Computer Architecture

An embedded approach

Module 8

CPU Design
(the DIY approach)

Contents

Soft core processors

Chapter 7 discussed the practicalities of real CPU ICs, namely the very common System-on-Chip approach that encompasses almost all microcontrollers, embedded CPUs and is beginning to be found in larger netbook and laptop processors (and which will, presumably also find its way to desktop processors in time).

So, for example, while there is only one actual ARM7 core design, there are several variants and literally hundreds of different CPUs based upon that core. Each of these different device families and individual devices has its own set of unique features and attributes.

Soft core processors

If you are designing the next iPod or similar mass-market product, it is entirely possible that an SoC manufacturer such as Samsung would take your list of desired features and design the hardware for a new device based around this.

But smaller manufacturers do not have that luxury: we don't sell enough products for these companies to design a new IC specif cally for us... we are expected to use one of the existing designs.

Luckily, for the ARM at least, there are thousands of choices.

However there is another approach – we can build our own!

Soft core processors

Building our own silicon may not be cost-effective, but we could create a computer design on FPGA (Field Programmable Gate Array) that exactly meets our needs.

If the FPGA (which is relatively expensive in quantity) works fine then we can later create a semi-custom ASIC – our own IC – using the design.

This would be cost-effective for medium quantities (small quantities would be better served by FPGAs and larger quantities by ASICs).

We will not argue the case for ASICs of either variety here, simply discuss how we might use one to implement our own CPU and required set of peripherals.

Soft core processors

So assume we need to implement some functions...

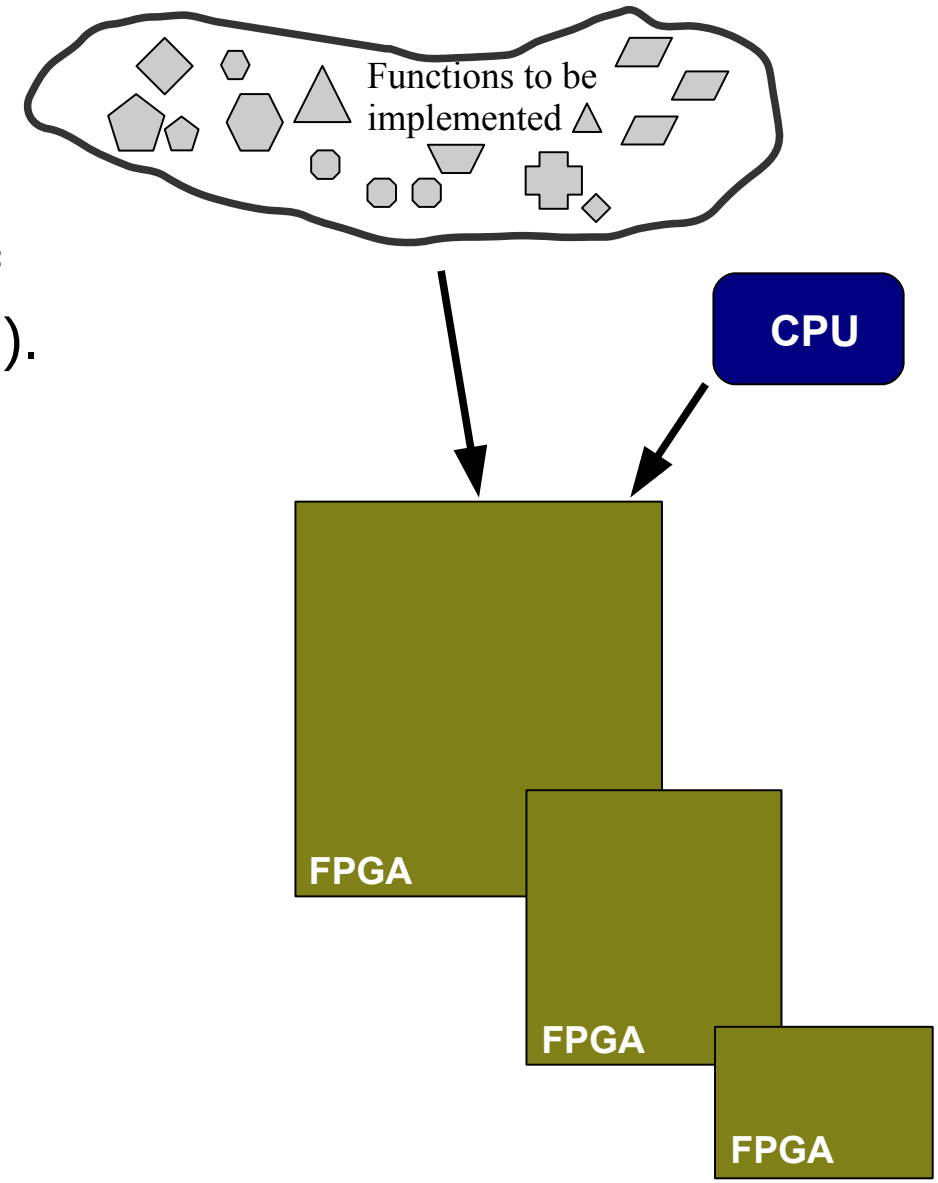
- We could code these into an FPGA
- We could write them as software on a standard CPU
- We could build our own CPU (and then write software for it)
- Any combination of the above

Soft core processors

The first tasks are usually deciding what functions need to be implemented, what type of CPU to use (our own/standard one) and what FPGA device to squeeze all of these into (if it is to contain the CPU).

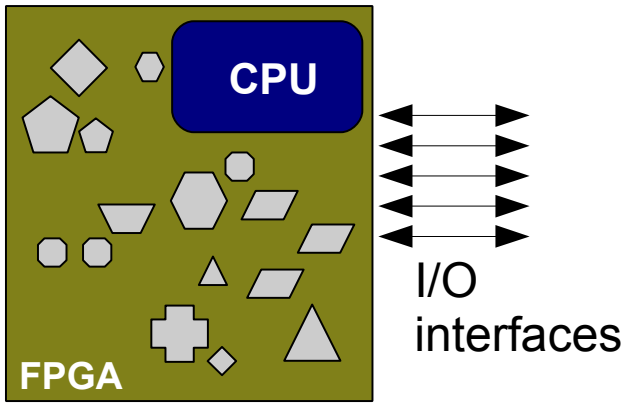
There may be some 'trading' of requirements between different methods of implementation, depending on the resources available.

