Intro. to Computer Architecture

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
⟨csdsp@bristol.ac.uk⟩

November 3, 2017

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
   - the associated notes page may be pre-populated with extra, written explaination of material covered in lecture(s), plus
   - anything with a "grey'ed out" header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

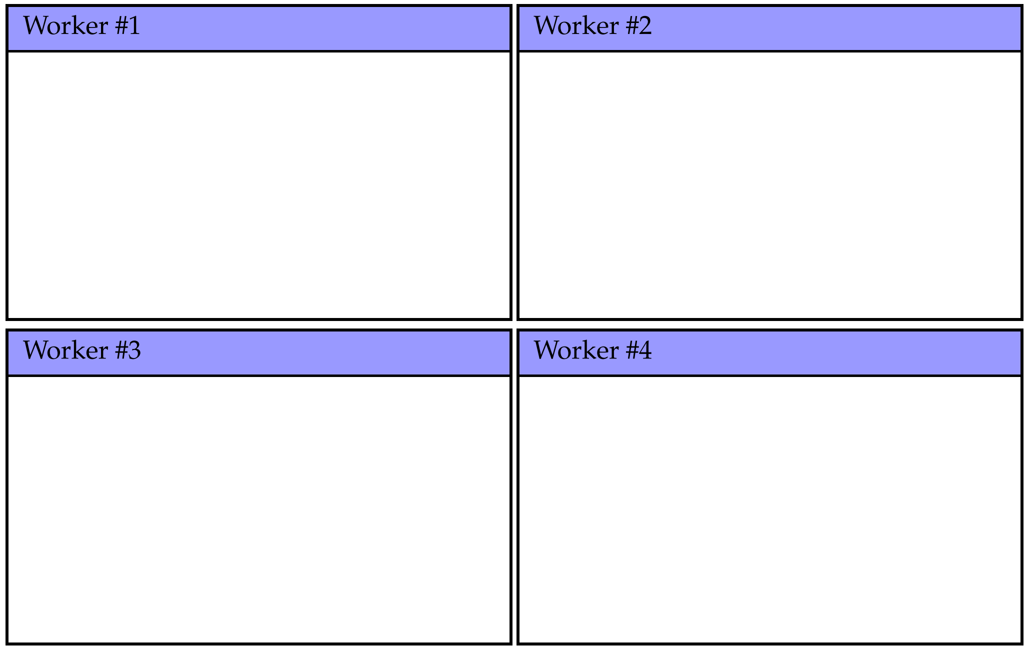
Notes:

COMS12200 lecture: week #6

- Agenda:
  1. comments, questions, recap, then
  2. a general-purpose implementation technique and related trade-offs, namely
     - the concept of **pipelining**, and
     - trade-offs between area, latency, and throughput.

Notes:

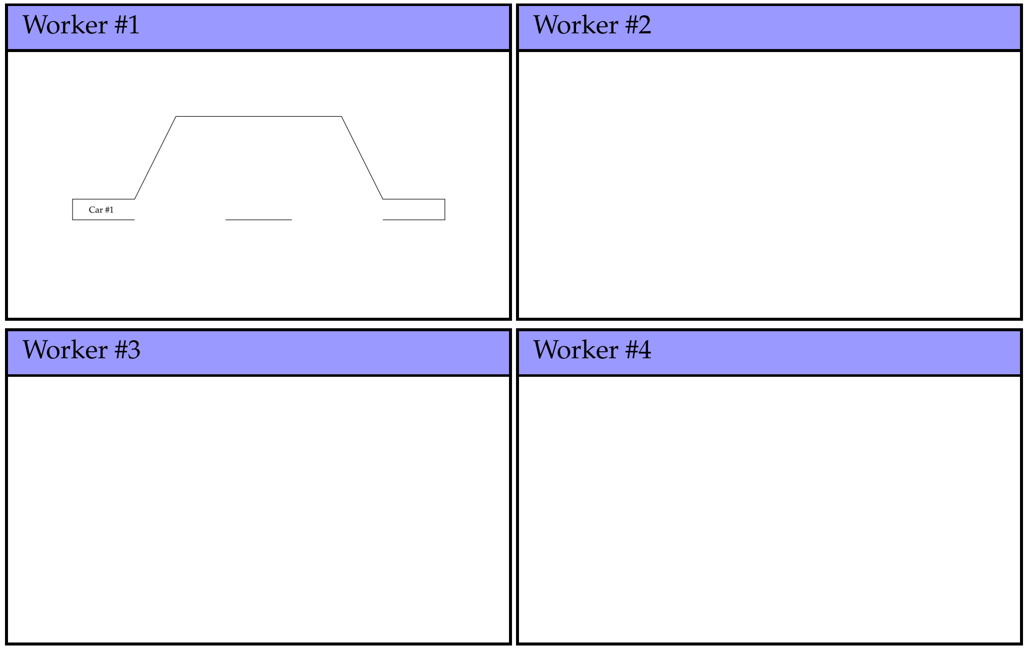
Sequential use of combinatorial stages ≃ pipelining (1)
An analogy: production lines



