IN3160, IN4160

Timing, pipelining

Yngve Hafting

Beskjeder:

Course Goals and Learning Outcome

https://www.uio.no/studier/emner/matnat/ifi/IN3160/index-eng.html

In this course you will learn about the **design of advanced digital systems**. This includes programmable logic circuits, a hardware design language and system-on-chip design (processor, memory and logic on a chip). Lab assignments provide practical experience in how real design can be made.

*After completion of the course you will*:

- understand important **principles for design** and testing of digital systems
- understand the relationship between behaviour and different construction criteria
- be able to describe advanced digital systems at different levels of detail
- be able to perform simulation and synthesis of digital systems.

Goals for this lesson:

- Know terms and principles for
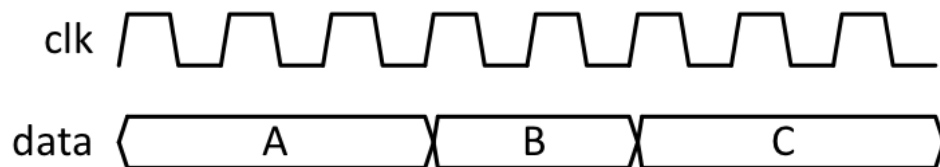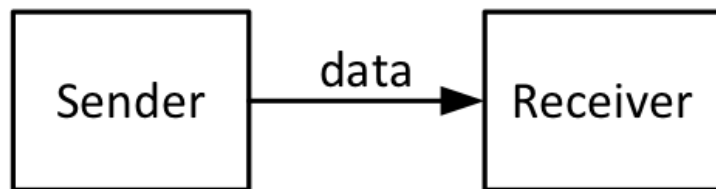   - timing
    - flow control
    - pipelining

Interface Timing

- How do you pass data from one module to another?
  - Open loop
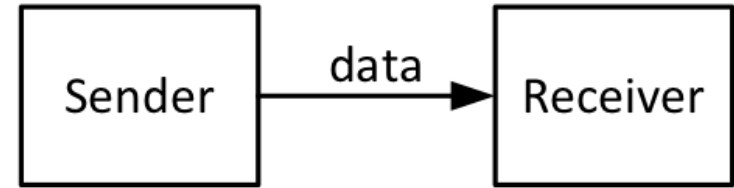 - Flow control
  - Serialized
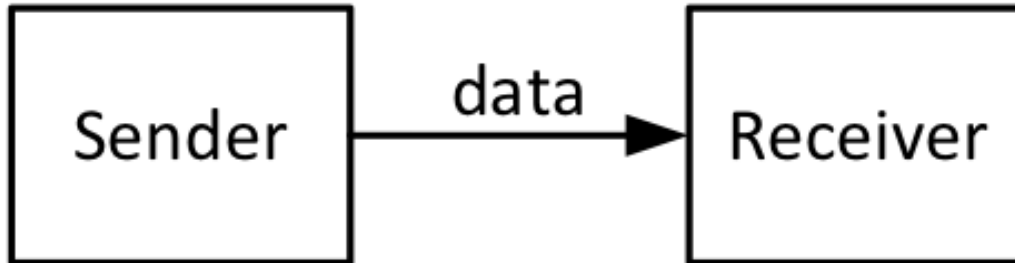
Always Valid Timing



- Current data always valid
  - Measurment data, such as
    - Temperature
    - Position
    - O10 PID control
    - Etc.
  - Static/ constant data

- Dropping data not critical
  - Sequence does not matter

- When crossing clock domains =>
  - Synchronization needed to avoid metastability
  - Synchronization needed to avoid errors
    - Ex. error: 1000 => 0111 being read as 1111
  - *Can* be passed without flow control
    - Signals unchanged from one cycle to the next *is* valid
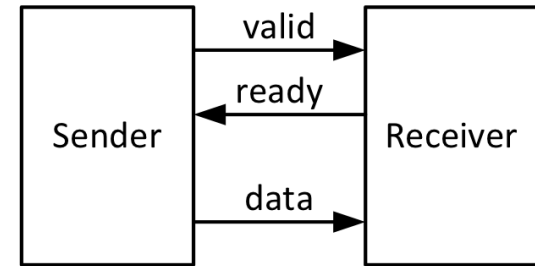   - No need to re-send data if there are errors.

