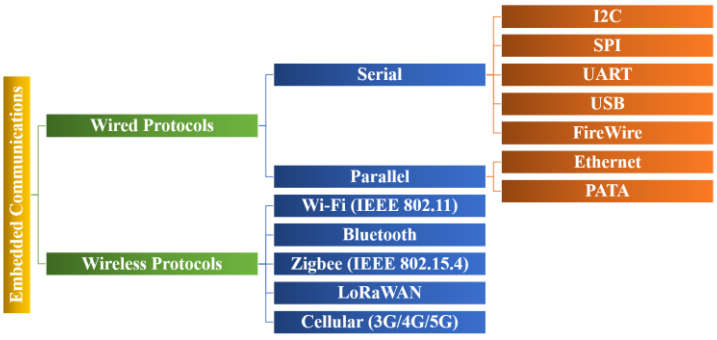
Signal Processing Algorithms & Architecture
Dr. Anirban Dasgupta
Department of Electronics & Electrical Engineering
Indian Institute of Technology Guwahati
Lec 21: Interfacing, Pipelining and Parallelism

Hello everyone, welcome to a fresh new lecture on the topics of interfacing, pipelining, and parallelism. So, I am Dr. Anirban Dasgupta and let us get started. So, we will start with the topic of interfacing. What is interfacing? Interfacing means connecting two or more components. In this context, we are going to interface or connect some external devices to our signal processor.

That is, we want to interface or integrate hardware components such as our ADC, our memory, or some sensors, and this is done to typically exchange data. So, there are several protocols used, out of which the three most common are the Serial Peripheral Interface, commonly known as SPI; the Inter-Integrated Circuit, known as I2C; and UART, or Universal Asynchronous Receiver Transmitter. Now, to get a bigger picture, these are the typical modes of interfacing or communicating from one set of equipment or component to the signal processor, or in general, to any embedded processing unit, you could say. So, naturally, the two types or two protocols—protocols mean a set of rules—are classified as wired protocols and wireless protocols.

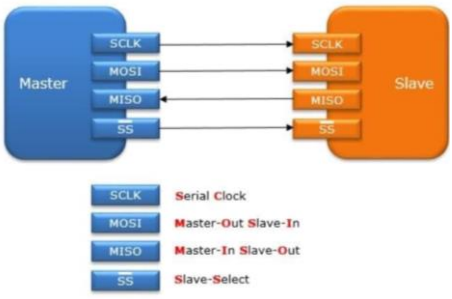


In wired protocols, they are typically classified as serial or parallel. I hope you know the meanings of serial and parallel. And in serial, the common modes are I2C, SPI, and UART, which I just highlighted in the previous slide. These three we will typically discuss because they are the main communication modules of the signal processor with external devices. But there are other modes like USB and FireWire; in serial and in parallel, we have Ethernet and PATA.

Similarly, in wireless protocols, now wireless protocols support both serial and parallel, and the common protocols or standards you can say are Wi-Fi, Bluetooth, ZigBee, LoRaWAN, and cellular. So, we will restrict our discussion to these three protocols because these are the most common protocols used to interface our signal processor with peripherals such as EDC. So, what is SPI, or Serial Peripheral Interface? So here, if you see this diagram, I have drawn four connections. The serial clock that is called SCLK is this. So, this is basically for sending the clock signals.

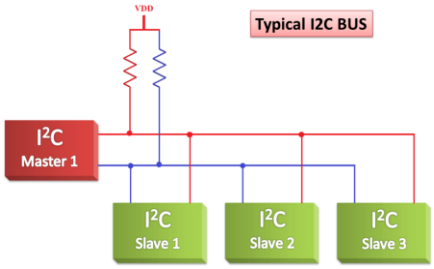
Then we have the MOSI and MISO. MOSI is master out, slave in, which basically connects the master to the slave. Now here, the master is our DSP or signal processor, and the slave is typically a component such as memory or, you could say, an ADC. So, there are a lot of SPI interface ADCs available. So, MOSI is basically master-in-slave-out.

Coming into the DSP that means it is gathering data. So, you have a sensor, and that sensor is sending data, so you will connect from the slave to the master through the MISO pin and then finally, this is one important pin in SPI called the slave select or chip select, and using this pin, you can basically select from a set of slaves. That is, it can have one master, multiple slaves but only one can communicate at a given time. Apart from this, you will have your VCC and ground connections as well.



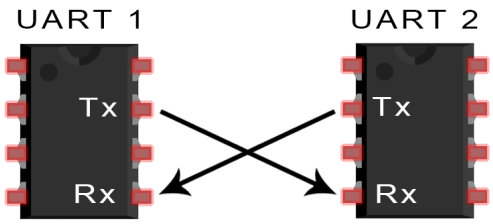
So, in total, there are usually 6 pins in the SPI protocol. One more protocol is the I2C Protocol. Now I2 is slightly different. Here we have the I2C bus and in this bus different slaves are connected. Like these slaves can be ADCs or memories whatever you say and this master is typically your signal processor.

