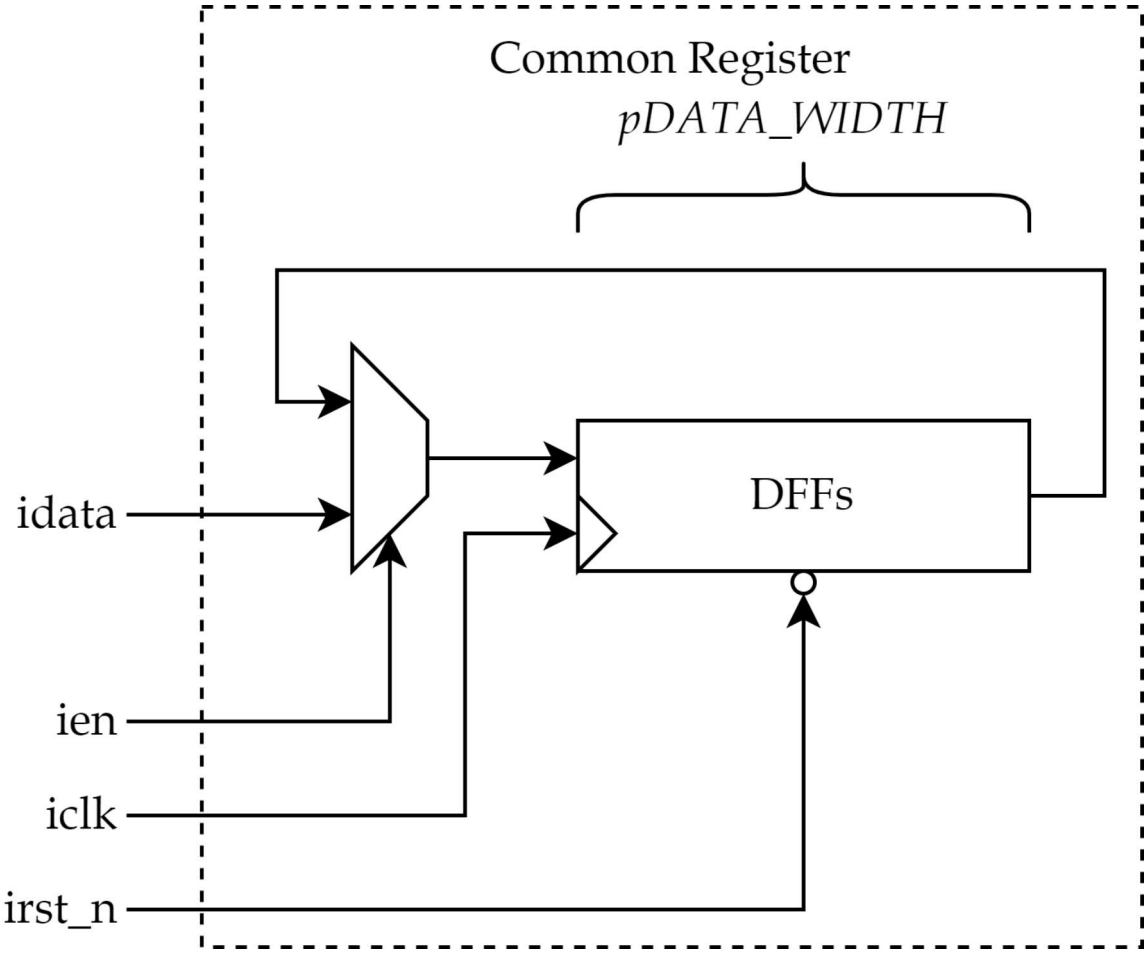
Introduction

Following sections describe the process of designing subblocks. The solutions are just examples, there are possibly different (and even better) solutions to explore.

Common register

As described, *data_bus* is common selected bus that goes around the design as input for multiple blocks. Thus, these blocks must have an enable signal (controlled by CPU) to decide when they should capture.

The design of this register is in below figure.



Normally `irst_n` and `iclk` are not shown in diagram because they are standard pins that all design must have. Other diagrams from now won't include them.

Pin list

Signal	Size	Direction	Description
iclk	1	in	Clock
irst_n	1	in	Asynchronous reset
ien	1	in	Enable register to capture input bus
idata	DATA_WIDTH	in	Data in
odata	DATA_WIDTH	out	Data out. The current value of the register

In this design, we use parameter `pDATA_WIDTH` to make it to be more reusable.