Periodically Valid Timing

- Data only valid in predefined intervals
  - Ex: an 8 bit shifter has one byte ready every 8th clock cycle.
  - Ex: Cryptographic keys that need to be decrypted using the previous key

- Dropping data may be unacceptable

- Flow control is *required* when crossing clock domains
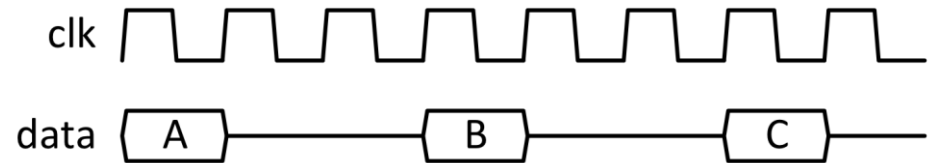
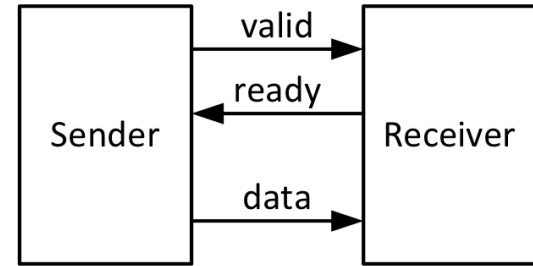# Flow Control

- When crossing clock domains:
    - Multiplexer / Enable synchronizer
        - data valid signal (=data ready..)

    - Handshake synchronizer
        - Data valid + receiver ready
          (= request + acknowledge)

    - FIFO synchronizer

- Flow control is also used between modules within a clock domain
    - *CDC is considered being taken care of for the rest of this lecture.*
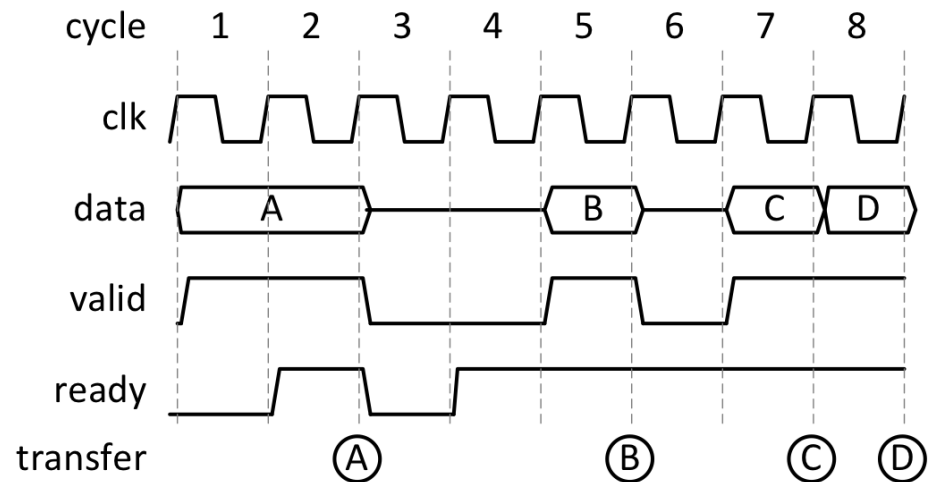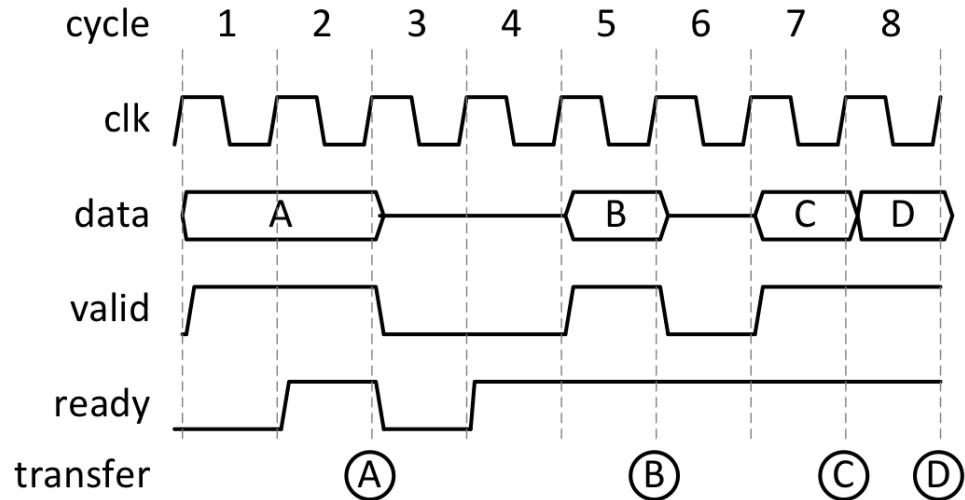
Flow-control types

- Valid – Transmitter (Tx) has data available
- Ready – Receiver (Rx) is able to take data

- Push flow control
  - assume Rx always Ready
- Pull flow control
 - assume Tx always Valid