There are also many other factors not mentioned here (speed, cost, end-of-life issues, availability, power, expandability, interfaces... these are just a few).



Soft core processors

Finally, in the picture below, we see that all of the required functions have fitted into the FPGA, creating our own custom SoC device.

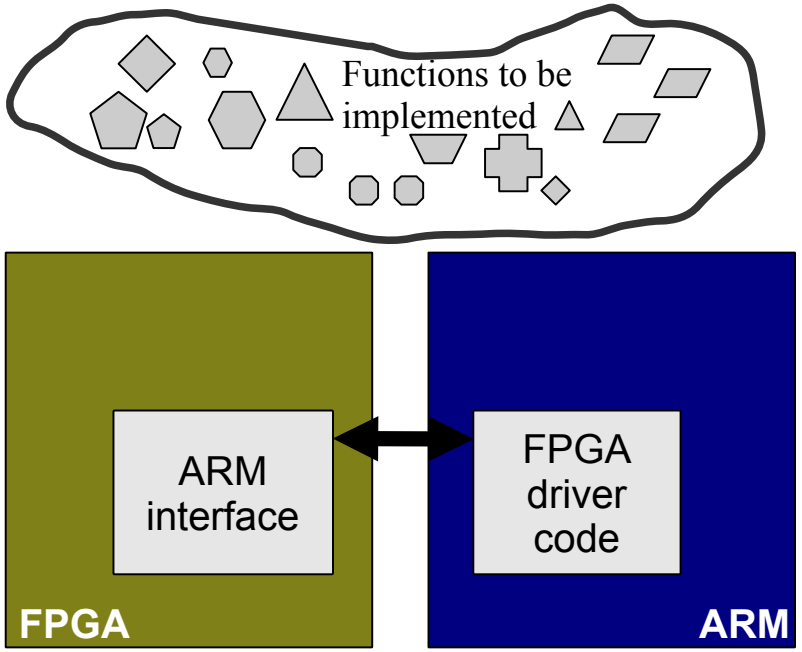


As you might have guessed, we have skipped a few stages – one of the most important is the unit testing phases, plus verification and validation (which we will meet again in a while).

Soft core processors

To be honest, we also have to note that in many cases these days, the FPGA (which may or may not contain a soft core), is used in conjunction with an existing processor such as an ARM. The main reason for this is software: ARM code is easy to develop – there are many tools and a huge library of existing resources. By comparison, the person designing his own CPU has to develop everything!

In this case, the previous step would be to determine the functions that need to be supported, and attempt to partition them to either the FPGA, to the processor or find some other solution for their provision.

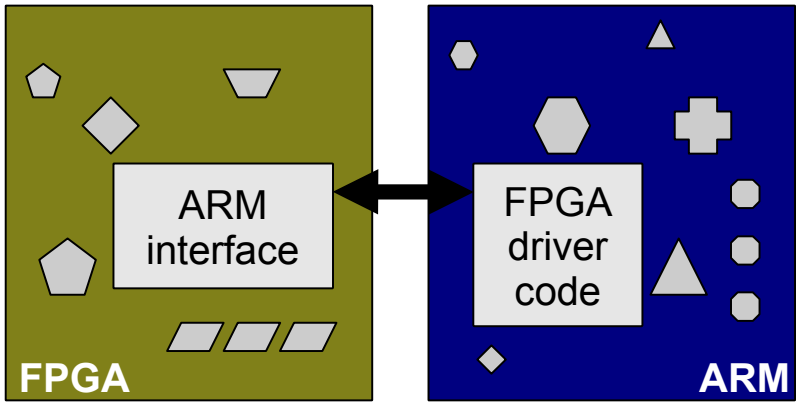


Soft core processors

When doing the kind of partitioning mentioned on the previous page, we would often start by defining a fixed interface between the two devices.