This register is directly used as instruction registers (AIR, DIR) and user register set (R0-R3) because these registers don't have any special functions to be designed in separated modules. See section CPU wrapper to see how this module is instantiated in the top level.

Code: Common Register

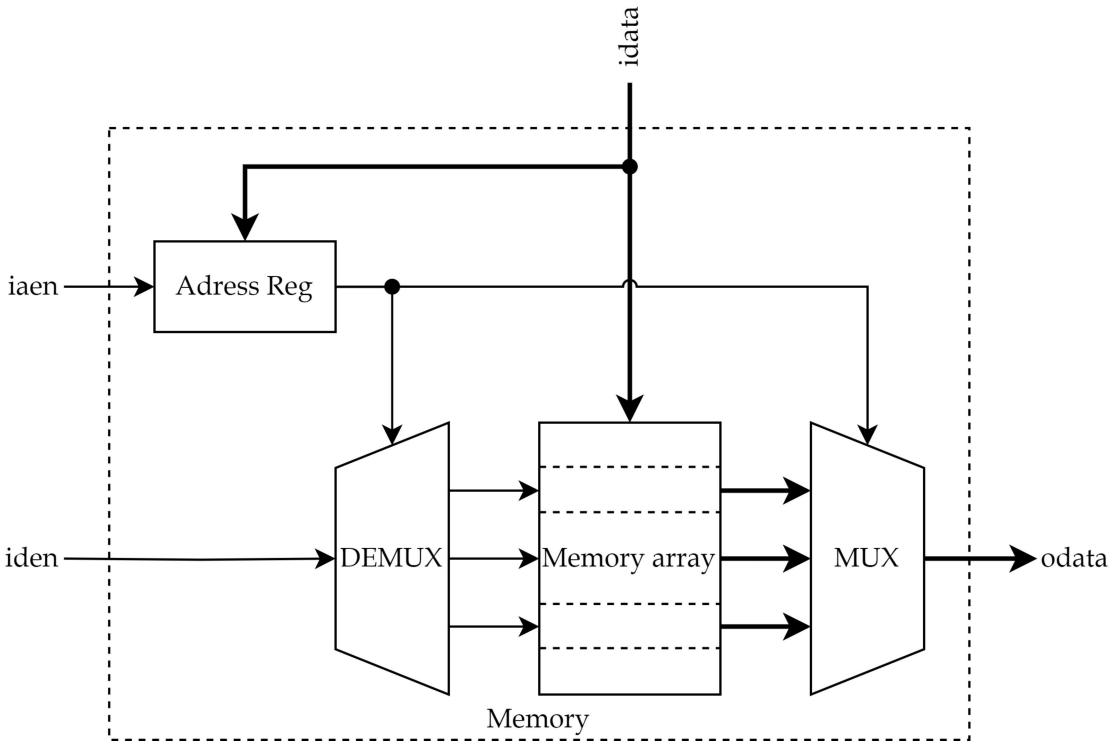
Note: Normally we don't create common cells to be this small as this `common_reg`, it is just used to illustrate how a module can be reused across the design.

Memory array

The memory consists 2 parts: Address and Data.

- Data is Memory array; we need it to hold commands and data for the program.
- Address is the pointer that points to the data we want to process (read or write). It can be either controlled directly from input or captured (hold) in internal registers. In this design, we want Adress to be captured because the input changes as CPU processes command.

The block diagram is shown in below figure.



Pin list

:want

Signal	Size	Direction	Description
iclk	1	in	Clock
irst_n	1	in	Asynchronous reset
iaen	1	in	Enable register to capture address to internal register
iden	1	in	Enable register to capture data to memory array
idata	DATA_WIDTH	in	Data in, use for both address and data

Signal	Size	Direction	Description
odata	DATA_WIDTH	out	Data out. The value of the memory that address is pointing to

The address and data use the same data bus, so we only need 1 input for both registers.

Code: Memory

In the code snippet above, `cpu_pgk` is package file of the whole design that contains various parameters and common data structures.

The signal `rmemory_array` can also be designed using `common_reg`, but this coding style is widely used when implementing a memory, so we use it here.

ALU

The ALU wrapper is divided in to 3 main parts:

- ALU: Calculating unit.
- Accumulator: Store the calculating result.
- Flag: Store the calculated flags from the ALU.

