Practical Assignment – IPC006 Processors

Designing the RUN2223 CPU

Hand in before Monday, March 27th at 07:00, via Brightspace.

In this practical assignment, you will create a design for the RUN2223 CPU based on the specifications in the Practicum Manual, and you will implement this design in Digital. Read the entire assignment before starting with Digital.

1 Assignment

1. Create a high-level design for your CPU (we recommend doing this on paper, not in Digital). Make sure that you have data flow diagrams for each phase of the instruction cycle, in particular for the Fetch phase and the Decode+Execute phase of each of the 6 instruction formats listed in Appendix B of the Practicum Manual. Use these diagrams to decide on the input and output interfaces for each component listed in section 3.2 of the manual *(note that the interface of the Instruction Decoder given in the manual is intentionally incomplete)*. Decide which signals require a multiplexer, and how these will be controlled. Incorporate the feedback you received from your TA on the homework of week 5 ("Data Path") into your design. If you did not do the homework of week 5 yet, you should do so now!

   Note that it is easier to change the interface of your Instruction Decoder at this stage than later on, so it is worth spending time getting this right.

2. Implement the RUN2223 CPU in Digital, according to the description in the Practicum Manual. You must base your implementation on the template files provided in the Practical Assignment module in Brightspace. The Timer and the Register Bank are given, and *may not be changed*. We suggest starting with the ALU, the Tester, and the Flag Register Bank; the Instruction Decoder is often considered more difficult.

   You need to use meaningful names for subcomponents and all input and output ports in your design. Create a neat layout: avoid unnecessary wire-crossings, and use 'Manhattan layout' where possible (only horizontal and vertical wires with 90-degree angles).

   Thoroughly test each individual component by putting test values on the inputs, and checking that the outputs are as expected. You can optionally use the 'Test case' component to automate this. We have provided a test case for the `ADD R1, R2, R3` instruction, which you can use as a template for other tests. Once you are confident that the individual components are working correctly, connect them according to your data flow diagrams. You can test the implementation of the combined CPU by putting machine codes on the **Databus$_{in}$** input and toggling the **Clock** input on and off, which will cause the timer to cycle through the execution phases of the CPU. For instructions that read from memory, put a suitable value on **Databus$_{in}$** during the Decode+Execute phase.

3. When your implementation of the CPU is mostly complete, you can also test its interaction with the rest of the computer. The file `Computer.dig` provides an implementation of a full computer, with all the elements of the Von Neumann-architecture: a data- and address bus, main memory, and I/O devices. Your CPU is included as a subcomponent of this design.

You may optionally create a short assembly language program that demonstrates the functionality of your CPU, preferably with a loop or other nontrivial control flow. This can be translated to machine code using the assembler that will be provided on Brightspace. The resulting `.hex` file can then be loaded into the the RAM component of the computer[1]. However, you should not spend time on this until the basic functionality of your CPU is working properly!

2 Technical Requirements

Your implementation must satisfy the following technical requirements. This is necessary to ensure that you are using registers that are compatible with the design of the RUN2223 CPU, and that it is possible for us to perform automated tests on your processor. We shall provide a tool at a later date which will allow you to verify if these requirements have been met.

Use acceptable components

All memory elements in your processor should be instances of Components > Memory > Register. Do *not* use any of the other components below the Components > Flip-Flops or Memory submenus. The number of data bits of Register is configurable, so it can be used both as a 1-bit Flip-Flop and as a multi-bit register. Additionally, each Register instance must have a short, but descriptive, name attached to it by using the *Label* attribute. For example, we have labeled the CPU registers in the register bank as `R0-R15`. It is also *not* allowed to use lookup tables (Components > Logic > Lookup Table) or any DIL Chips (under Components > Library).

Preserve interfaces from template

We require that you keep the interfaces of the template components intact. This means that you must not change the filenames of the toplevel components (e.g. `InstructionDecoder.dig`). It is allowed to add subcircuits to create a hierarchical design; the filenames of those subcircuits may be chosen freely.

You must also preserve the input and output interfaces of the toplevel components. You may add input and output ports to the interfaces that are incomplete, i.e. the Instruction Decoder, but you must not rename or remove any of the existing ports. The Timer and the Register Bank may not be changed at all. You may not remove or rename Registers that are labeled.

Ensure simulation can be started

Finally, to evaluate your processor we need to be able to perform a simulation on it. Before submitting, you must test that it is possible to start the simulation (by pressing the "play" button) at the level of `CPU.dig`, without any errors. Note that this will not work on the template CPU as provided, because the circuit and its subcomponents are incomplete.

If you want to run the simulation on a circuit that is not yet complete, you can use the following workaround: add constant values to the circuit (Components > Wires > Constant value), and connect these to all unconnected outputs. This makes it possible to test the part of the functionality that *is* implemented. However, this only works if all subcircuits are complete, or at least simulatable.

In case you are unable to implement all of the functionality of the CPU before the submission deadline, you are still required to make sure that the simulation can be started by applying the workaround above. In that case, please (briefly) document the unimplemented functionality in your `README` file.

[1] Via Edit > Circuit specific settings > Advanced > Preload program memory on startup / Program file.

3 Roadmap

As building an entire CPU can be a daunting task, we provide you with a suggested roadmap to follow while working on the assignment. While you are free to deviate from this, we recommend following it as it follows the structure of the course, allowing you to make progress as soon as possible. Do not postpone working on the assignment till you have a full understanding of the entire CPU, i.e. near the end of the course. You will run out of time by doing so. By following the roadmap you can work on pieces in isolation that do not require this full understanding.