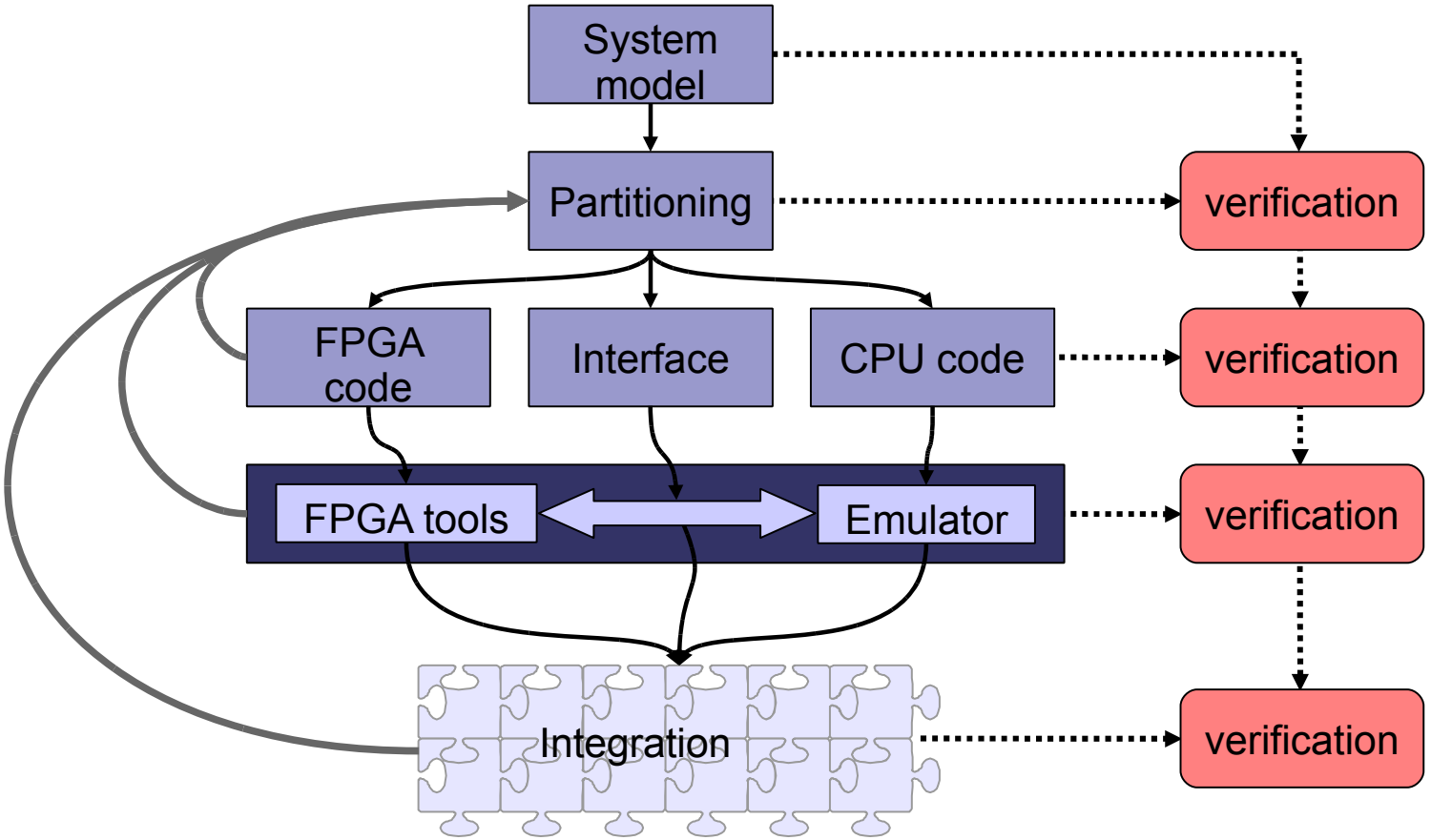
This is something that can often remain static – we simply have to change the information that passes over that interface depending upon where the functionality has been allocated.

Of course we must not forget to ensure the interface has sufficient bandwidth, control capabilities, and it's data/control latency is not excessive.



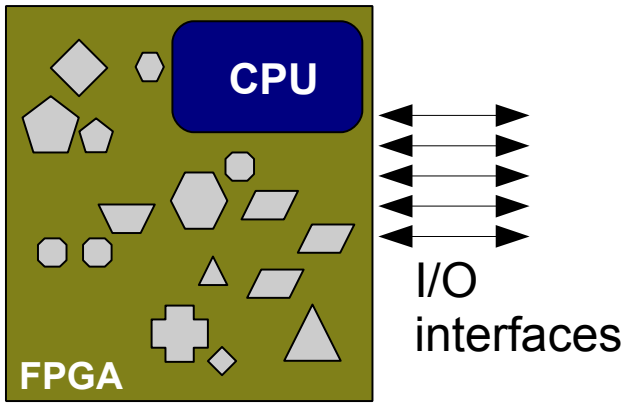
Custom computing design f ow

This approach can be described as a kind of iterative process as shown below. Note that **verif cation** is done between every stage. It is very very easy for errors to creep in during this process that could be extremely diff cult to track down later – so verif cation is essential!



Custom computing

Of course, if you decide you don't need the power of an entire ARM CPU, or you want to save cost, or stick with a one-chip solution, then you need to use a CPU core inside the FPGA:



Each FPGA manufacturer has their own cores (such as Nios-II, MicroBlaze), and some FPGAs actually contain in-built hard CPUs.

However you might want to design your own. To show how this approach works, **we will develop TinyCPU**.

What is TinyCPU?

A fully working CPU, synthesisable in an FPGA

16-bit stack based architecture

Simple but fully functional

Written in Verilog, programmable in assembler (and C)

Highly modif able, quick to learn

Input and output ports

Extensive range of conditional instructions

What is TinyCPU?

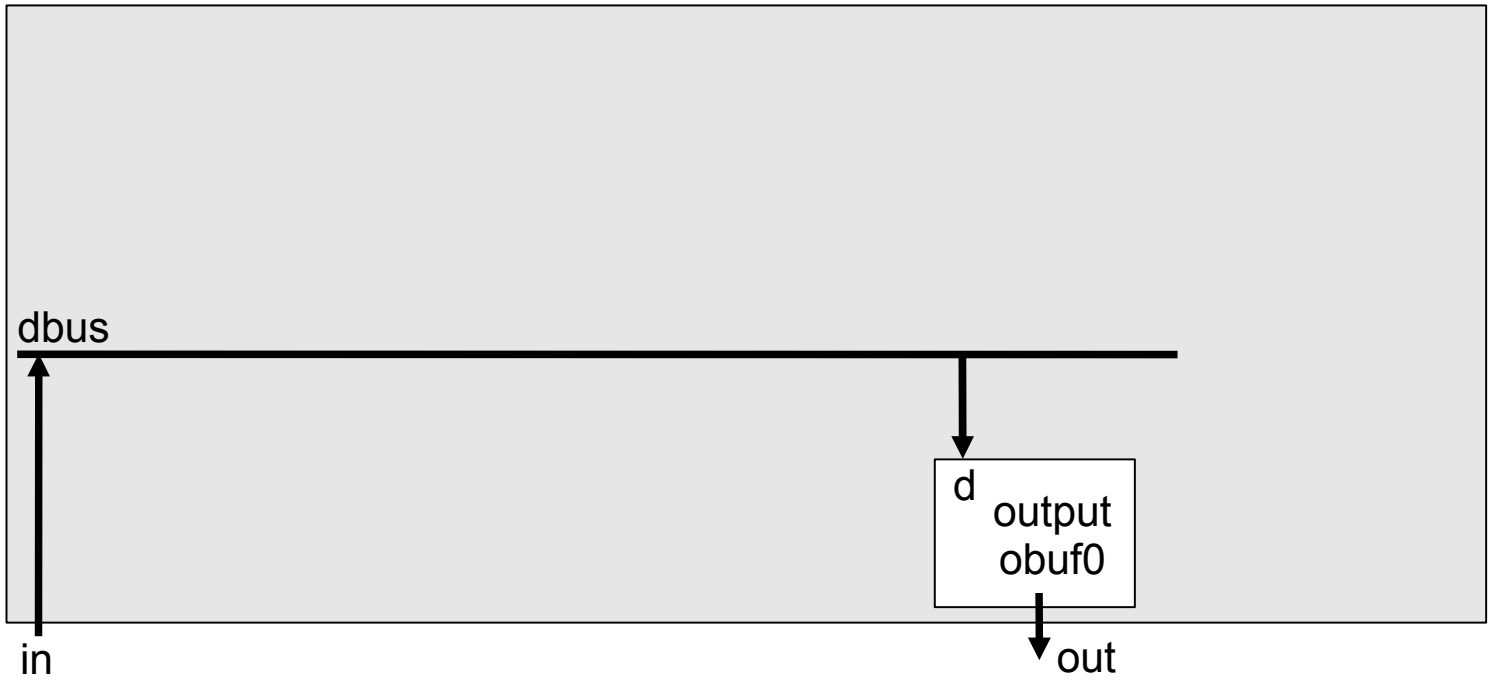
TinyCPU was designed by Prof. Koji Nakano of Hiroshima University