The `ALU` block is purely combinational block thus it needs the latter 2 blocks to store the calculating result.

ALU core

As described before, ALU must support maximum 2 registers at a time, thus it has 2 ports to be as data input. Most operations result in data out (e.g., ADD 2 inputs to get 1 final result) The block must support following functions (ALU command set in CPU's Instruction set):

- ADD: Use 2 data buses.
- AND, OR, XOR, NOT: In these operations, NOT only takes 1 data bus.
- SHIFT LEFT, RIGHT
- COMPARE: This operation aims to get flags (equal or larger), not the data bus out.

Beside the calculating result, it also sends out following flags:

- ZERO
- EQUAL, LARGER: Use in COMPARE command
- CARRY: Carry value for commands (ADD, SHIFT)

The flags are the requirements of the JUMP commands in CPU's Instruction set.

With above requirements, we can have pin list as following table.

Signal	Size	Direction	Description
iopcode	OPCODE_WIDTH	in	The operation code (command type) for the ALU
ira	DATA_WIDTH	in	The data input for RA
irb	DATA_WIDTH	in	The data input for RB
icarry	DATA_WIDTH	in	The carry flag from previous calculation
odata	DATA_WIDTH	out	Data result
oflags	FLAG_WIDTH	out	Flags of the command

This ALU is pure combinational block thus there is no clock signal for it. As a result, it needs the port `ici` as an input to provide the carry value of previous calculating result.

Code: ALU

In this code snippet, we don't define `iopcode` and `oflags` as data bus but `enum` and `struct`. These types are defined in the package `cpu_pkg`. When using these common define, we can reuse the same definitions across design and thus reduce the human erorr when duplicating codes.

Flag

This block simply stores the flags from `ALU` to use in the next instruction. This block also has clear function as requirement from instruction set (`CLF`) thus it can't use the `common_reg` module.

Pin list

Signal	Size	Direction	Description
iclk	1	in	Clock
irst_n	1	in	Asynchronous reset
iflag_clf	1	in	Clear all flags
ien	1	in	Enable register to capture input bus
iflags	FLAG_WIDTH	in	Flag from ALU
oflags	FLAG_WIDTH	out	Flag to CPU, the current value of the register

Code: Flag

Accumulator

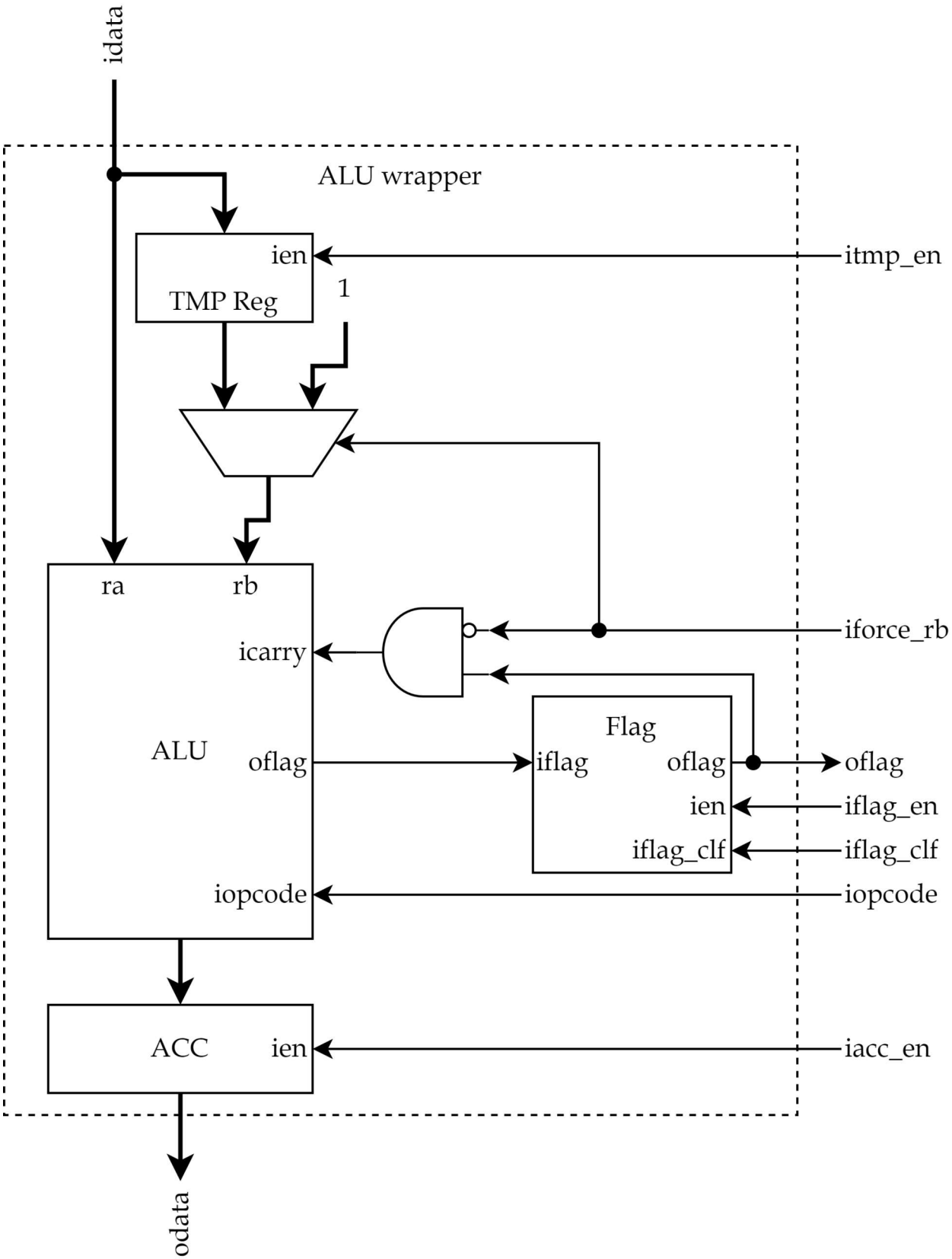
The accumulator is simple register, we can use `common_reg` for it.

