Hardware vs. Software

Choosing between implementing an application in HW or implementing it in SW may seem like a no-brainer -- clearly writing software is easier!

Software is flexible, compilers are very fast, there are lots of libraries available and computing platform are cheap and plentiful

More importantly, it seems a waste of effort to design a new hardware system when it is easy just to purchase one, e.g., a desktop computer

So what are the drivers for custom hardware?

Let's consider two important metrics that are used to compare hardware and software

Performance and Energy Efficiency

Performance:

Expressed as the amount of work done per unit of time

Let's define a unit of *work* as the processing of 1 bit of data

Hardware vs. Software

The figure shows various embedded system cryptographic implementations in software and hardware that have been proposed over 2003-2008.

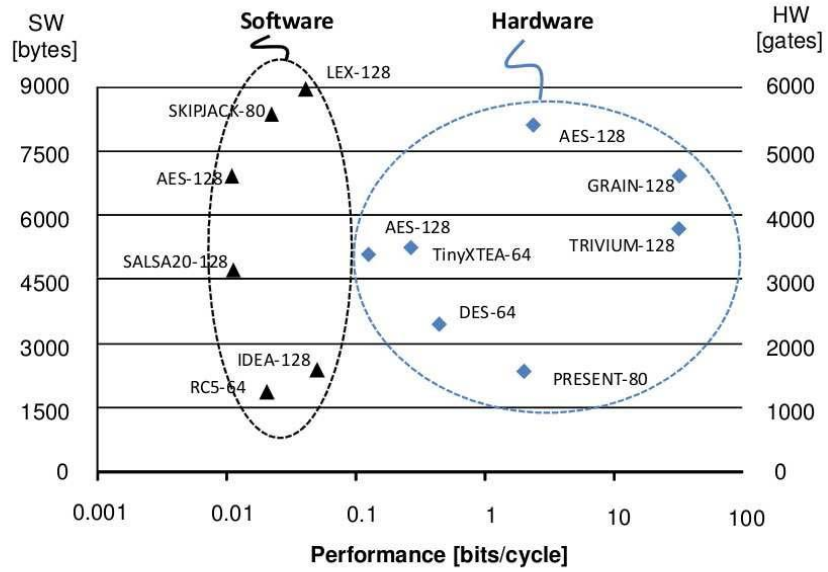


Fig. 1.4 Cryptography on Small Embedded Platforms

Performance in bits/cycle shown along x-axis shows that hardware has better performance than embedded processors (software)

Bear in mind that even though hardware executes more operations in parallel, high-end micro-processors have VERY high clk frequencies

Therefore, software may in fact *out-perform* dedicated hardware

Hardware vs. Software

Therefore, performance is **not** a very good metric to compare hardware and software, especially in resource-constrained environments such as IoT

A better metric (that is independent of clk frequency) is **energy efficiency**, i.e., the amount of useful work done per unit of energy