HIROSHIMA UNIVERSITY

It is also used extensively for teaching elsewhere, including at Nanyang Technological University, Singapore

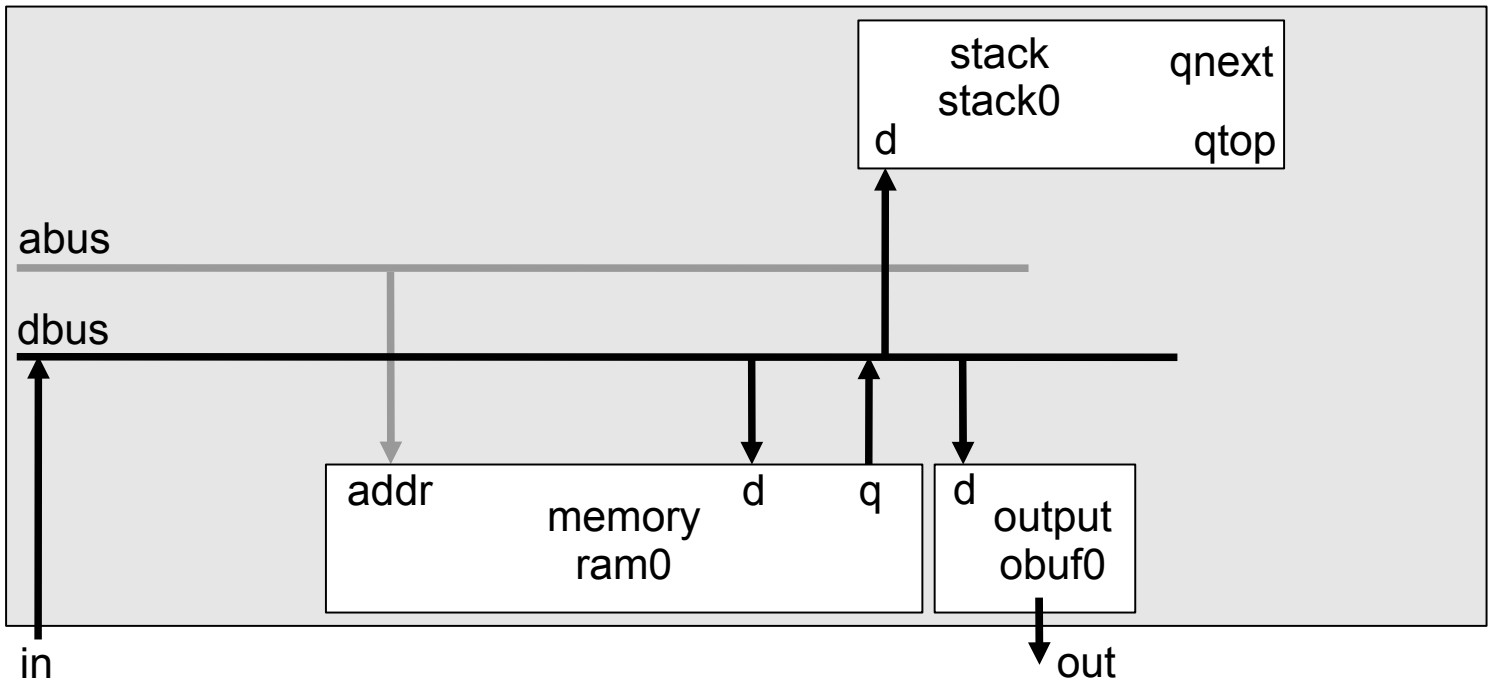
NANYANG TECHNOLOGICAL UNIVERSITY

Creating TinyCPU



