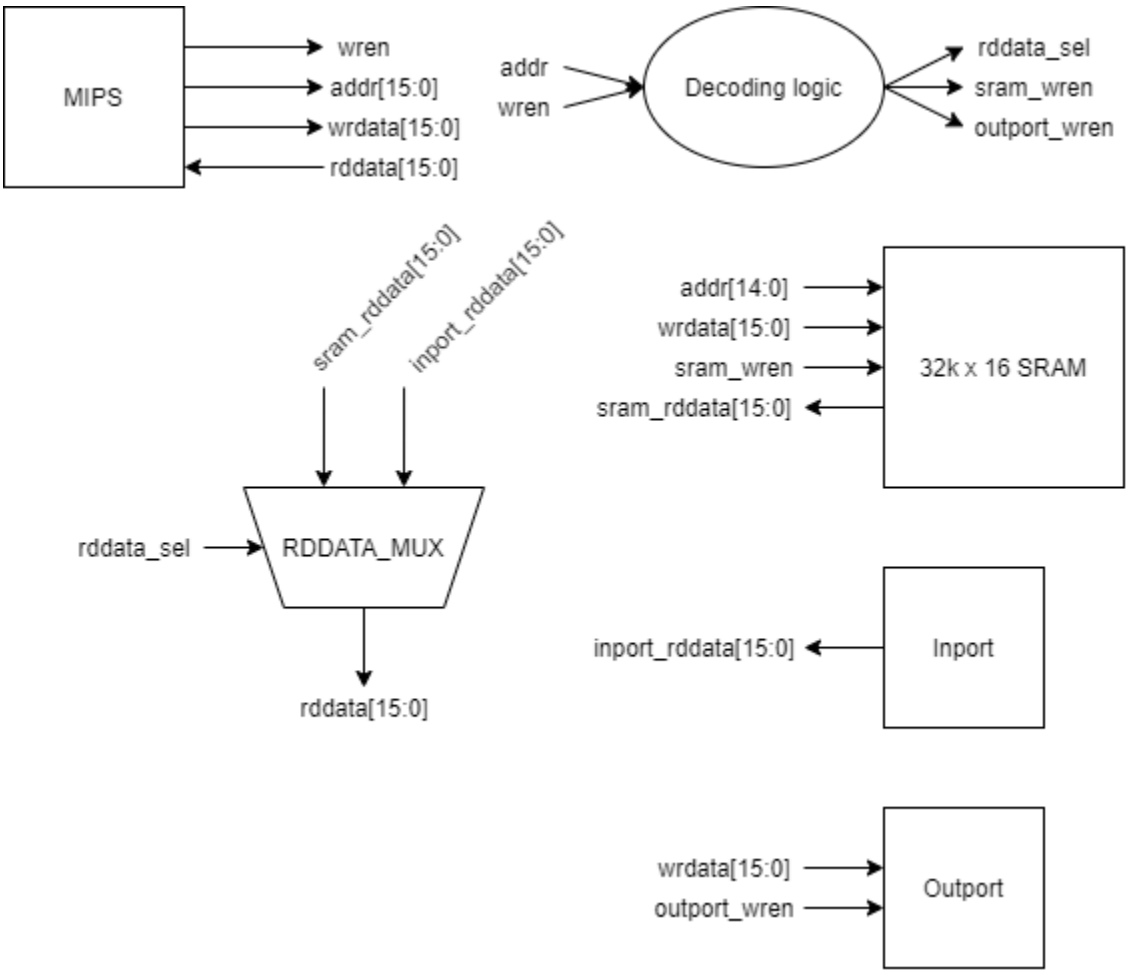
In lab 5, you created a FSMD and FSM+D to implement a factorial calculator. The factorial calculator had 4 relevant Inputs/Outputs: go, n, result, and done. In a similar manner to how we create inport0, inport1, and outport, you will create memory-mapped registers for go, n, result, and done.

How does memory mapping work in general?