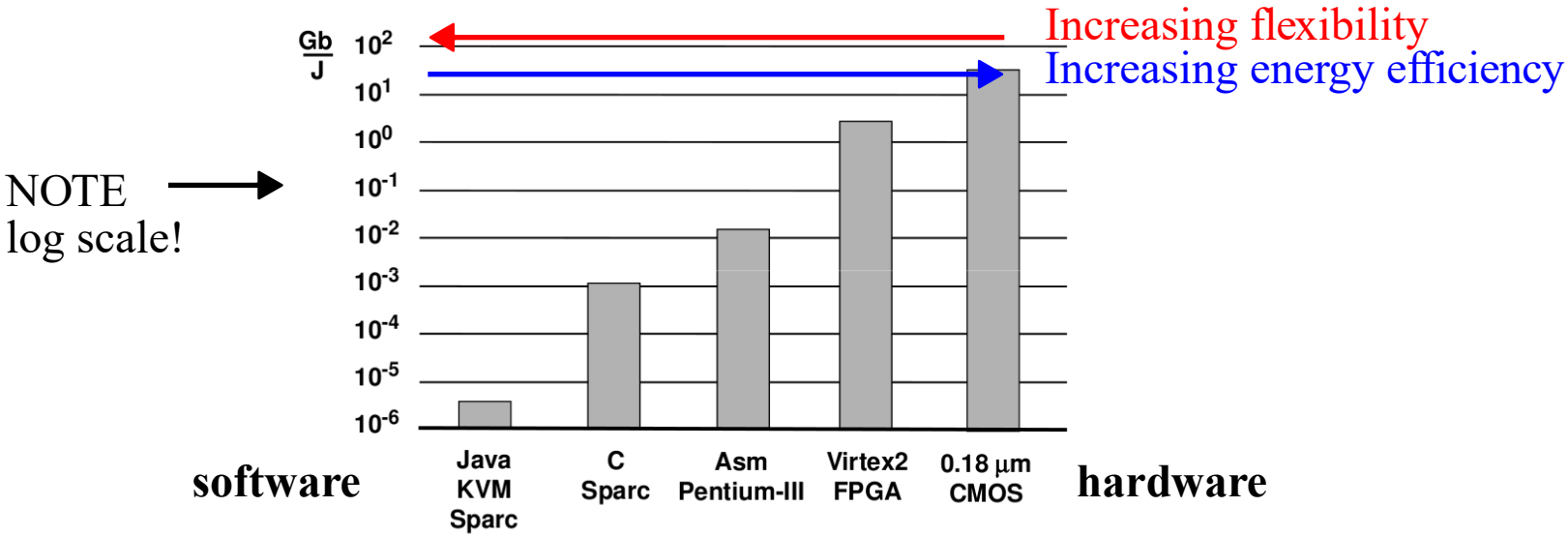


Fig. 1.5 Energy Efficiency

This graph shows energy consumption of an AES engine (encryption) on different architectures, with y-axis plotting Gigabytes per Joule of energy

This shows battery-operated devices would greatly benefit using less flexible, dedicated hardware engines

Hardware vs. Software

This is true b/c there is a **large overhead** associated with executing software instructions in the microprocessor implementation

- Instruction and operand fetch from memory
- Complex state machine for control of the datapath, etc.

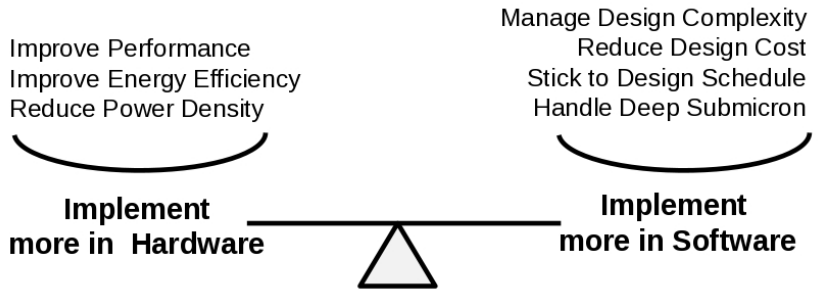
Also, specialized hardware architectures are usually also more efficient than software from a *relative performance perspective*, i.e., amount of useful work done per clock cycle

Flexibility comes with a significant energy cost -- one which energy optimized applications cannot tolerate

Therefore, you will **never find** a Pentium processor in a cell phone!

Hardware vs. Software

The complete picture of whether and how to implement a system is more compli- cated



Hardware or software present many trade-offs, some of which have conflicting objec- tives

Arguments in favor of increasing the amount of **hardware** (HW):

- **Performance and Energy Efficiency**:

  As indicated above, improvements in relative performance and energy efficiency is a big plus for hardware, especially battery-operated devices

  HW/SW codesign plays an important role in optimizing **energy-efficiency** by helping designers to decide which components of flexible SW should be moved into fixed HW

Hardware vs. Software

- **Power Densities**:

Further increasing clock speed in modern high-end processors as a performance enhancer has run-out-of-gas because of thermal limits

This is driven a broad and fundamental shift to increase **parallelism** within processor architectures

However, there is no dominant parallel computer architecture that has emerged as 'the best architecture' -- commercially available systems include

- Symmetric multiprocessors with shared memory
- Traditional processors tightly coupled with FPGAs as accelerator engines
- Multi-core and many-core architectures such as GPUs

Nor is there yet any universally adopted parallel programming language, i.e., code must be crafted differently depending on the target parallel platform

This forces programmers to be **architecturally-aware** of the target platform

Hardware vs. Software

Arguments for increasing the amount of software (SW):