Serialization

- Serialization is often used when dealing with large portions of data
  - Further parallelization is not practical.
- Requires some sort of convention on how data is sent

cycle   1   2   3   4   5   6   7   8   9   10   11   12   13

clk

data a_3 a_2 a_1 a_0 b_3 b_2 b_1 b_0 c_3 c_2 c_1 c_0
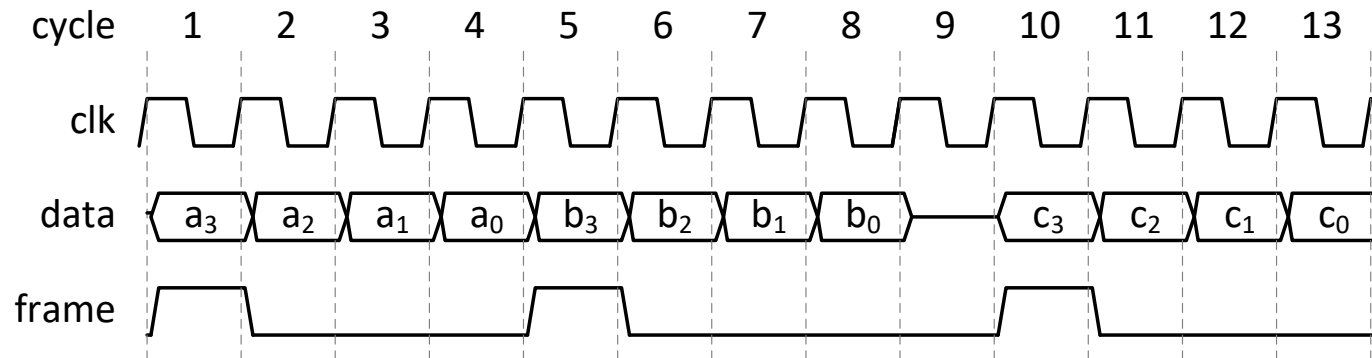
frame

- Frame signals start of new serial frame (here: a packet of 4 words)
  - *An example of push-flow control.*
- Flow control can be at frame granularity or word granularity

Serialization with word granularity flow control
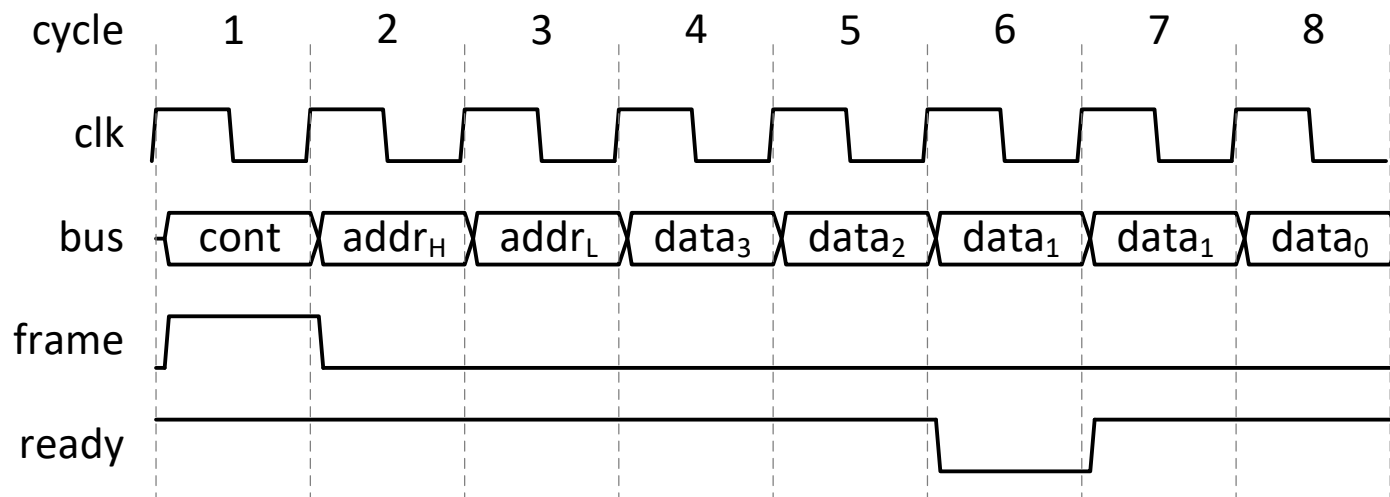
- Two way flow-control
  - Predefined packet size
    - New data only when receiver is *ready*

Packet size

- Often the packet size is given
  - ex. UART: usually 8 bit character+*(parity)*+start/stop bit

| start bit | bit 0 | bit 1 | bit 2 | bit 3 | bit 4 | bit 5 | bit 6 | bit 7 | stop bit |

- Varying packet sizes requires logic that determines packet size from data or additional flow control.

- Inside a chip, data is mostly passed in parallell

- Outside a chip it is normal to serialize
  (to reduce number of wires, avoid the n-bit problem)

Isochronous timing

- Data is sent with regular time intervals

- Isochronous timing is required when data "must" be read in a certain timeframe
  - Examples:
    - Screen output when playing video
    - Music or speech

- Ex. USB devices can be set up having isochronous endpoints which ensures a certain amount of data always can be transfered from a device, such as a microphone.