Fundamentally, memory mapping involves assigning an address to everything we want to be able to access. We are limited in how many "things" we can put in our memory map based on how many addresses are available == 2^{addr_width}. If you have taken microprocessor applications at UF, the EBI lab includes memory mapping. So, this may sound familiar.

A Quick Example – mapping an SRAM, an input port, and an output port

Assume we have a 16-bit wide address bus, 16-bit wide rddata bus, and a 16-bit wide wrdata bus. We want to add a 32k by 16 SRAM starting at address 0x0000. We want the SRAM to use "full-address-decoding". We want to add a 16-bit wide input port and a 16-bit wide output port. We want the I/O ports to use "partial-address-decoding" with 256 images starting at address 0x8000.



Sample VHDL code to implement the Decoding logic

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity decoding_logic is
	port (
		addr : in std_logic_vector(15 downto 0);
		wren : in std_logic;
		rddata_sel : out std_logic;
		sram_wren : out std_logic;
		outport_wren : out std_logic);
end decoding_logic;

architecture BHV of decoding_logic is
begin
	process(addr, wren)
	begin
		-- Assume we are reading SRAM by default.
		-- If an addr we did not map is read from,
		-- such as 0x8100, we will end up reading
		-- an image of an address in SRAM.
		-- In this case the address is 0x100.
		rddata_sel <= '0';
		-- If we had more rddata sources in our memory map,
		-- then rddata_sel would need to be multiple bits wide.
		-- We also could have a "bad addr" select like
		-- rddata == "11" if we try to read from a location
		-- that has not been memory mapped.

		-- assume we are not writing by default
		sram_wren <= '0';
		outport_wren <= '0';