And what are the pins here? So basically, you have your SDA, or serial data, which actually transmits the data. Then we have the serial clock, which is used for synchronization. We have the ground, and we have the VDD, and this is basically done using external pull-up resistors. Now, I2C is slightly better at connecting multiple devices because there is a common bus and you can connect multiple devices. In the I2C bus,



each I2C slave will have a unique address, and by the address, the master can get specific data from a specific slave.

There are several ADCs, for example, and each is connected via the I2C bus, with each having a unique address. So which ADC do you want to capture data from? You give that specific address in the code. But SPI is better if you are transferring large amounts of data because it operates at higher speeds compared to I2C. And the third is a universal asynchronous receiver-transmitter. So, the first concept here is asynchronous which means there is no clock signal involved in this communication compared to our I2C and SPI. And then we have the Tx and Rx pins, and there is a cross connection, like in SPI or I2C, what we did? We connected the same labeled pin with each other, but here, if you connect so many, you make this mistake while making the hardware. So, some connect the Tx to Tx and the Rx to Rx, but they do not get any signal. So, this you have to keep in mind in UART: Tx needs to be connected to Rx and vice versa, and of course, there will be a ground. So, these are the four pins: Tx, Rx, Vcc, and ground.

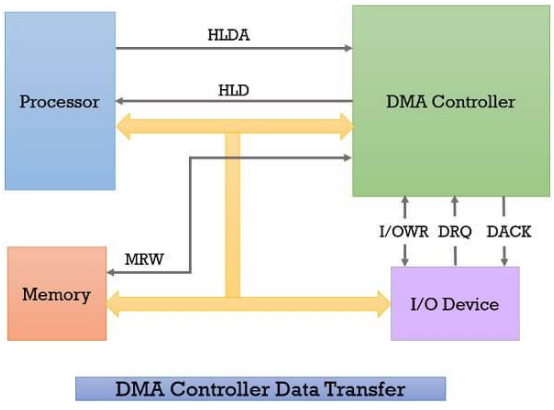


Tx and Rx have cross connections, Vcc you connect to the common source, which may be 3.3 or 5 depending on the communication protocol, or you can say the hardware, and then there is the ground. So, one very good thing that you can see is that you must have a pen drive or any USB device that you have. If you look carefully, there will be 4 pins, and these 4 pins are basically related to this, and that is why, by using a USB cable, you can charge, where this Vcc and ground are involved, as well as data transfer, where this Tx and Rx are involved and then there is a concept called baud rate.

So, this baud rate has to be the same for both the source and the receiver. If they are different, then something else will be transmitted. So, for proper data transmission both should be communicated at or configured at same baud rate. Now, this transmission can be simplex, half duplex or full duplex. What does it mean? Simplex means only one-way communication.

Say only the master is sending data and the slave is receiving. Although the master-slave configuration is not used here, you can say that one device, for example, one is sending and two is receiving only. So, that is simple communication. Half duplex means two-way communication is possible, but not at the same time and full duplex means both way communication is possible even simultaneously.

And here, data is transmitted as frames, where we have a start bit followed by some data bits. Then the parity bit, which is optional, is used to check whether we have even parity or not, and if there is no even parity, that means there is some noise corruption and some stop bits to indicate where we should stop receiving the data. Now, when this data is transferred from an external device to our processor, what is happening? It is first going to the processor or our signal processor, and then the signal processor will send this data to some memory. Now, if too much data is coming in, the processor is busy capturing the data and sending it to the memory, and in the meantime, the processor is not able to devote itself to other tasks. So, what is the solution? Can't I send the data directly from the interface to the memory instead of passing it through the processor? So that solution is by DMA, or direct memory access, where we can directly transfer the contents from the peripherals, like ADC, to the memory.

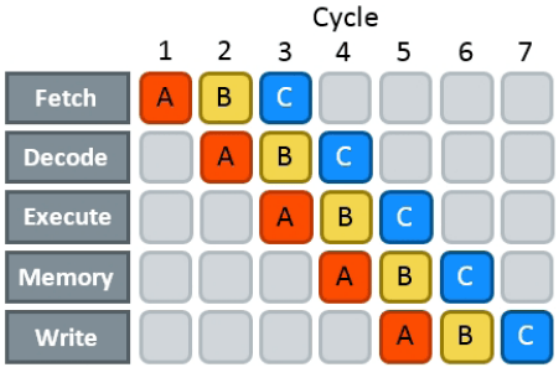


without the intervention of the main signal processor. And this is useful when we are transferring a large amount of data; in this manner, a large amount of data can be transferred without disturbing the main processor. So, who will control it? It is like an unruly mob. So, there is something called a DMA controller. And what will this DMA controller do? This will assist in the data transfer and reduce the workload of the main

processor.