We start with a single-bus architecture - one data bus with I/O

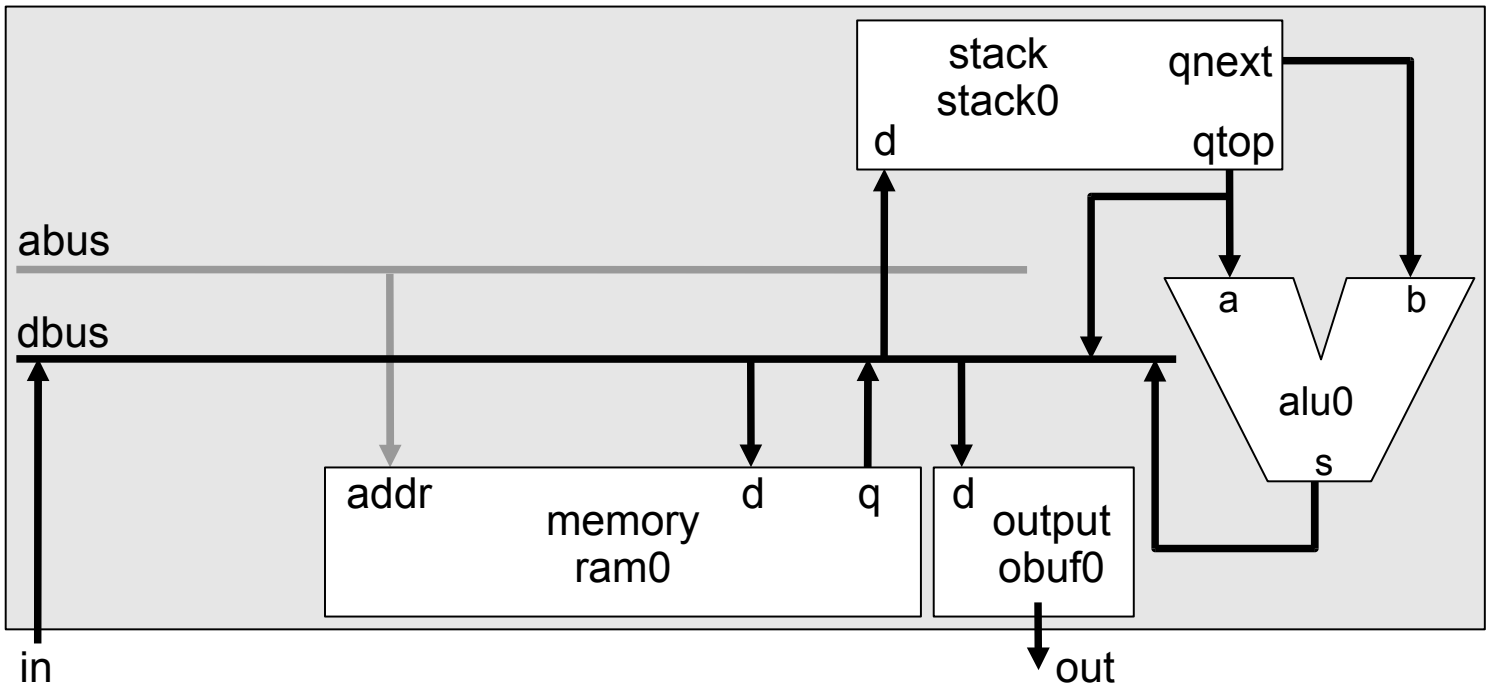
Creating TinyCPU



We start with a single-bus architecture - one data bus with I/O

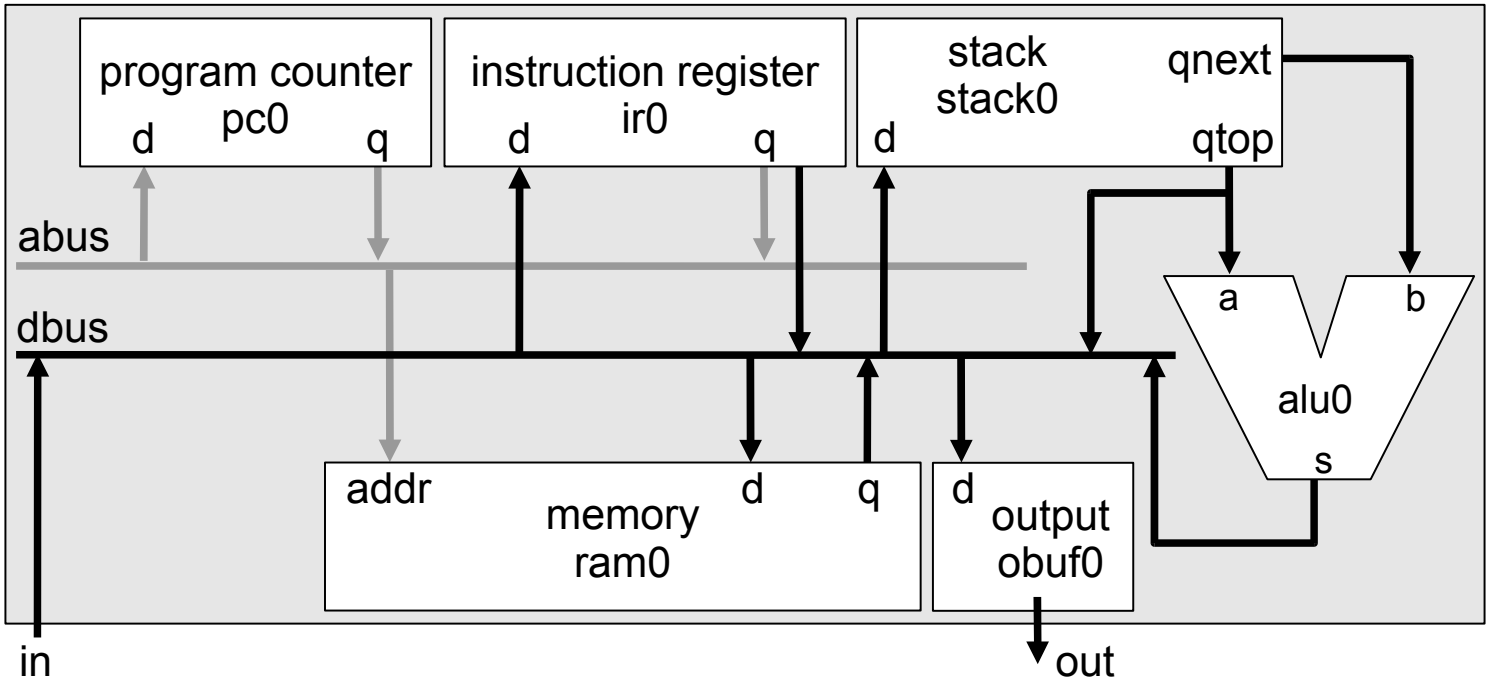
Then add memory (vonNeumann) and a stack