ALU wrapper

As we put all modules together, we have 2 problems to solve:

- The common data bus `cpu_bus` only allow 1 output at a time. We can't get data for both `ira` and `irb` at the sametimes for the ALU.
    - This requires us to create a temporary register, that hold data for 1 register, then we move to the next one and do operation for both of them.
    - In this design, we choose temporary register connects to port RB of the ALU.
    - This register can be the module `common_reg`.
- As mention in previous sections, the CPU need to increase instruction register `AIR` to the next address to get the next command. We can use the `ADD` function of the ALU to do so.
    - RA is the current value of `AIR`
    - RB is number 1 (because we want to increase `AIR` step by step). In this case, we will need an input from CPU to tell the ALU wrapper to put number 1 to ALU, regardless the value of the temporary register.
    - This function also requires carry to be always 0 (otherwise `AIR` may jump 2 steps), we will use the same input to force the carry flag in this case to be zero.

From that requirement, we can have the figure below.



Pin list

Signal	Size	Direction	Description
iclk	1	in	Clock

Signal	Size	Direction	Description
irst_n	1	in	Asynchronous reset
itmp_en	1	in	Enable for temporary register
iacc_en	1	in	Enable for accumulator register
iforce_rb	1	in	Force RB to be 1, also force the carry flag to be zero
iflag_clf	1	in	Clear all flags
oflags	FLAG_WIDTH	out	Flags result from the ALU
icarry	1	out	Stored carry flag to the ALU
idata	DATA_WIDTH	in	The data from common bus
odata	DATA_WIDTH	out	Latched data result, value of accumulator register

Code: ALU Wrapper

Instruction register

The instruction register is separated into 2 parts:

- Address (AIR): Store the address of the memory that the CPU wants to access.
- Data (DIR): Store the data that the address points to.

There is no special requirement for this group, thus we can use `common_reg` directly

User register set

There are 4 registers in this set, and just like Instruction register, there is no special requirement for this set.

CPU

The state machine of CPU bases on the requirement, the instruction set and the design of other blocks. We can divide 2 main functions of the CPU:

- Fetch instruction: CPU get the current instruction to operate
- Execute instruction: Bases on the fetched instruction, CPU decides to calculate, jump, etc. and prepares for the next round.