Notes:

Sequential use of combinatorial stages ≃ pipelining (1)

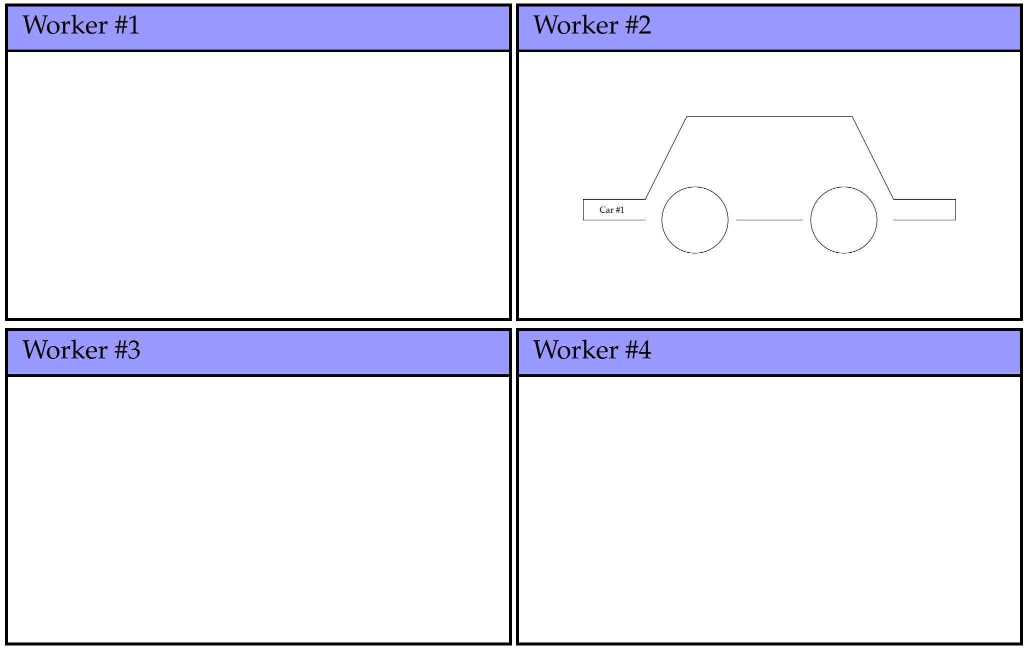
An analogy: production lines



Notes:

Sequential use of combinatorial stages ≃ pipelining (1)