Creating TinyCPU



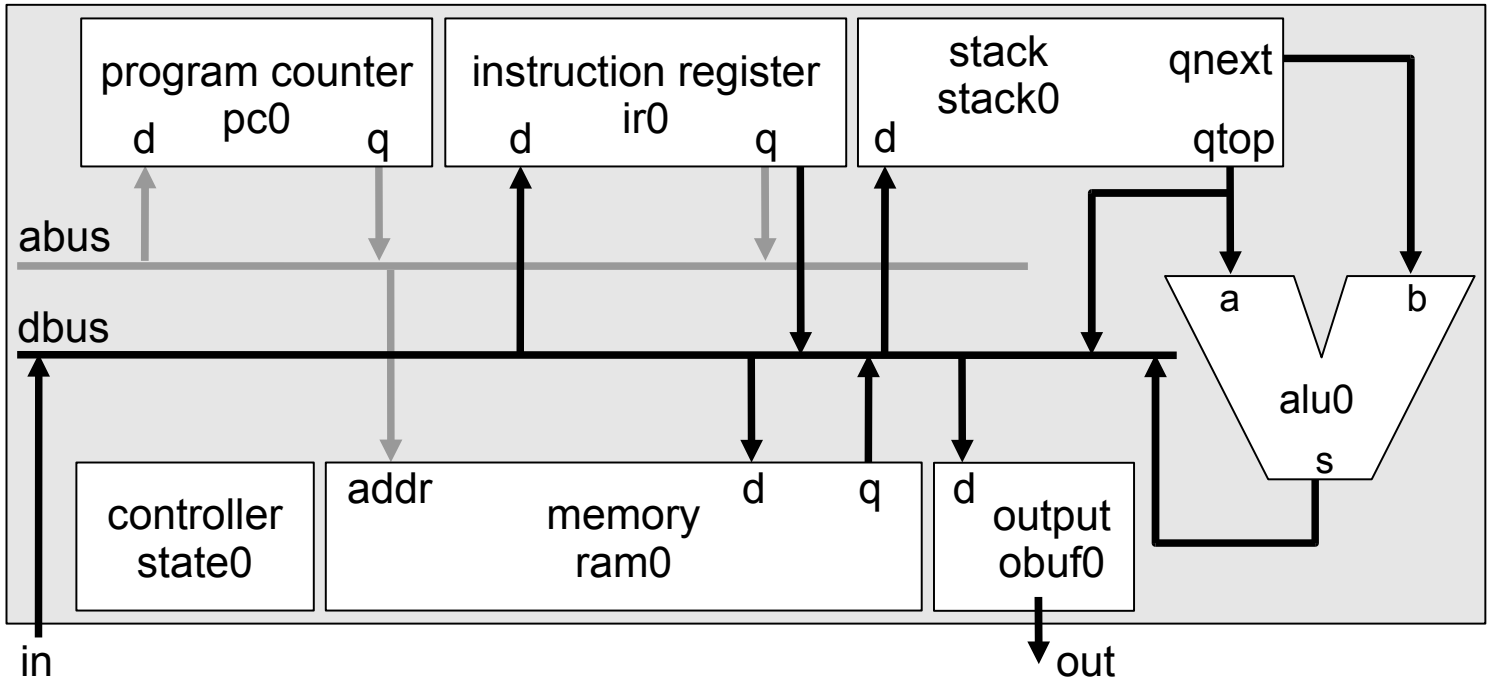
We start with a single-bus architecture - one data bus with I/O
Then add memory (vonNeumann) and a stack
Just one functional unit – an ALU (also contains a multiplier)

Creating TinyCPU



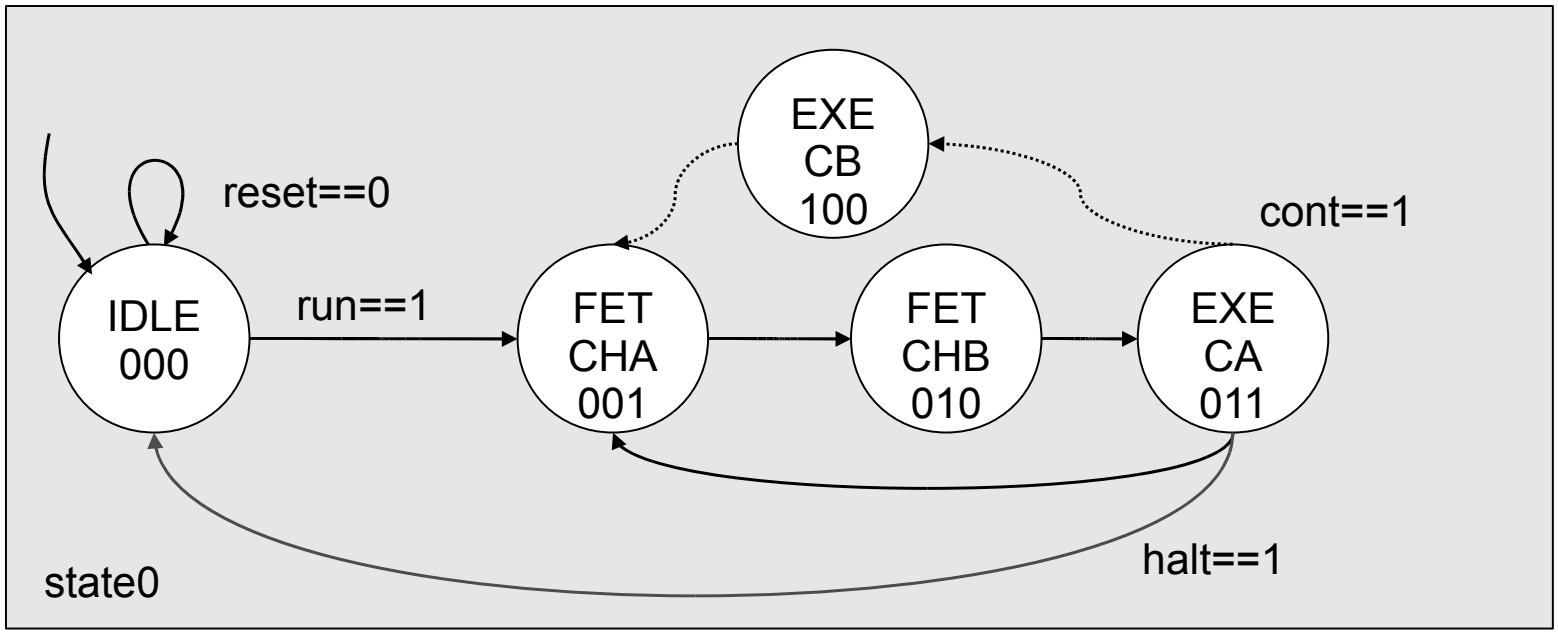
We start with a single-bus architecture - one data bus with I/O
Then add memory (vonNeumann) and a stack
Just one functional unit – an ALU (also contains a multiplier)
A PC (program counter) and instruction register (IR) come next