1. Start with the Tester. If you followed the introduction to Digital exercise, you already built part of it, and extending it should be relatively straightforward. After the lectures of week 1 (Boolean algebra) and week 2 (Gates and circuits) you should have all the knowledge required to fully implement the Tester by using Table 2 of the CPU manual.

2. Afterwards, move on to the ALU. This too only requires the material of the first two lectures. Use Table 1 and Table 3 of the CPU manual to see how each ALU operation should be implemented.

3. The Flag Register Bank is a fairly simple component that you can implement after lecture 3 (Memory and sequential logic).

4. The Instruction Decoder is without a doubt the most complex part of the CPU, as it effectively acts as the brain of the CPU, instructing how the other components should be used to perform machine code instructions. You should be able to make a start after the Machine code lecture, and combined with the Data path exercises you will have been introduced to all necessary concepts. Use Table 4 of the CPU manual, along with Appendix B, to determine how each machine code instruction should be decoded. See Section 5 for some additional design tips.

5. With all major components completed, all that remains is connecting them together at the CPU top level circuit. Use your data paths created in week 5 to see which connections must be made, and how multiplexers should be used to resolve signal conflicts. The last two weeks discuss assembly programming and pipelining. While this may help you understand the big picture, and allows you to write programs to test your CPU, it does not introduce fundamentally new concepts needed to implement the CPU itself.

We strongly recommend properly testing each component as soon as they are completed. This ensures components are working as intended *before* you combine them into a single CPU, at which point hunting down the origin of bugs becomes much harder.

4 What to hand in

The practical assignment *must* be done in pairs. Hand in the following items before the deadline:

1. The implementation of your processor in Digital.

   Your processor must be submitted as a `.zip` file, which is created as follows: open the CPU component (`CPU.dig`), and choose File > Export > Export to ZIP file. In this way, we ensure that only relevant files are included, and that you receive an error if any files are missing. The filename must contain your student numbers and names, according to this pattern: `RUNCPU_s1234567_Janssen_s7654321_deVries.zip`.

   Your processor must satisfy all the technical requirements, in particular:

   - only use the Register component as a memory element, where each instance of this component has a descriptive label.
   - do not use lookup tables or DIL chips.
   - you must retain the filenames and I/O interfaces from the template (select interfaces may be extended, but not changed).
   - it must be possible to start the simulation.

   Use the tool we provide to verify that you have met these requirements before uploading to Brightspace. Failing to meet these requirements likely results in a failing grade!

2. A `README` file in plain text or PDF (not `.docx`), containing:

   - your names and student numbers
   - a description of any unimplemented functionality or bugs in your CPU that you are aware of. If you can clearly explain why a bug exits and what could be done about it, you may still receive points for the corresponding part.
   - (optional) if you think it is necessary to explain certain design choices you made, you can do so briefly here.

   Please keep the `README` as short as possible.

3. (Optional) If you tested your CPU with an assembly program running on the complete computer, include the commented program. Give a clear description of the behaviour of your program in the comments.

Submit all files via Brightspace before Monday, March 27th at 07:00.

5 Tips

Brightspace

We will maintain a collection of recommendations for dealing with specific Digital issues, and other tips, on a separate Brightspace page. This will be updated as we encounter new issues, so please check this from time to time.

Instruction Decoder design

As the Instruction Decoder is rather complex, it really helps if you decompose the problem into smaller sub-problems, which you can solve using sub-circuits. Think of it like using helper functions to make a complex function easier to implement in a programming task. Two notable sub-problems you can solve using sub-circuits are:

1. Determine the machine code format of the instruction being decoded, as listed in Appendix B of the CPU manual. The output of this sub-circuit could be a series of 1-bit output signals, which are high if and only if the machine code matches that format. Another alternative is to output a single 3-bit signal that represents the number of the format (1-6). Values 0 or 7 could be used to indicate an invalid machine code that does not decode to any defined instruction format.

2. Extract the variable parts (`c10`, `c22`, `opcode`, `A`, `B`, etc) from the machine code, and output each part as a separate output signal. The Splitter/Merger component in Digital is especially useful for this.

Now that we have access to all the variable parts, and know which machine code format we are dealing with, implementing the Instruction Decoder becomes a lot more manageable. We strongly recommend you create Data paths for each machine code format to see which output signals the Instruction Decoder should have, and what their value should be during each format and phase of the CPU instruction cycle. By overlaying these Data paths, you can see which signals you connect directly, and at which places you need to introduce multiplexers to select the right signal at the right moment. Some other tips:

1. Bitmasking[2] is a powerful technique that can be used to test if a machine code `MC` matches a certain format using `(MC & MASK) == VALUE`. Here `MC & MASK` is used to extract specific bits of `MC`, controlled by `MASK`, which we then compare against `VALUE`. We only want to extract bits which we expect to have a constant value. All variable bits are set to 0 due to the mask used in extraction, thus creating a predictable bit pattern we can test for. It is possible to create a `MASK` and `VALUE` pair for each machine code format listed in Table 4 of the CPU manual.

2. Rather than your Instruction Decoder outputting both a 10-bit `c10` and a 22-bit `c22` signal, it is easier to have it output a single 32-bit `c32` signal. This signal is formed by either right-padding `c22` with ten 0 bits, or by sign extending `c10` to 32-bits, depending on the machine code format.

3. During the Fetch phase, the CPU must increment the `PC` with 4, as well as signal a memory read at the `PC`. By decoding a constant machine code value you can get most of the required signals to be output, reducing the number of multiplexers required.

[2] `https://stackoverflow.com/questions/10493411/what-is-bit-masking`