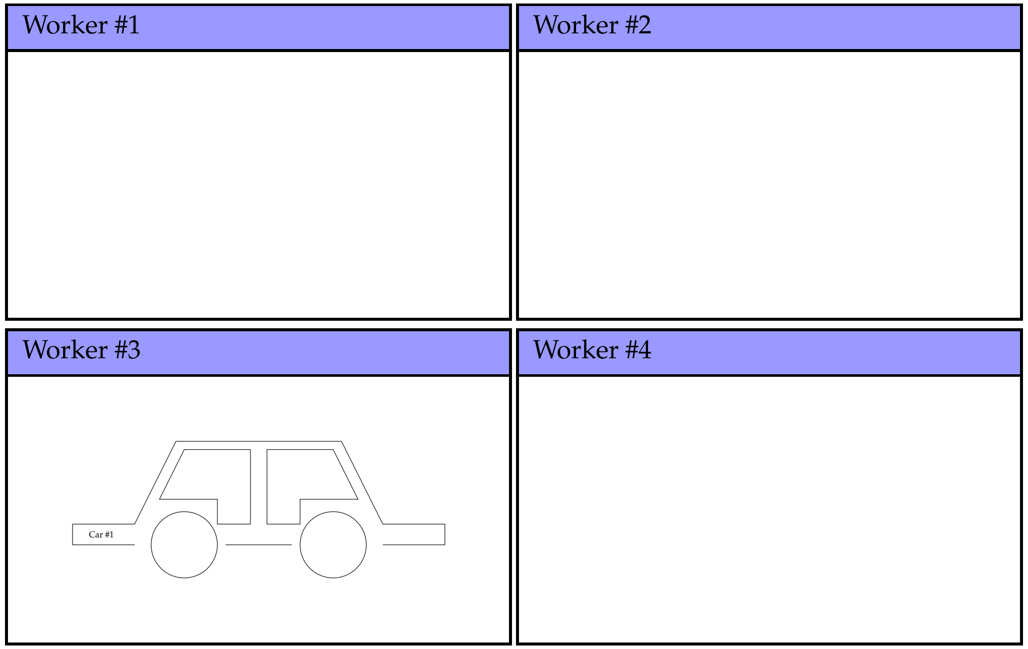
An analogy: production lines



Notes:

Sequential use of combinatorial stages ≃ pipelining (1)

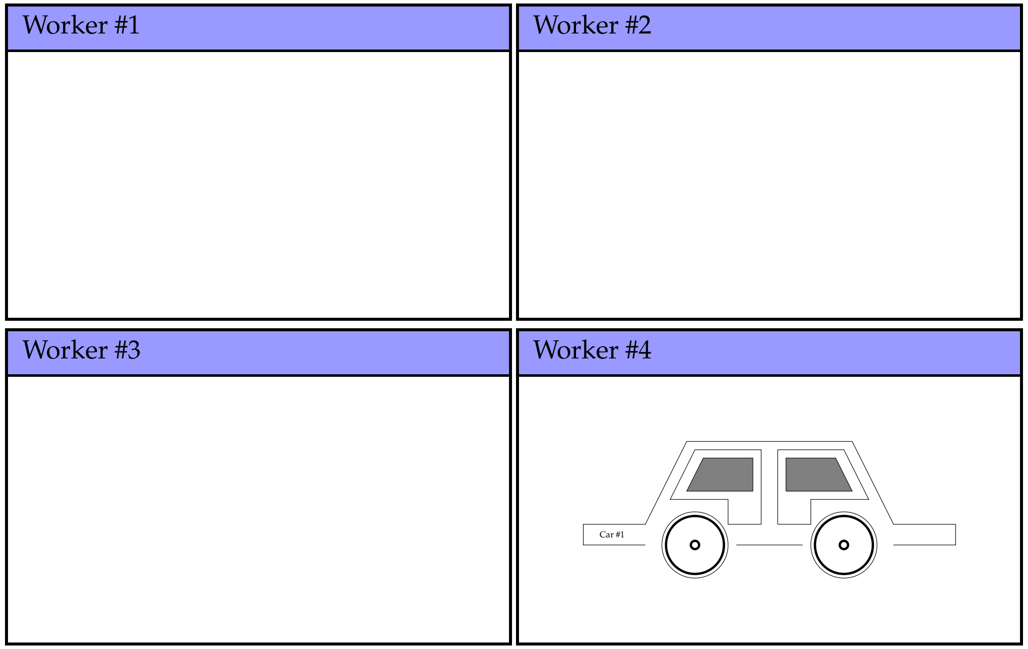
An analogy: production lines



Notes:

Sequential use of combinatorial stages ≃ pipelining (1)

An analogy: production lines



Notes:

Sequential use of combinatorial stages ≃ pipelining (1)

An analogy: production lines

Worker #1

Worker #2

Worker #3

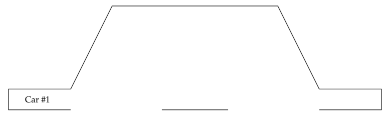
Worker #4

Notes:

Sequential use of combinatorial stages ≃ pipelining (1)

An analogy: production lines

Worker #1



Worker #2

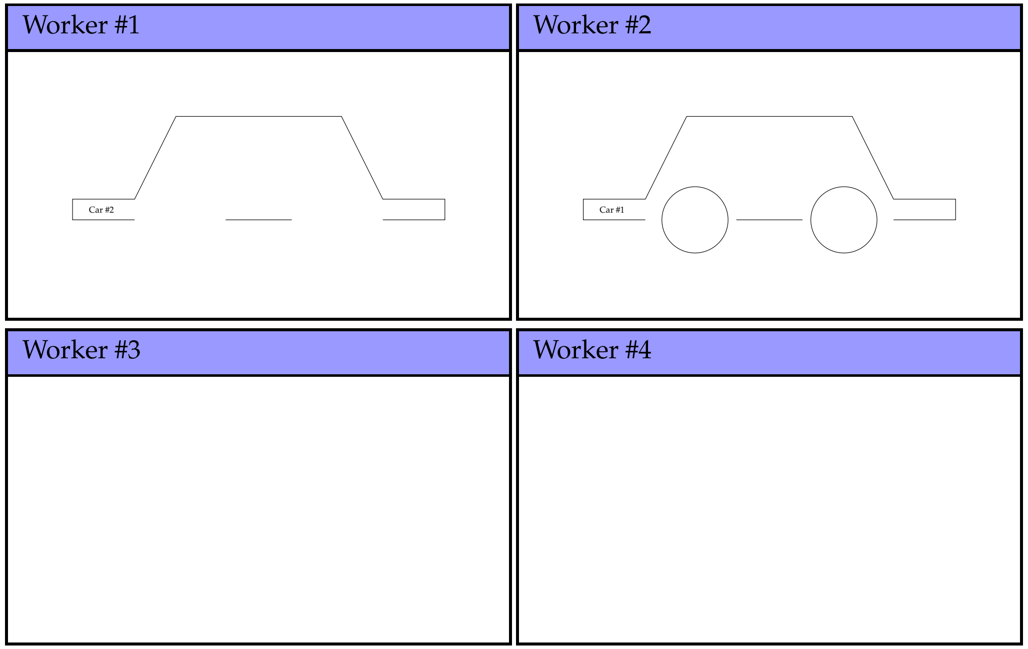
Worker #3

Worker #4

Notes:

Sequential use of combinatorial stages ≃ pipelining (1)