  - The USB host will then have to set up interrupts to poll the data from the device regularly.

Interface timing summary

- Always vs Periodically valid

- Flow control (FC)
    - Valid: Push
    - Ready: Pull
  - CDC synchronization may use Flow control
  - Periodically valid signals need FC regardless of CDC

- Serialization uses FC
  - Frame+ready
  - Packet sizes must be defined

- Isochronous timing
  - Periodically sending
  - For time-critical data, such as AV-streams.

Pipelining content

- Terminology

- Parallelization vs pipelining

- Example: 32 bit ripple carry adder

- Stalls

- Load Balance

- Resource sharing

Pipelining terms

- Throughput (Θ)
  - tasks performed per unit time
    - MIPS : Millions instructions per second
    - FLOPS : Floating point operations per second
    - etc

- Latency (T)
  - The time needed to complete one task fully

Parallellized vs pipelined
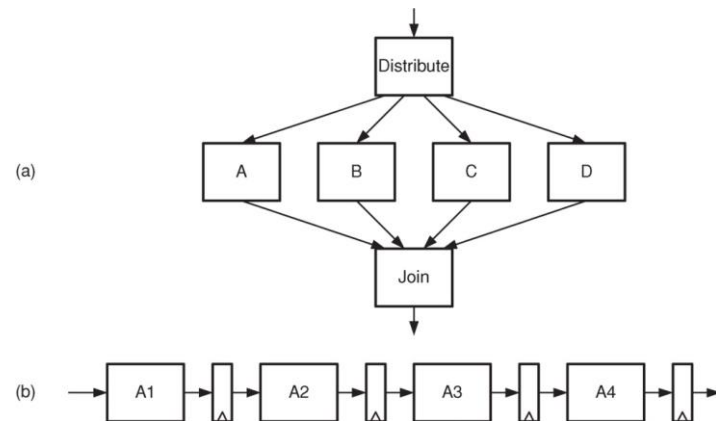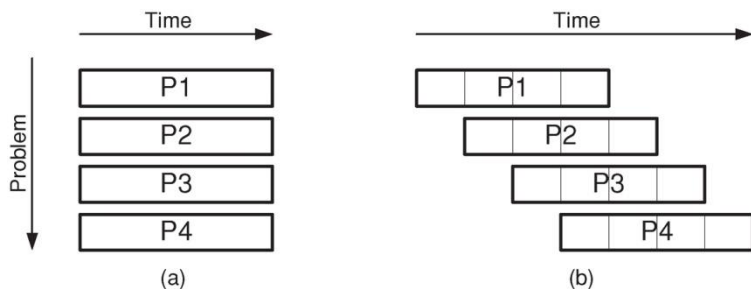
a) Needs 4x HW to achieve compared to solving one task