For every CPU step, 2 clocks cycles are required:

- 1st: Select the data source for `cpu_bus`
- 2nd: Enable the destination to capture `cpu_bus`

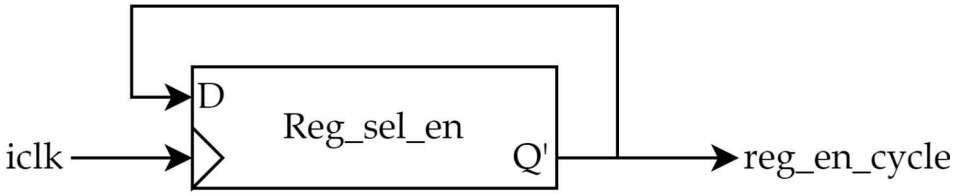
In the FSM sections, we only see the main states while the 2 cycles are not mentioned again.

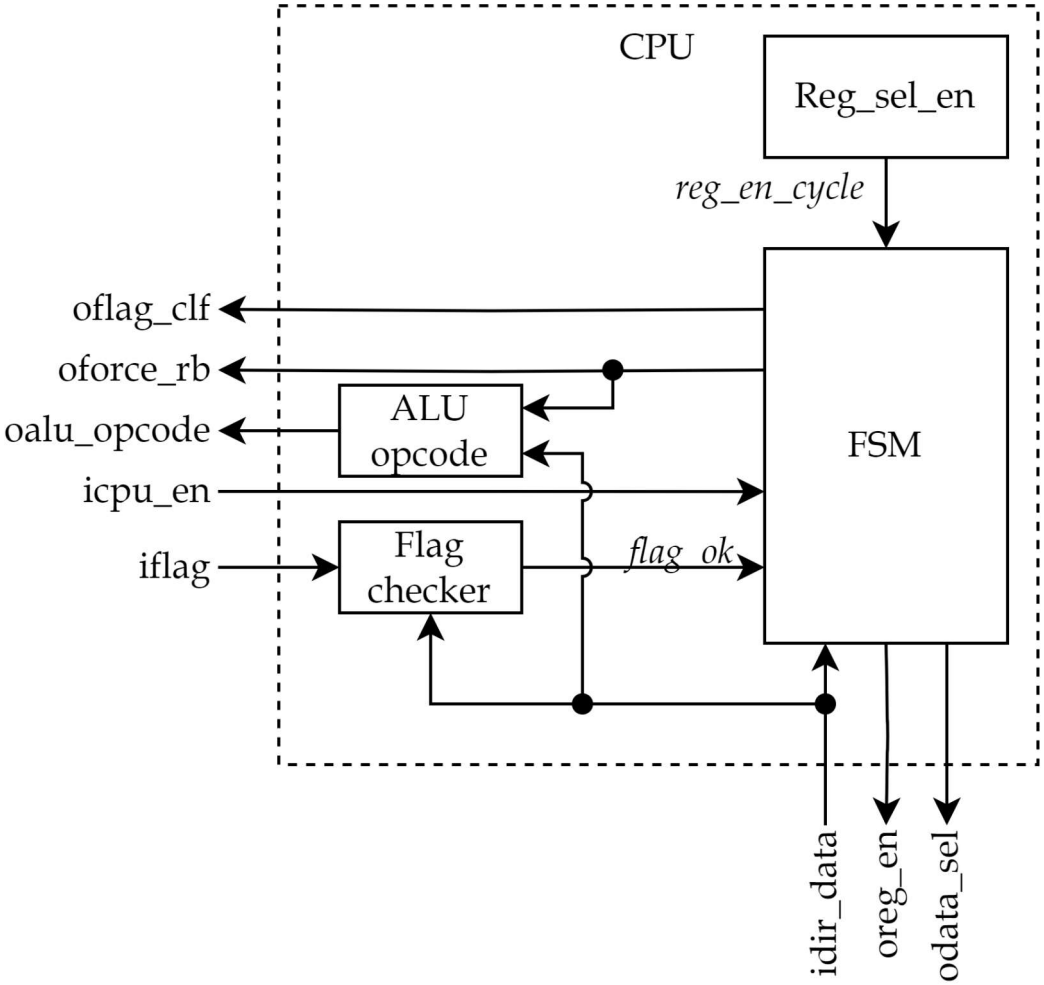
Block design

The main part of this block is FSM, but FSM doesn't need to handle every output by itself.

- FSM: Loop through program & execute instructions.
- Register select/enable: Notifies FSM which cycle is bus selection and register enable.
- ALU decode: Decode the instruction to ALU operation code
- Flag checker: Decode the instruction to enable flag comparison

Figure below shows the CPU block design and how the `Reg_sel_en` block is implemented.