- **Design Complexity**

  Modern electronic systems are extremely complex, containing multiple processors, large embedded memories, multiple peripherals and input-output devices

  It is generally difficult or impossible to design all components in fixed hardware

  On the other hand, software implementations running on processors can better handle complexity and additionally allows for updates and bug fixes

- **Design Cost**

  New chips are very expensive to design and fabricate

  Programmable architectures including processors and FPGAs are becoming more attractive because they can reused over multiple products or product generations

  System-on-Chip (SoCs) are good examples of this trend

Hardware vs. Software

- **Shrinking Design Schedules**

  Each new technology is more complex than the previous generation, and the move to the next generation happens more quickly

  For the designer, this means that each new product generation brings more work that needs to be completed in a shorter amount of time

  Shrinking design schedules require engineering teams to develop the HW and SW components of a system concurrently

  These trends also increase the attractiveness of software and microprocessor-based solutions

Finding the correct balance, while weighing in all these factors, is a complex problem

Instead, we will focus on optimizing metrics related to ***design cost*** and ***performance***

In particular, we will consider how adding hardware to a software implementation increases performance while weighing in the increase in design cost

The Hardware-Software Codesign Space

The proceeding discussion makes it apparent that there are a multitude of alternatives available for mapping an application to an architecture

The collection of all possible implementations is called the ***HW/SW codesign space***

The following figure represents the design space symbolically

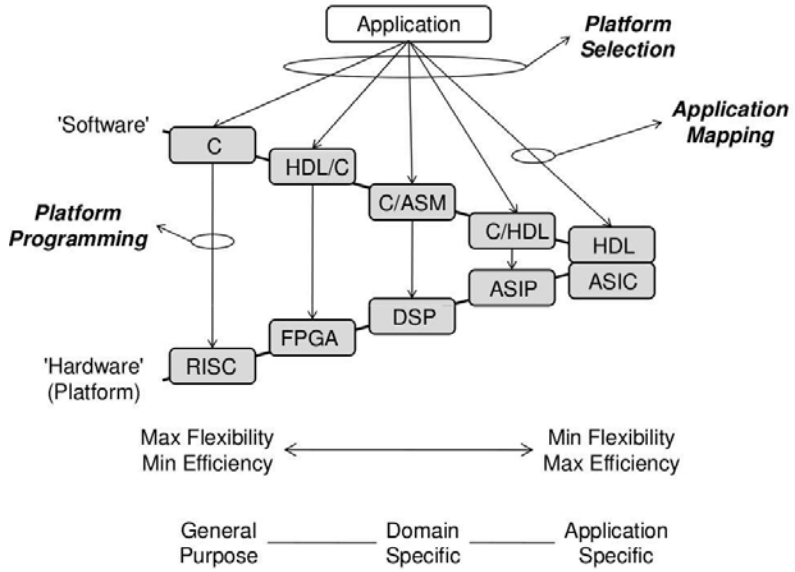
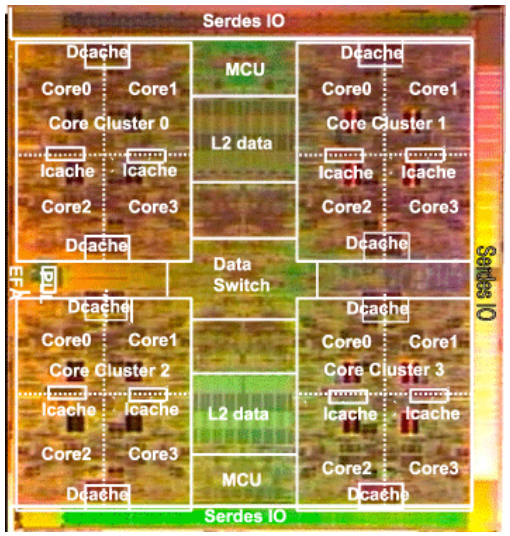