- Here: *Throughput*, $\Theta = 4x$

b) Needs registers for each pipeline stage

- can run on a higher clock frequency than a).
- $\Theta \leq 4x$
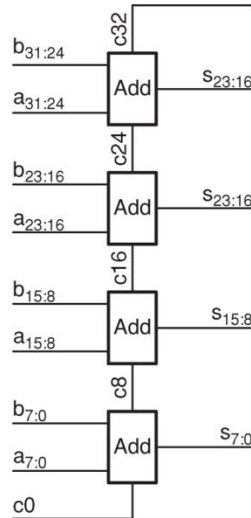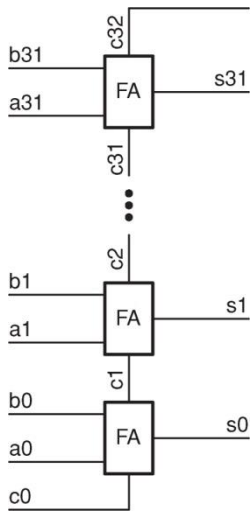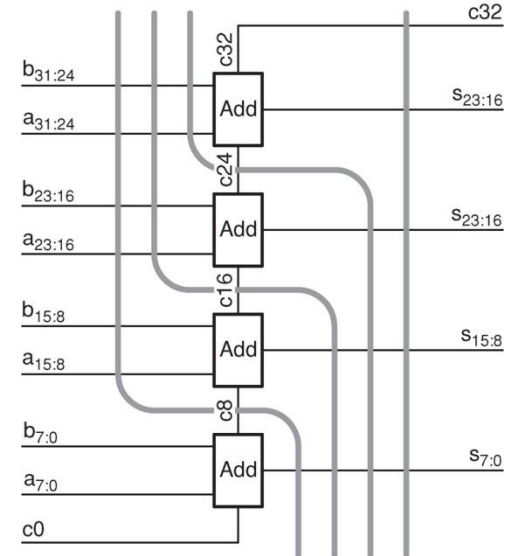
32 bit ripple carry adder continued
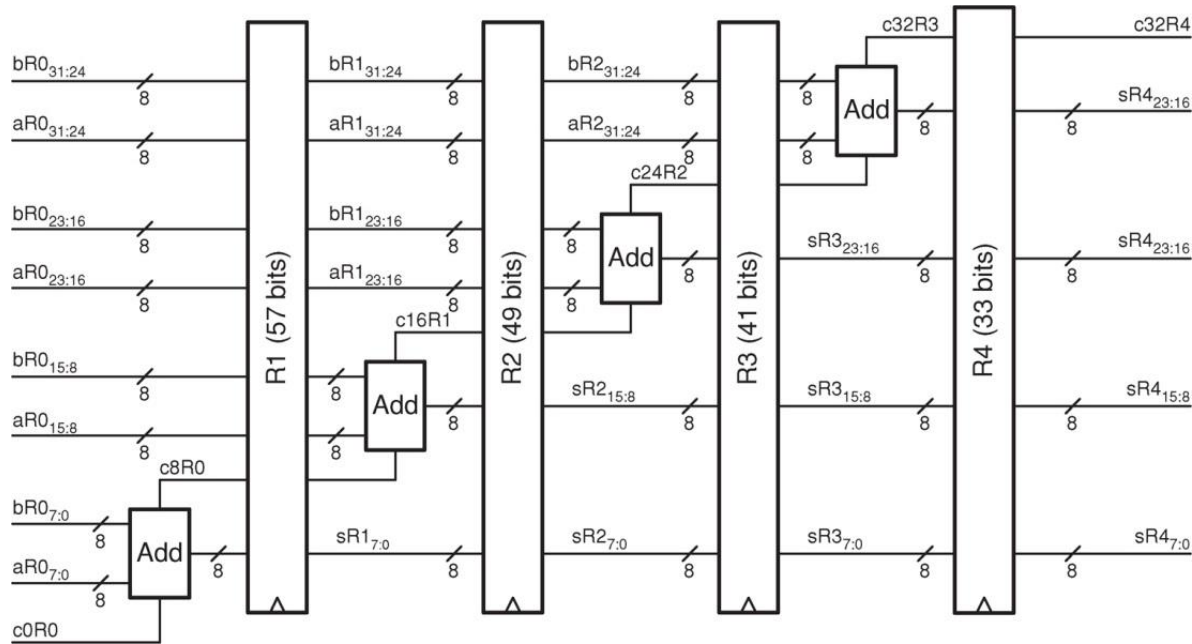
t_{FA}: 100 ps

t_{Reg}: 200 ps

$t_{cycle} = 8\ t_{FA} + t_{Reg}$
$= 800ps + 200ps = 1ns$

Latency $T = 4*t_{cycle}$
$= 4ns$

$\Theta = 1Gops$

Try: What would be the latency and throughput if we use four bits per pipeline stage?

- $t_{cycle} = 4t_{FA} + t_{Reg} = 400ps + 200ps = 600\ ps$
- Latency $T = 8*t_{cycle} = 600ps*8 = 4,8ns$
- Throughput $\Theta = 1op/600ps = 1,67$ Gops

Pipelines

- Pipelines may have stall and idle functionality...
- When should these happen? How can you prevent them?
- Max latency vs. average latency (absorbing bursts)
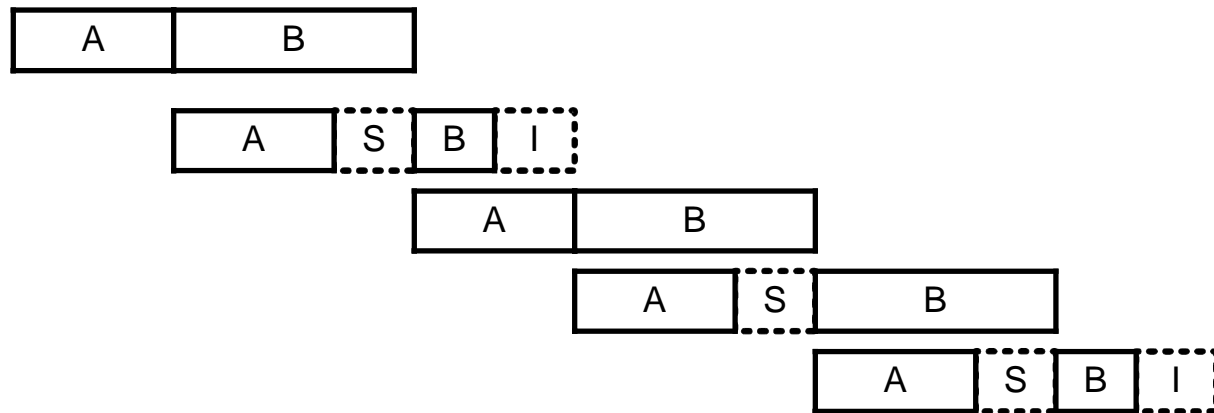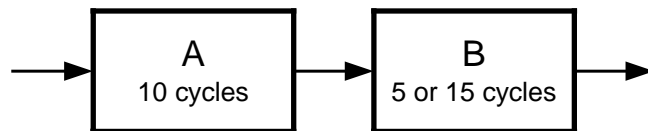
Pipeline stalls (1)