The `reg_en_cycle` is simply a register that flips itself in every clock cycle. With this signal, we can make the FSM to change state whenever this signal asserts to solve the requirement above.

The block *Flag checker* compares the expected flag from DIR and the output from ALU. If they match, signal `flag_ok` is asserted.

ALU decode simply converts the instruction to ALU opcode. This can convert incorrectly when the instruction is not ALU type. However, CPU won't enable ACC in those cases so there is no problem.

NOTE: This CPU will have IDLE state, which is not described in next sections. This state makes sure the CPU stays IDLE if it is disabled, or if it has finished the program (see END instruction)

Pin list

Signal	Size	Direction	Description
iclk	1	in	Clock
irst_n	1	in	Asynchronous reset
icpu_en	1	in	Enable CPU to start executing program
idir_data	DATA_WIDTH	in	The data from DIR
oforce_rb	1	out	Force RB to be 1 in ALU ADD command
oflag_clf	1	out	Clear all flags
oalu_opcode	1	out	ALU OPCODE
odata_sel	CPU_REG_NO_ADR	out	Select data source for `cpu_bus`
oreg_en	CPU_REG_NO	out	ALU OPCODE

FSM Fetch instruction

Following are the steps for this part:

- Select `cpu_bus` to be AIR, enable AMEM (MEM address) to capture the address.
  At the same times, **force** ALU to do ADD 1 (see section ALU wrapper), enable ACC to capture the result. In this operation, carry is forced to zero to avoid AIR jumps 2 steps.
- Select `cpu_bus` to be DMEM (MEM data), enable DIR to capture the data.
- Select `cpu_bus` to be ACC, enable AIR to capture the *next* address.

Step 1 & 2 fetch the instruction from the memory. Step 3 prepares the next address for AIR. In step 1, ACC doesn't take data from `cpu_bus` but the ALU itself, thus we can enable ACC and MEM at the same time. This means we can make both parts can capture data at the same time without worring about data confliction.

FSM Execute instruction

Each command requires different behavior of the CPU to execute. CPU use the bits [7:4] of DIR (`idir_data`) to get command what to operate.

ALU instruction

- Step 1: Select `cpu_bus` to be RB, enable ALU temporary register to capture the value.
- Step 2: Select `cpu_bus` to be RA, send opcode to ALU, enable ACC to capture the result.
- Step 3: Select `cpu_bus` to be ACC, enable RB to capture the value.

NOTE:

- Data in temporary register will not be used in some instruction, but in the meantime, we do the same steps for all operations to reduce the complexity of the code.
- From now on, all steps are shorten to make it easier to read. *Select* means selecting cpu_bus, *Enable* means enabling a block to capture the data on its' input.

LD & ST instruction

LD instruction:

- Step 1: Select RA, enable AMEM
- Step 2: Select DMEM, enable RB
- Step 3: NOP (No operation)

ST instruction:

- Step 1: Select RA, enable AMEM
- Step 2: Select RB, enable DMEM
- Step 3: NOP

DATA instruction: This instruction is different with the previous ones because it requires 2 memory location instead 1. In this instruction, we need the *next* memory address to get the data, so it involves in AIR.

- Step 1: Select AIR, enable AMEM. (AIR has been in the *next* address due to the last step in Fetch phase) At the same time, ALU ADD force 1, enable ACC (This step prepares for AIR increases address one more time because the current one is data)
- Step 2: Select DMEM, enable RB.
- Step 3: Select ACC, enable AIR. (After this step, AIR has jumped 2 steps comparing with previous command)

In this section, we also use NOP state. This means the CPU does nothing. As the LD and ST don't require 3 steps, they don't do anything in this one. We can also reduce the step 3 for those instruction, but to reduce code complexity, we use still use it.

JUMP instruction

JMPR: In this instruction, we won't use the AIR value that is calculated in the Fetch state.

- Step 1: Select RB, enable AIR
- Step 2: NOP
- Step 3: NOP

JMP

- Step 1: Select AIR, enable AMEM.
- Step 2: Select DMEM, enable AIR.
- Step 3: NOP

J*: (Remaining command)

- Step 1: Select AIR, enable AMEM. At the same time, ALU ADD force 1, enable ACC. This operation is required for the next step. Enable flag checking function.
- Step 2: Select DMEM or ACC (depends on flags value and the instruction), enable AIR.
- Step 3: NOP

For the Jump with flags instruction, we will need a flag checking block that use the instruction and the flags' value to decide which source that the AIR need to capture in the Step 2. (The signal `flag_ok` in the CPU block diagram)

This block captures the flag comparison info directly from `idir_data` and sends the signal `flag_ok`. This signal may be incorrectly if the instruction is not Jump type. However, there is no issue because the FSM won't care about `flag_ok` in that case.