Fig. 1.7 The Hardware-Software Codesign Space

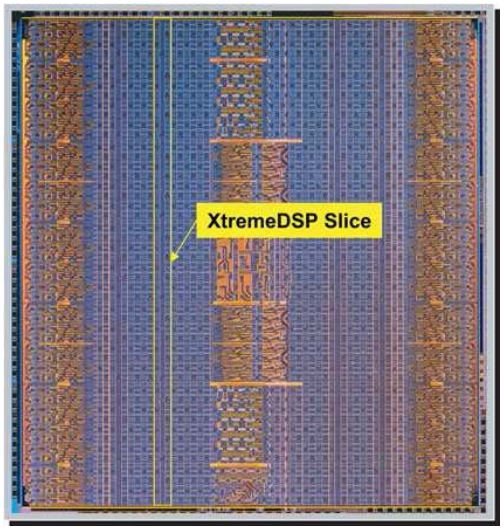
Platform Design Space: The objective of the design process is to map a *specification* onto a *target platform*

Examples Micrographs of Target Platforms

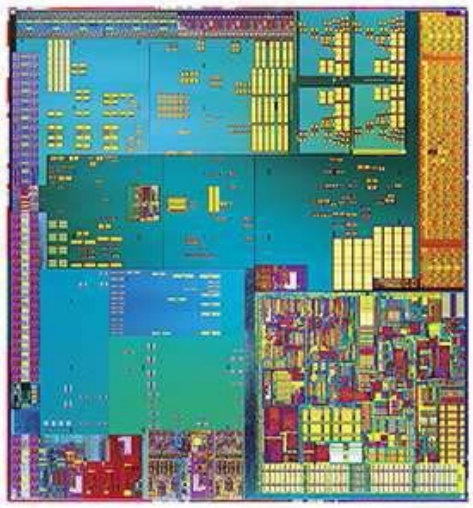