- Variable execution time may occur in larger systems.
  - Ex: A floating point operation in a series of calculations that mostly are integer based

- Flow control is needed in the pipeline
 - Each stage has its own data_valid and ready signal

Pipeline stalls (2)

(a)
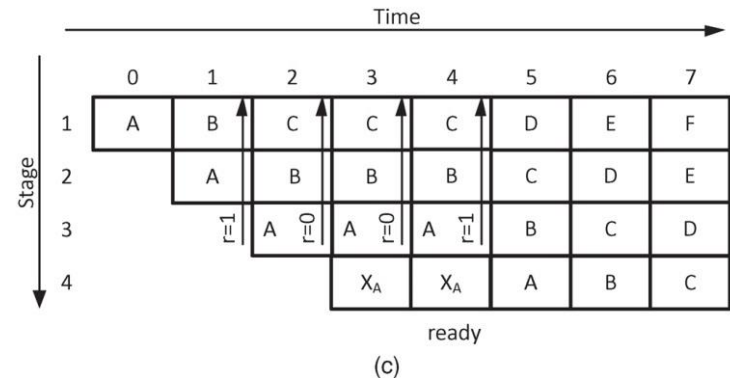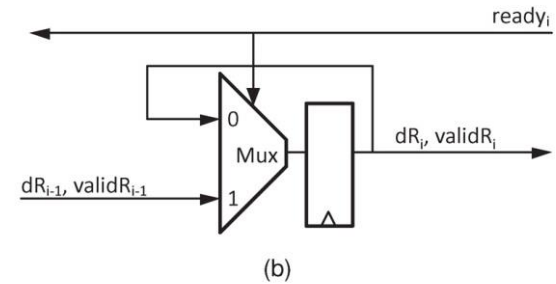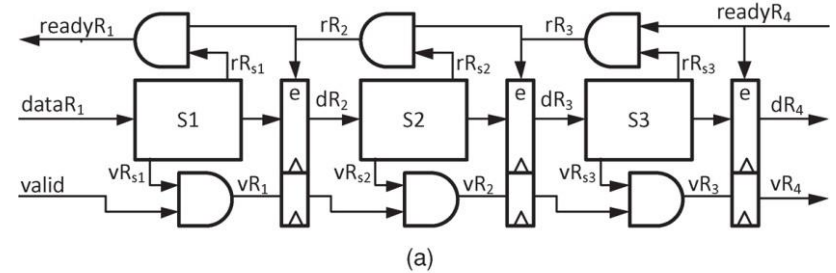
- When a stage is not ready, either
  - the whole pipeline stalls (previous slide)
  - or the results need to be double buffered to absorb the delay
    - (much like an accordion)
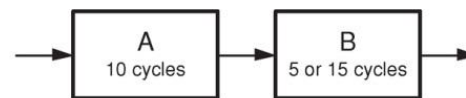   - ready signal is buffered upstream

(b)

(c)
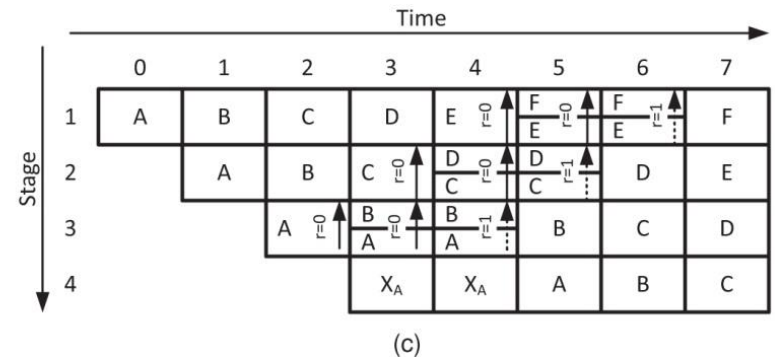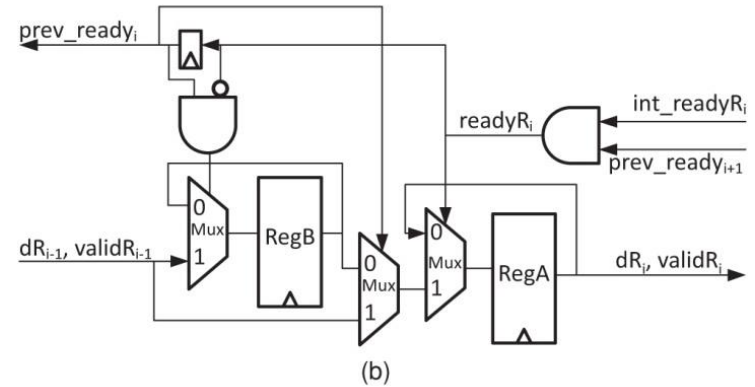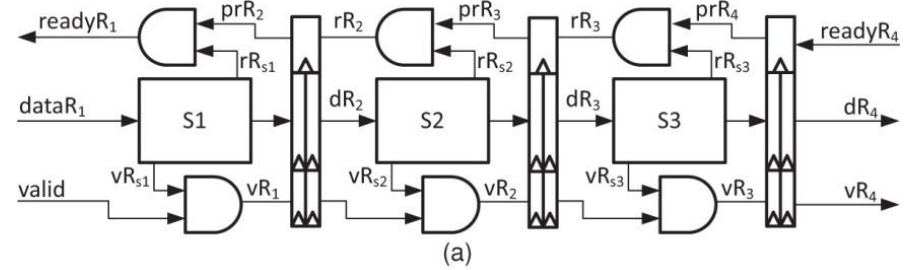
Load balancing