Creating TinyCPU



We start with a single-bus architecture - one data bus with I/O
Then add memory (vonNeumann) and a stack
Just one functional unit – an ALU (also contains a multiplier)
A PC (program counter) and instruction register (IR) come next
Finally, a centralised state machine controller

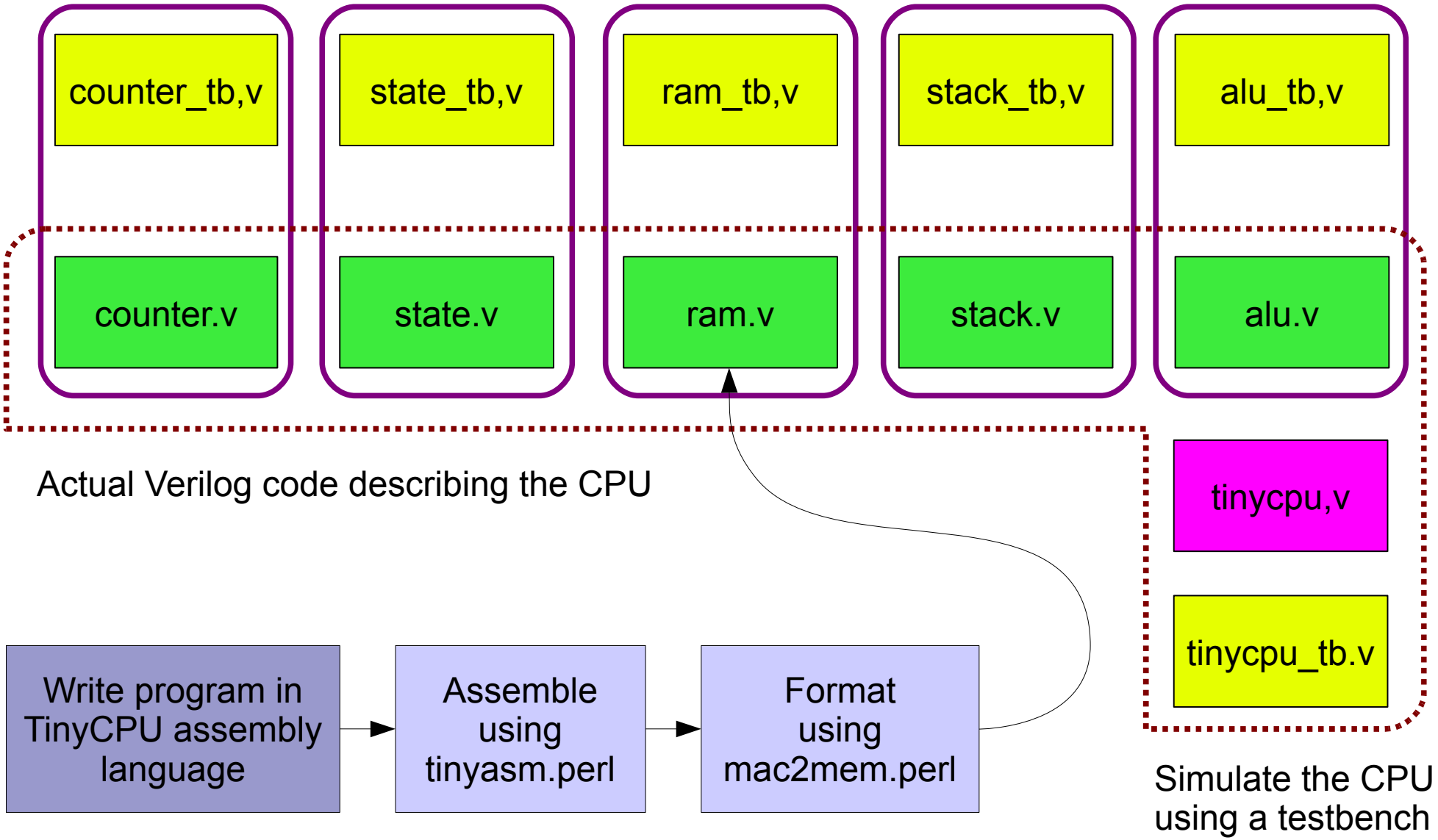
Creating TinyCPU



Control involves 4 active states:
FETCH_A,
FETCH_B
EXEC_A
EXEC_B (not needed for some instructions)