Miscellaneous instruction

CLF:

- Step 1: Clear flags
- Step 2: NOP
- Step 3: NOP

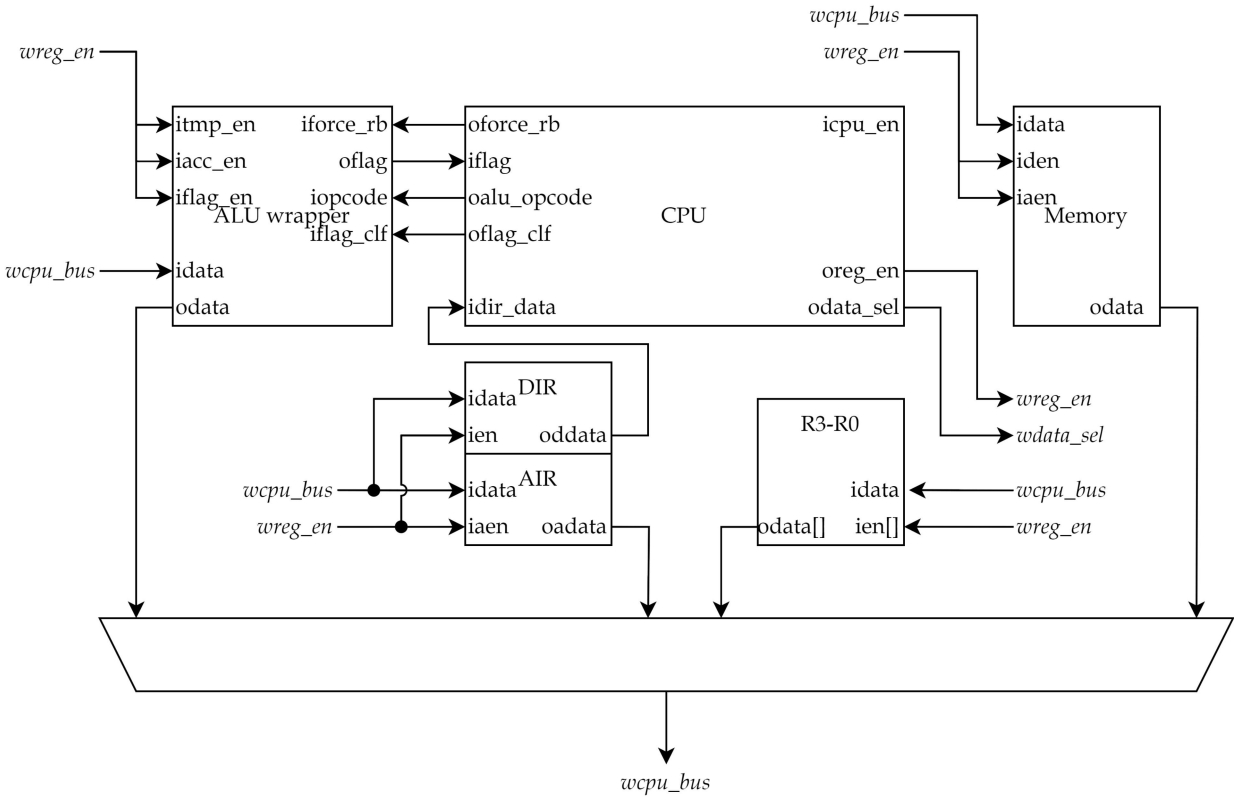
END: NOP in all steps, return to IDLE

CPU code

Code: CPU

CPU wrapper

To put all components into a complete design, we use wrapper for that.



As shown in the figure, the data bus is implemented in this module, the output of this MUX will be connected to all data input of the subblocks.

Code: CPU wrapper

Limitations

This design of course has multiple points to optimize, some examples:

- It is not a full design, where we don't have any method to write program into the memory. We should have ports to control memory from user's side before enabling the CPU. (The simulation will need to *force* the program into the memory because of this.)
- The CPU state machine waste clock cycles for commands that don't requir all 3 execution states.
- The data bus is simply a MUX, it might be a problem for high-speed design.