- One or more of the stages in a pipelining doesnt meet timing requirement:

  => we can sometimes
    - pipeline that stage internally
    - parallellize that stage
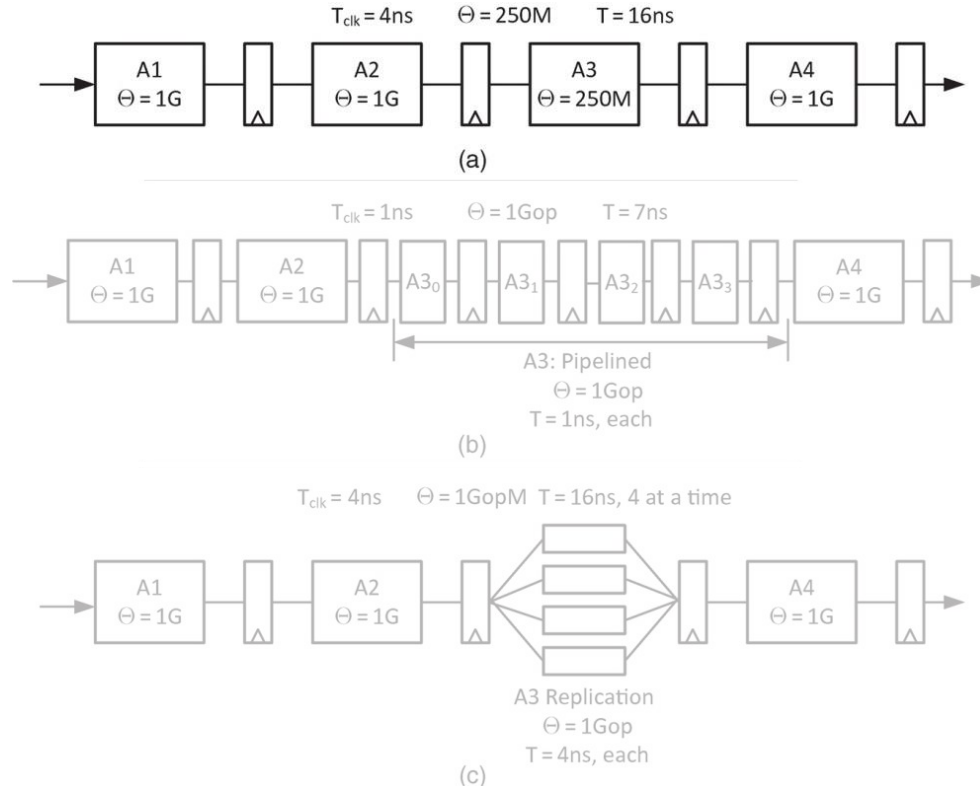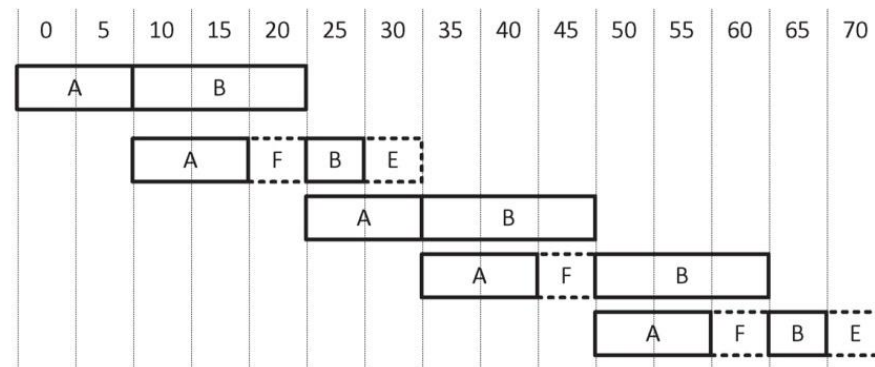
Variable loads