So, how is it initiated? This DMA transfer is initiated by either a hardware trigger, such as a hardware button, or by a software configuration, which involves writing code. And now, once this transfer is initiated, the DMA controller will directly read the data from the source, which is the external device, and once the transfer is complete, the DMA controller will notify the signal processor, sometimes or often using an interrupt. So interrupt means the DMA controller sends an interrupt to the main processor, and the processor knows that, okay, the direct memory transfer is completed. Okay, so now we have understood the process of transferring data about the interfacing protocols and how we can directly transfer the data. Now we will go back to something related to the processing part, and a good example of intelligent processing is pipelining.

Now you might have heard this word, pipelining, but what is pipelining? Is it sequential? Is it parallel? Well, I would say it is a blend of both worlds, and what is pipelining? So, pipelining is basically overlapping the execution of multiple instructions. What does it mean? Say I have to complete a process. So, what will I do? I will divide this process into five stages. First is fetch, next is decode, then execute, and finally select the memory and write back to it. Now why will I do this? Because these are independent stages.



So, say there are several instructions; A is an instruction, B is an instruction, and C is an instruction. So, in the first clock cycle, what will I do? I will fetch the instructions, and that is fine. So, in the second clock cycle, a should get decoded; that is also true, and in the third clock cycle, a should get executed. In the fourth clock cycle, a should select a memory, and then finally the result is written back in the 5th clock cycle. So, after 5 clock cycles, the execution of A should be completed.

And you can see that you cannot do these things completely in parallel for instruction A because once you fetch it then only you can decode. Once you decode then only you can execute. So, this is a sequential operation. But these operations as such are independent

provided the previous stage is done. So, in the second clock cycle, while I am decoding my instruction A, I can fetch instruction B parallelly because fetch and decode for different instructions are independent.

And similarly, while A is being executed, B can be decoded, and C can be fetched. In this manner, I am utilizing the resources efficiently. So, this is basically the concept of pipelining. So, these are the stages that I said comprise a standard risk architecture pipeline. RISC means reduced instruction set computing, but it is not mandatory to use these 5 stages.

But this is a good example, and this risk pipeline is often taken as an example to explain the concept of pipelining. So, what are these stages? Let me just quickly tell you. So instruction fetch is, so this is the first thing. So, when I want to do a process, I want to fetch the instruction from the program memory. So, the instruction is basically the code that is written in the program memory, and then when the processor needs to execute it, it has to first fetch the instruction.

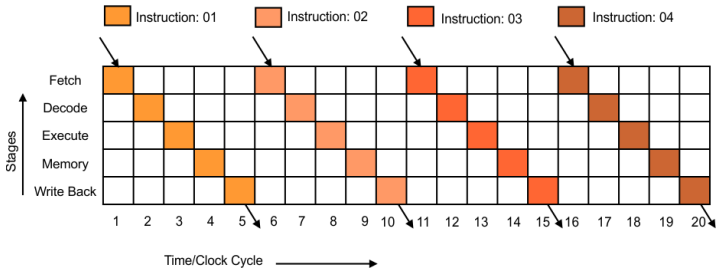
The next stage is that it has to decode because when it is fetching, it is getting a sequence of 0s and 1s. But what is the meaning of that? So, the processor needs to understand whether that means multiplication or convolution; what is it? So, the next stage is decoding that instruction. That means from that sequence of zeros and ones, I should know whether it is multiplication, addition, loading, or storing, and what is to be done. And once that instruction is decoded, now I know what to do. Next step is execution that is performing that operation.

Now, once that operation is performed, next I need to write that result, and for that, where should I write it? So, first I have to select the memory, and once I select the memory, the final stage of the pipeline is to write back the result. So, if we go to the mathematics of this pipelined architecture, I can break the whole operation into S stages and say that if I have I instructions and each instruction takes C clock cycles, the assumption is that each stage will take exactly the same number of clock cycles for one instruction. Then just summarizing, S is the number of stages, C is the number of clock cycles each stage takes to process an instruction, and I is the total number of instructions that you need to execute. So, if you have a non-pipelined architecture, what will happen? So, one instruction should take S number of stages, and each stage takes C clock cycles. So, S into C clock cycles will be required in a non-pipelined architecture for a single instruction.