		-- check if we are accessing the SRAM
		if (unsigned(addr) >= 0x0000 and unsigned(addr) <= 0x7FFF) then
			rddata_sel <= '0'; -- reading from the SRAM
			if (memwrite = '1') then
				sram_wren <= '1'; -- writing to the SRAM
			end if;
		-- check if we are accessing the I/O ports
		elsif (unsigned(addr) >= 0x8000 and unsigned(addr) <= 0x80FF) then
			rddata_sel <= '1'; -- reading from the inport
			if (memwrite = '1') then
				outport_wren <= '1'; -- writing to the outport
			end if;
		end if;
	end process;
end BHV;
```

For more information regarding memory mapping: https://en.wikipedia.org/wiki/Memory-mapped_I/O

How will we interface with our MIPS in our specific case?

For the MIPS project, we are doing "full-address-decoding" for everything we make. We have a 32-bit wide address bus, a 32-bit wide wrddata bus, and a 32-bit wide rddata bus. We want to add a 256 x 32 SRAM starting at address 0x00000000. We want INPORT0 at address 0x0000FFF8 and INPORT1 at address 0x0000FFFC. We want OUTPORT at address 0x0000FFF8 as well. In addition to this, we will also map the GO, N, RESULT, and DONE registers for our factorial calculator.

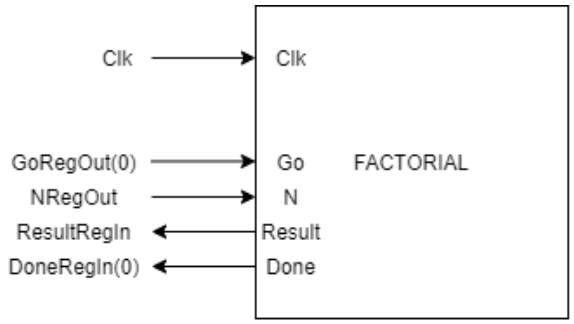
The GO register should be readable and writeable. **The address of GO is 0xFFE8**.
The N address should be readable and writeable. **The address of N is 0xFFEC**.

The reason we want to make GO and N readable is to allow for debugging. You will be able to check with software (the assembly code) what the current value of GO and N are by making these registers readable. From a pure functional perspective, we could make these registers write only. However, you should make them readable for this project.

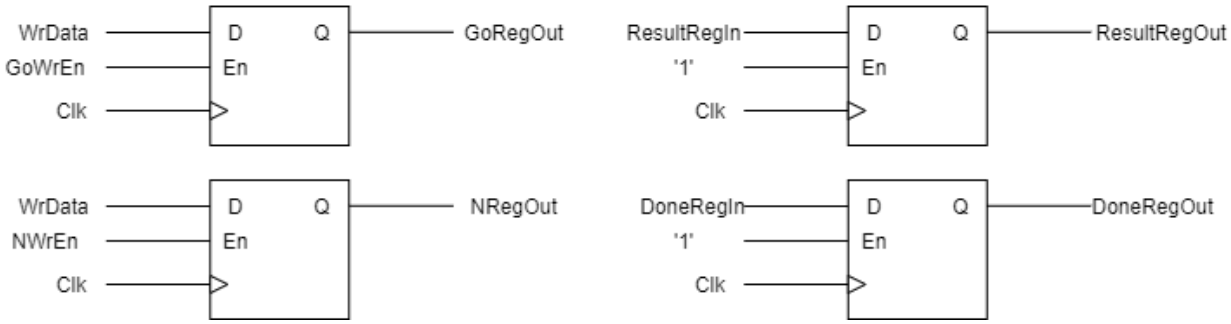
The RESULT register should be READ ONLY. **The address of RESULT is 0xFFF0**.
The DONE register should be READ ONLY. **The address of DONE is 0xFFF4**.

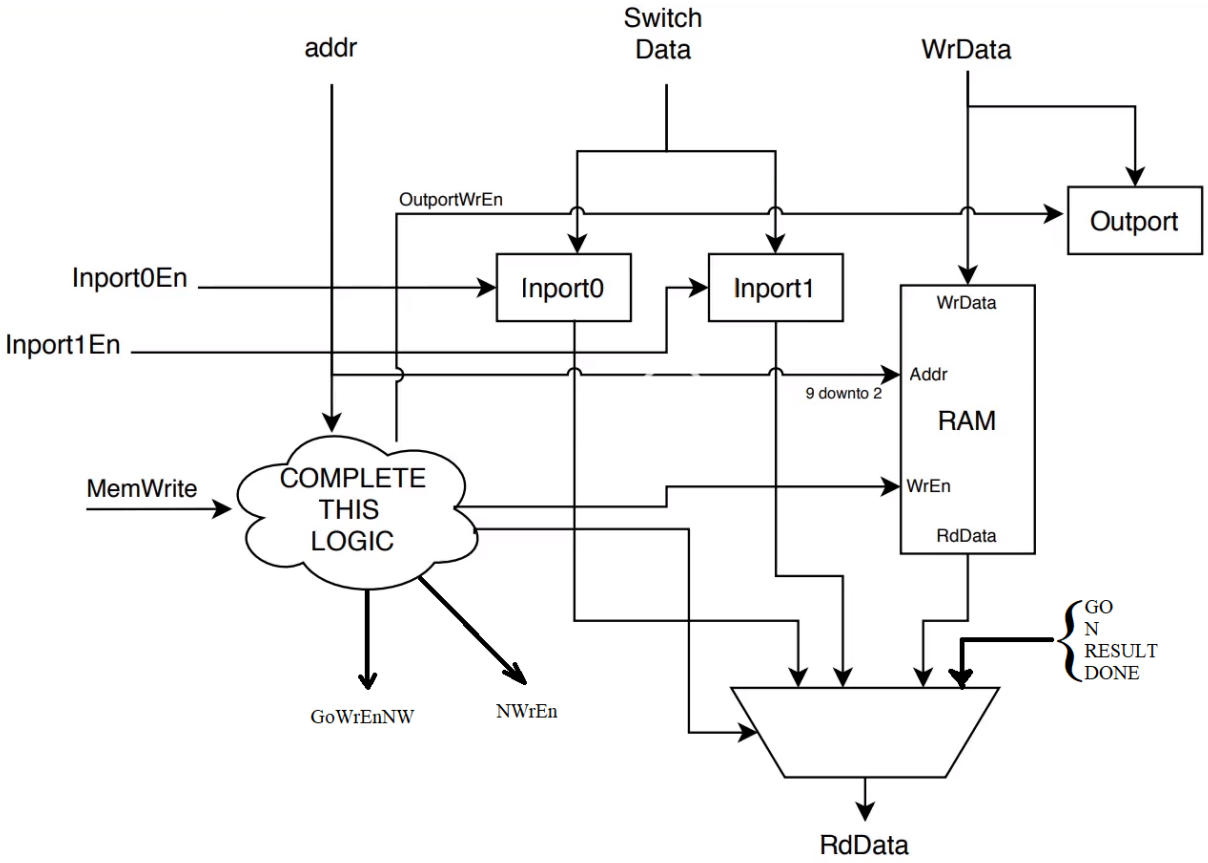
Due to naming conflicts (you make have the same names for modules created for the lab 5 FSM+D as modules for the mips project. For example, you have a datapath module for the mips and also a datapath module for the FSM+D), ***you may find it easier to use the FSMD version of your factorial*** module. However, either version is acceptable.

See below for a block diagram showing the connections between the factorial calculator and the GO, N, RESULT, and DONE registers.



Note: The GO and DONE registers really only need to be a single flip flop. It is up to you how wide you want them to be.