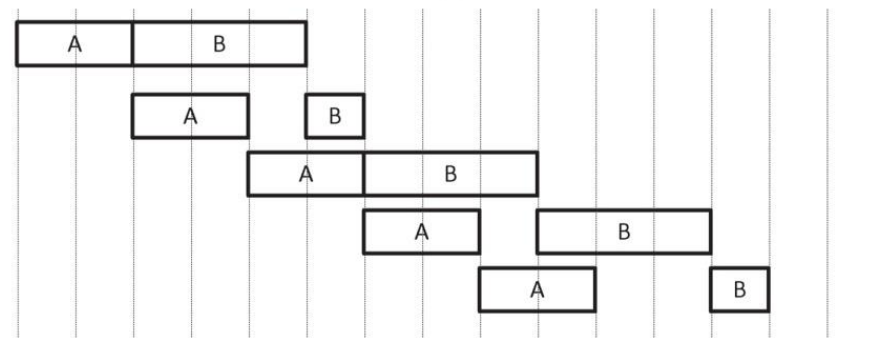- Using a FIFO between stages with variable loads may ensure throughput

Resource sharing

- When sharing resources
  - Use an *arbiter* to sort who can use the resource
    - a): which stage in a pipeline
    - b): which pipeline

- Within a pipeline (a)
  - the arbiter (priority encoder) should prioritize the stage furthest down stream
    - *to avoid deadlock.*

- Between separate data paths:
  - avoid *starvation* (one being stalled at all times)
  - => *use a toggle, round robin scheme, etc*
    - *Implement arbitration using an FSM*

(a)

(b)
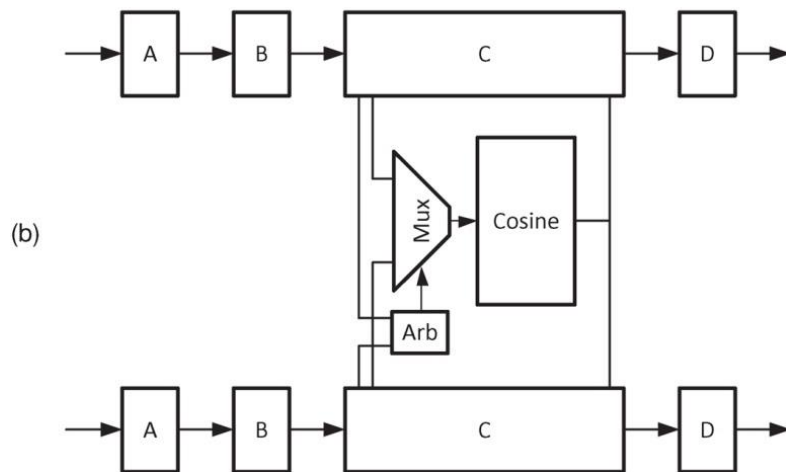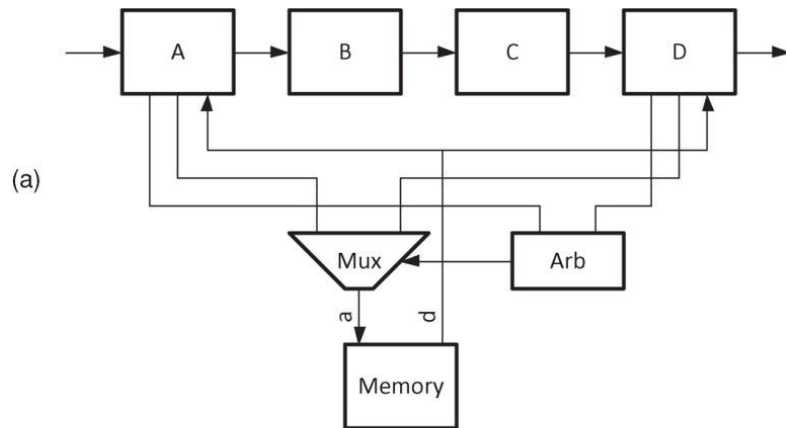
Summary

- Terms for timing:
  - Always or periodically valid
  - Flow control
    - Push – Pull – Two way
- Simple pipelines:
  - adding registers between operations that can be split

- Advanced pipelines (*Multi module systems*):
  - Stalling
    - Flow control
    - Double buffering
  - Load balancing
  - Resource sharing
    - arbitration

Suggested reading

DHA:

- 22 p479-494
- 23 p497-518