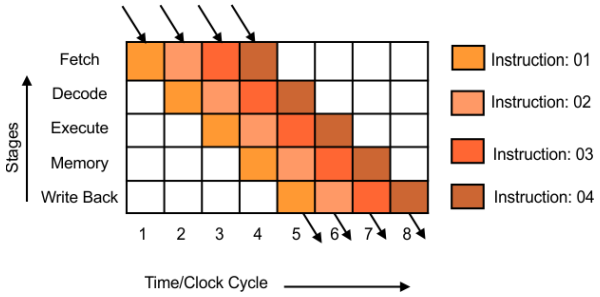
For non-pipelined, total time for execution $S \times C \times I$ clock cycles.

For Pipelined, total time for execution $(S + I - 1) \times C$

But if instructions are there, it will be multiplied by I. So total clock cycles will be S x C x I. Now, the assumptions that pipelined architecture makes are that each process is broken into S stages, each stage takes exactly 6 cycles, and the stages are fully pipelined, which means multiple instructions can be processed simultaneously, one at each stage. and this pipeline is fed continuously with new instructions that means there is no lag or delay. So, once one instruction is gone, the next is sent to the pipeline, and then the processor will start to produce results after the pipeline has been filled, that is, after S number of stages.



So, to fill the pipeline, how much time or clock cycles do we need, S x C, that is, like for one instruction to fill the pipeline? So once one instruction is done, what is the remaining number of instructions, that is I -1, and these will come in each like C number of clock cycles. So, the total time to process will be this plus this, which is S times C plus I minus 1 times C, and so this is the formula and if you consider c to be equal to 1, meaning that each instruction will take 1 clock cycle per stage, then this is very much similar to the convolution operation. And if you see it graphically, the operation is very much similar to convolution. Like this is for a regular non-pipelined architecture, where if I say I have one clock cycle for an instruction, there are four instructions and five stages.



So, this is what is happening. So, first instruction, fetch, decode, execute, memory, write back. Once this first one is finished, then only I can start the second one. And then the third, and then the fourth. So, the total will be 5 times 4, which is 20 clock cycles. But in pipelined architecture, as I showed in the formula, it will be

$$S + I - 1.$$

which is 8 and this is the graphical or the visualized version of this pipeline architecture like when my first instruction is decoded, second instruction is being fetched and so on. So, once this first instruction is done, my second instruction has completed 4 stages, and my third instruction has completed 3 stages, and so on. So now, coming to something that is called full parallelism: what is parallelism? Now parallelism means you can perform multiple things simultaneously. So, there are several types of parallelism. One is instruction-level parallelism, and the other is data-level parallelism.

So, in instruction level parallelism, what is happening is that you can execute multiple instructions simultaneously. Like the processor is not treating a single instruction at a time; rather, it is treating multiple instructions that are independent of each other, and that is very important. If they are dependent, then you cannot process them in parallel, right? So, for example, if one operation is A plus B and E is C plus D, you cannot do this in parallel because C depends on the first operation. So, this cannot be parallel. But if the operations are, and then another is A plus B, and this is, say, D plus E, now these are independent of each other.

So, you can perform these operations in parallel. and this is instruction level parallelism. Data-level parallelism basically means you are doing the same operations but on different data. Let us see an example. So, an example of instruction-level parallelism is a very long instruction word. It is like, say for example, if a genie comes to you and says that he will grant you 3 wishes, but you have 5 wishes.

So, if you are exhausted with three, you have the option to select only the top three wishes, or you can do one thing: select the top two wishes and combine the remaining three wishes as a single wish. That is basically your very long instruction words like these are executed in parallel, and this big word or big instruction word is treated as a single instruction although this big instruction word actually contains a lot of small instructions. Now, there are a lot of things that need to be considered, such as how many operations can be executed simultaneously, which depends on the number of functional units available; that is the first thing, and how many operations can be packed into one single instruction word.

Like you cannot just pack all your wishes into a single wish. So, that has to be logical. The second example of parallelism is single instruction multiple data, or SIMD, and this is data-level parallelism. So here what is happening is that these days our GPUs are working. So, GPUs are performing the same operation, which is a convolution, but on different data.

So, here data is usually packed into wide registers like 128-bit and 256-bit registers, and this allows multiple data elements to be processed in parallel within the same instruction

cycle. So, this SIMD processor can add multiple pairs of numbers in parallel with a single addition instruction. So, this instruction is single and simple, but the data are huge. And this is an example. If you have to solve this problem with the vectors (1, 2, 3, 4) and (5, 6, 7, 8), you need to add these two vectors.

A=[1,2,3,4]

B=[5,6,7,8]

Then A+B=[6,8,10,12]

So, what will you do? You will first add 1 plus 5, which is 6; 2 plus 6, which is 8; 3 plus 7, which is 10; and 4 plus 8, which is 12. But if I take 4 people and ask the first guy to add this, the second guy adds this, the third guy adds this, the fourth guy adds this, and when I say add, I get the result together in one clock cycle. But if one person is doing it, he needs to do it in four clock cycles.

So that is SIMD. So, thank you so much. We will meet again in another lecture.