Microprocessor



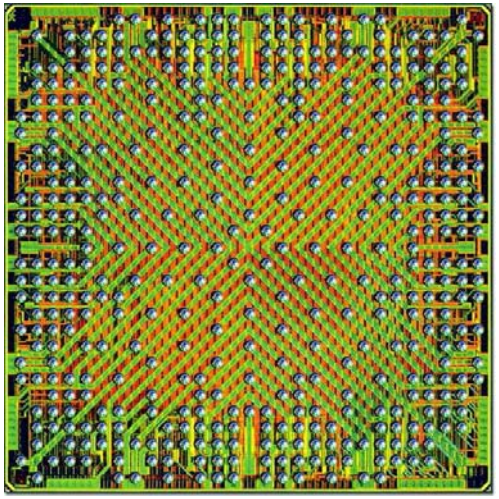
FPGA



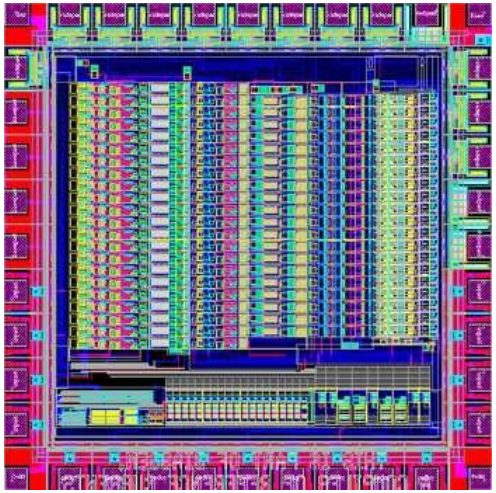
SoC



DSP

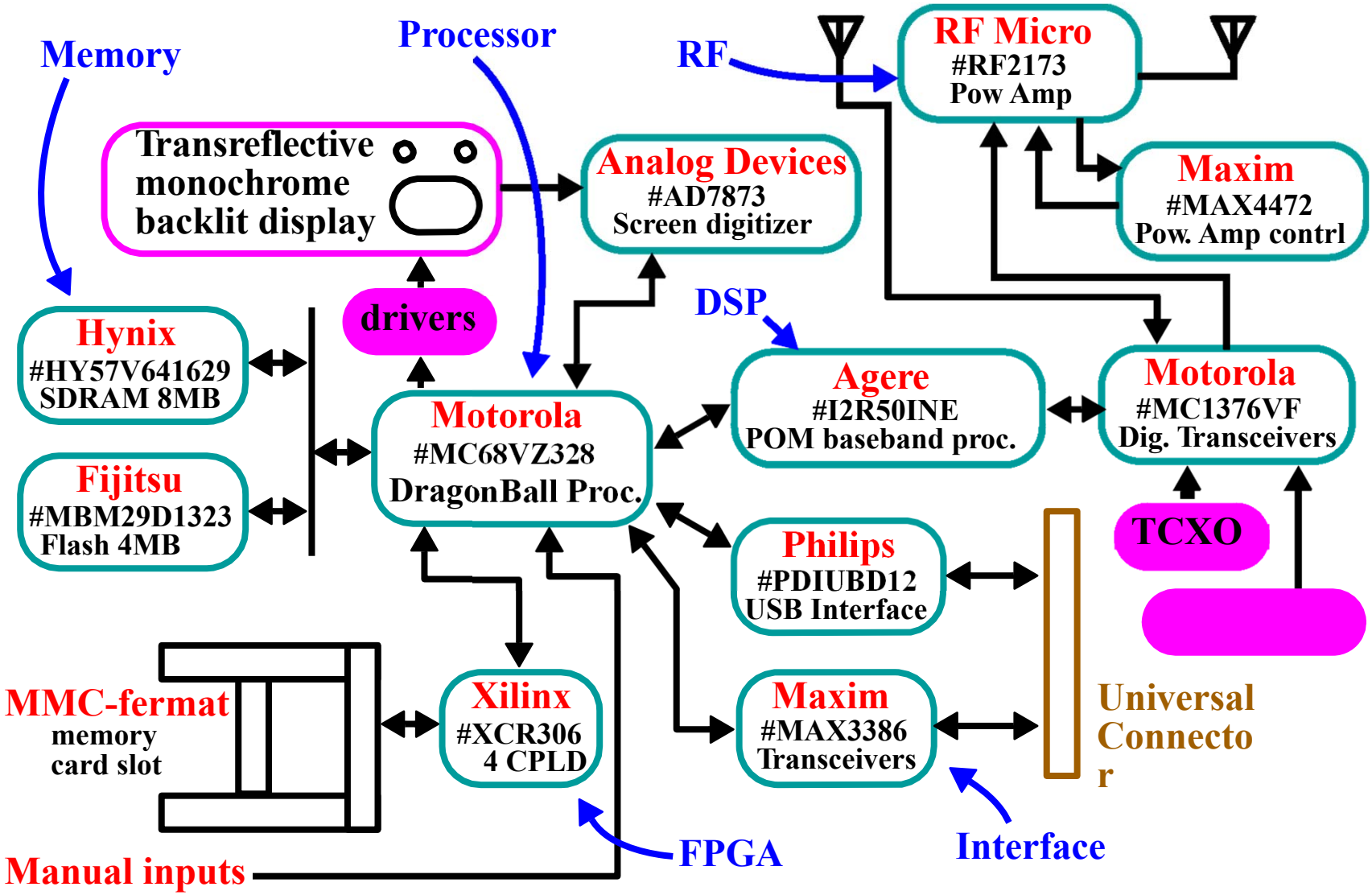


Microcontroller



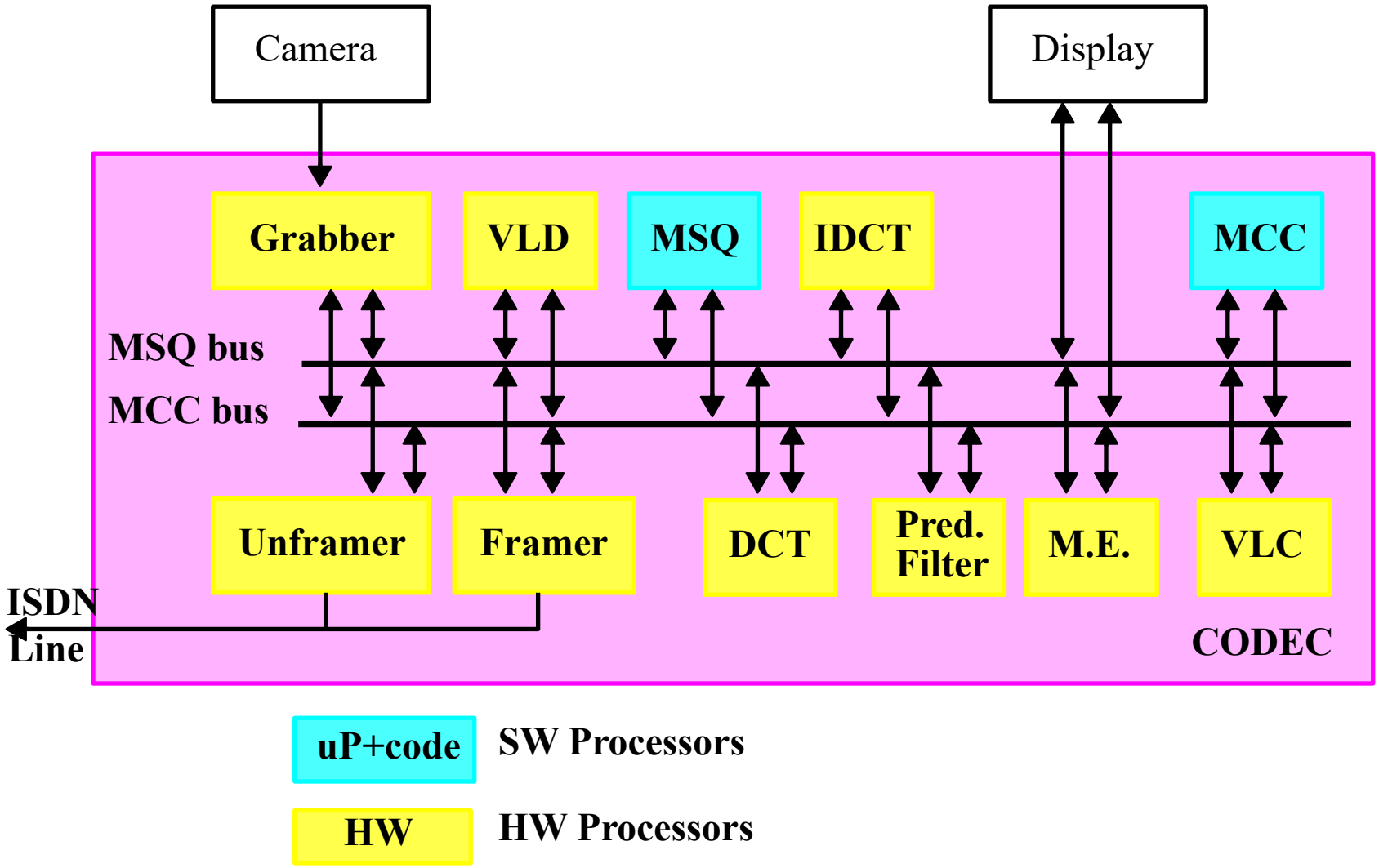
SoC Examples

Example System-on-Chip (SoC) with IP cores



Codesign Examples

Video Codec (H261)



The Hardware-Software Codesign Space

Each of the above platforms presents a trade-off between **flexibility** and **efficiency**

The wedge-shape of the diagram expresses this idea:

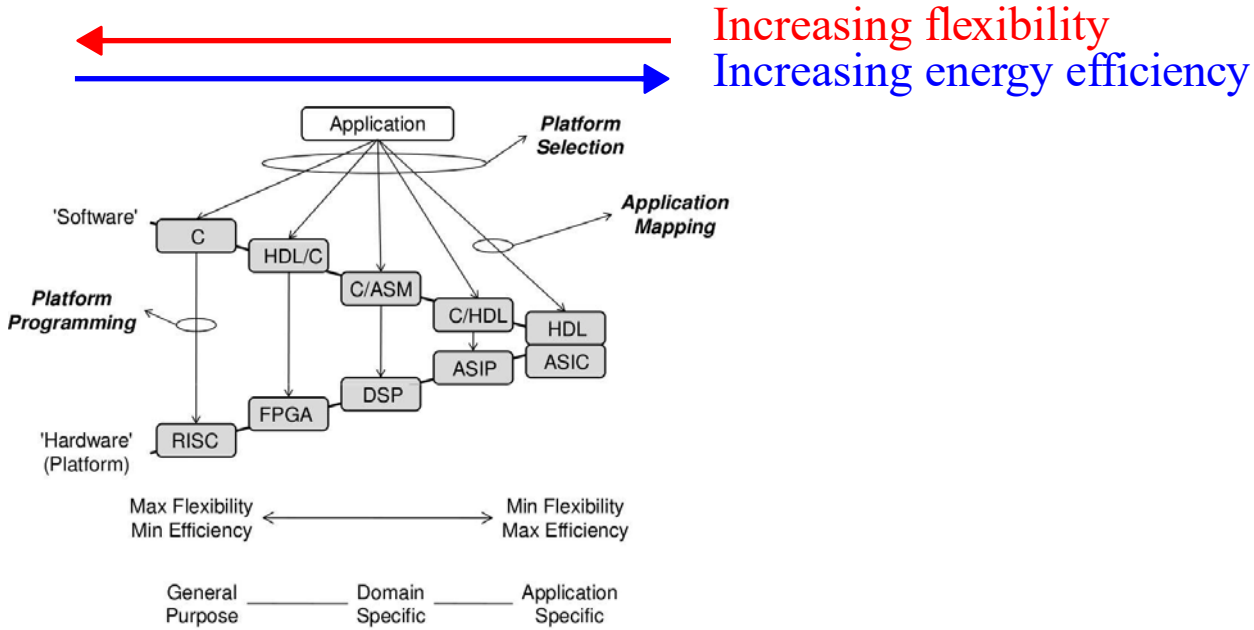


Fig. 1.7 The Hardware-Software Codesign Space

Flexibility refers to the versatility of the platform for implementing different application requirements, and how easy it is to update and fix bugs

Efficiency refers to performance (i.e. **time**-efficiency) or to energy efficiency

The Hardware-Software Codesign Space

Another concept reflected in the wedge-figure is the **domain-specific platform**

General-purpose platforms, such as RISC and FPGA, are able to support a broad range of applications

Application-specific platforms, such as the ASIC, are optimized to execute a single application

In the middle is a class called ***domain-specific*** platforms that are optimized to execute a **range of applications** in a particular application domain

Signal-processing, cryptography, networking, are examples of domains

And domains can have *sub-domains*, e.g., voice-signal processing vs. video-signal processing

Optimized platforms can be designed for each of these cases

DSPs and ASIPs are two examples of domain-specific platforms

The Hardware-Software Codesign Space

Codesign involves the following three activities:

- Platform selection
- Application mapping
- Platform programming

We start with a **specification**:

For example, a new application can be a novel way of encoding audio in a more economical format than current encoding methods

Designers can optionally write C programs to implement a prototype

Very often, a specification is just a piece of English text, that leaves many details of the application undefined

Step 1: **Select a target platform**

This involves choosing one or more programmable component as discussed previously, e.g., a RISC micro, an FPGA, etc.

The Hardware-Software Codesign Space

Step 2: **Application mapping**

The process of mapping an application onto a platform involves writing C code and/or VHDL/verilog

Examples include:

- **RISC:** Software is written in C while the hardware is a processor
- **FPGAs**: Software is written in a hardware description language (HDL)
  FPGAs can be configured to implement a soft processor, in which case, software also needs to be written in C
- **DSP:** A digital signal processor is programmed using a combination of C and assembly, which is run on a specialized processor architecture
- **ASIP:** Programming an ASIP is a combination of C and an HDL description
- **ASIC**: The application is written in a HDL which is then synthesized to a hardwired netlist and implementation
  Note: ASICs are typically non-programmable, i.e., the application and platform are one and the same

The Hardware-Software Codesign Space

Step 3: **Platform programming** is the task of mapping SW onto HW

This can be done automatically, e.g., using a C compiler or an HDL synthesis tool

However, many platforms are **not** just composed of simple components, but rather require multiple pieces of software, possibly in different programming languages

For example, the platform may consist of a RISC processor and a specialized hard- ware coprocessor

Here, the software consists of C (for the RISC) as well as dedicated coprocessor instruction-sequences (for the coprocessor).

Therefore, the reality of platform programming is more complicated, and automated tools and compilers are NOT always available

The Hardware-Software Codesign Space

Difficult questions:

- How does one select a platform for a given specification (harder problem of two)
- How can one map an application onto a selected platform

The first question is harder - seasoned designers choose based on their previous experience with similar applications

The second issue is also challenging, but can be addressed in a more systematic fashion using a **design methodology**

A **design method** is a systematic sequence of steps to convert a specification into an implementation

Design methods cover many aspects of application mapping

- Optimization of memory usage
- Design performance
- Resource usage
- Precision and resolution of data types, etc.

A **design method** is a canned sequence of design steps