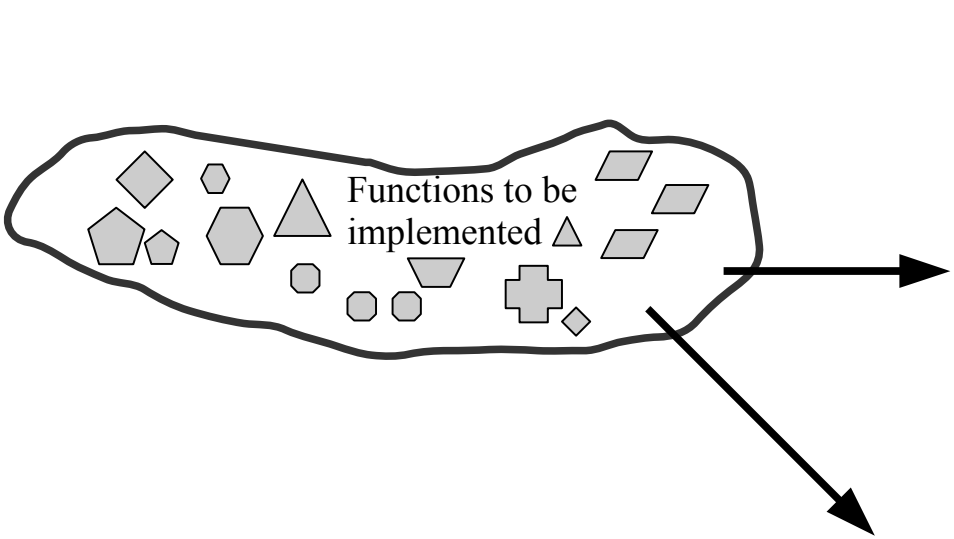
Creating TinyCPU

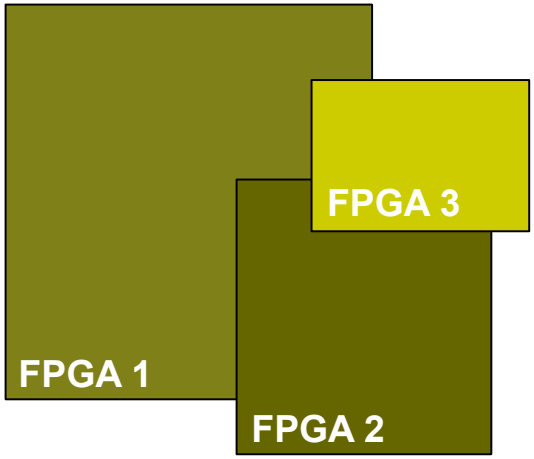
Test benches are provided for individual verilog modules, and for overall.

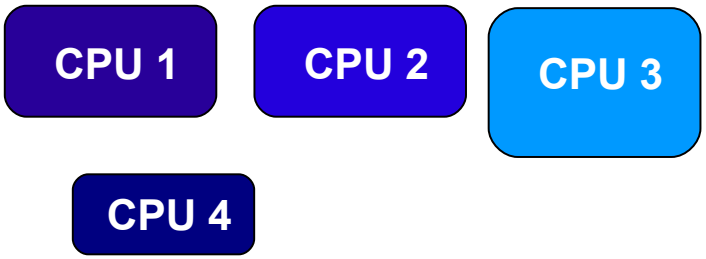


Hardware/software co-design

Moving back to the issue of implementing a set of functions... This is a very common task for the embedded systems engineer.





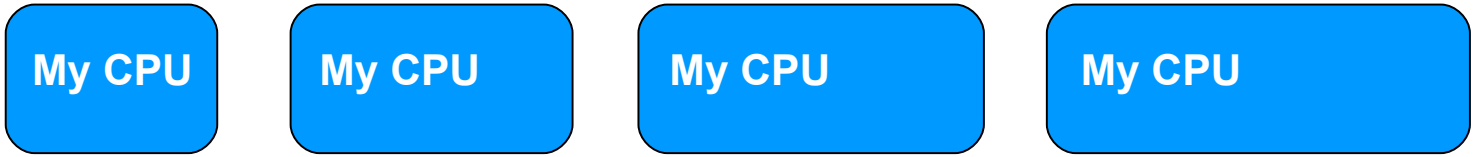


Sometimes we implement a function or a task as software on a CPU, sometimes as f rmware on an FPGA, sometimes as actual hardware.

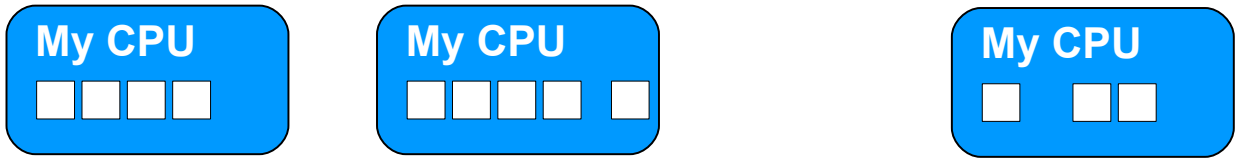
But if we design our own CPU, we can customise it to provide exactly the capabilities we want.

Hardware/software co-design

We can expand or shrink the capabilities of our CPU as required.



Can can add specialised instructions to the instruction set
(just like in a CISC processor!) - or remove unused ones for eff ciency!



Note: we had seen this first in Chapter 5 (see Fig. 5.23 on page 234)

Hardware/software co-design

Hardware/software co-design can be, at best, is a way of jointly optimising a system... juggling the hardware capabilities and software programs through a number of alternatives (relatively quickly) to choose a working system with lowest cost/power/size (you choose one of these – the smallest may not be the cheapest or the lowest power!).

Of course, all this depends on how good the tools are that you use!

More research is def nitely still needed in this area.

Conclusion

This chapter is not one found in many (or any?) other computer architecture textbooks. The world is becoming an exciting place for embedded system developers who can now harness the growing power of FPGAs to implement their own CPUs.

We have explored TinyCPU, a very small Verilog processor that we can build into an FPGA and extend (full steps for this are given in the textbook, and a set of laboratory sessions which explore this process are available as supplementary material).

Finally, the flexibility we have to choose different solutions for hardware and software has given rise to a set of development tools that allow us to explore many potential options with the aim of choosing the best one for our design criteria.