Pictured above is a modified version of the memory map block diagram. To memory map the 4 relevant I/O registers for the factorial module, you will need to modify this block diagram. Specifically, the COMPLETE THIS LOGIC section needs to have additional outputs. It needs to create a GoWrEn and NWrEn. It also needs to generate the select for the RdData mux. The RdData mux currently only takes in Inport0, Inport1, and the RAM output for its inputs. The mux also needs to take in the GO, N, RESULT, and DONE register outputs to make these registers readable as discussed.

Note:
The RAM component we use to implement RAM has a synchronous read architecture. In other words, when we supply an address to the SRAM, the rddata from the SRAM is not available until after the rising edge of the clk. You should therefore be delaying your rddata_sel signal by 1 clock cycle to account for this. (Hint: what component can be used to delay a signal by a clock cycle?)

Known Quartus issues:
If you are getting the following error:

"Error (16031): Current Internal Configuration mode does not support memory initialization or ROM. Select Internal Configuration mode with ERAM."

you can fix it with the following:

Assignments -> Device -> Device and Pin Options -> Configuration -> Configuration Mode: Single uncompressed image with Memory Initialization

There is also an addition to Deliverable 2 from week 1. This addition will verify that your interface with the factorial calculator is working correctly.

Deliverable 2 (10 points): Design and simulation of the memory (RAM, ports, and factorial registers). Create a testbench that demonstrates the following:

Write 0x0A0A0A0A to byte address 0x00000000
Write 0xF0F0F0F0 to byte address 0x00000004
Read from byte address 0x00000000 (should show 0x0A0A0A0A on read data output)
Read from byte address 0x00000001 (should show 0x0A0A0A0A on read data output)
Read from byte address 0x00000004 (should show 0xF0F0F0F0 on read data output)
Read from byte address 0x00000005 (should show 0xF0F0F0F0 on read data output)
Write 0x00001111 to the outport (should see value appear on outport)
Load 0x00010000 into inport 0
Load 0x00000001 into inport 1
Read from inport 0 (should show 0x00010000 on read data output)
Read from inport 1 (should show 0x00000001 on read data output)
Write 0x00000004 to the N reg
Read from the N reg (should show 0x00000004 on read data output)
Write 0x00000001 to the GO reg
Read from the GO reg (should show 0x00000001 on read data output)
Read from the DONE reg until DONE = '1' (wait until read data output = 0x00000001)
Read from the RESULT reg (should show 0x00000018 on read data output)