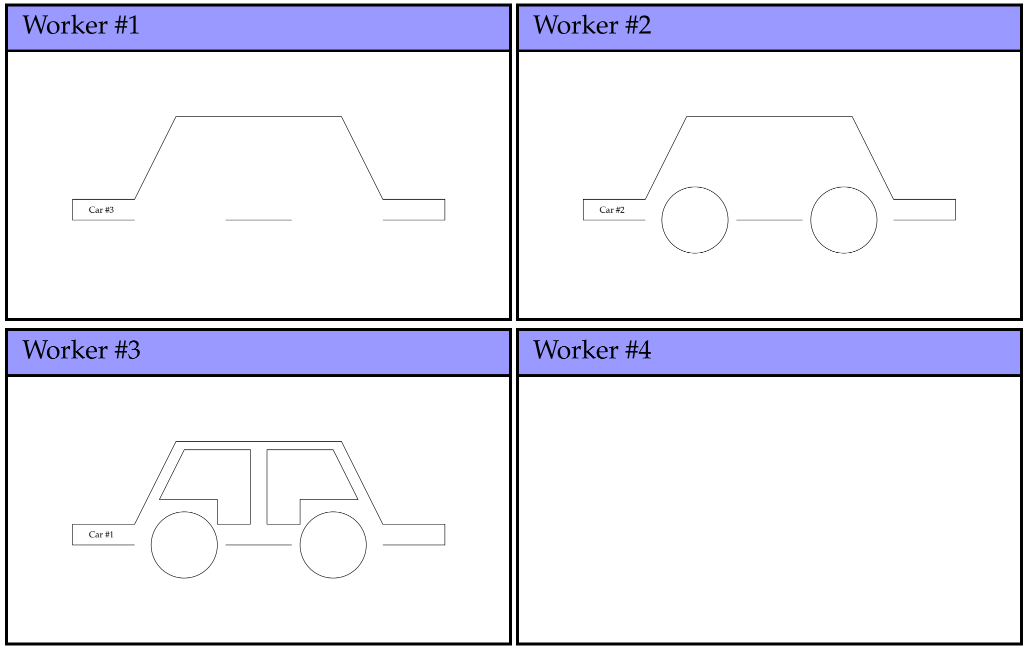
An analogy: production lines



Notes:

Sequential use of combinatorial stages ≃ pipelining (1)

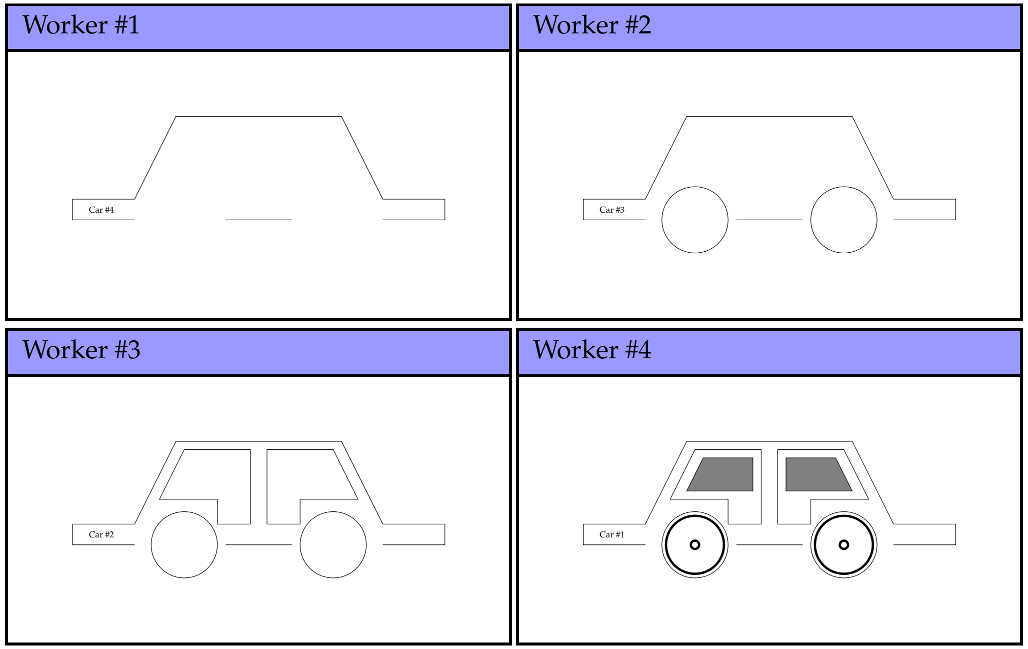
An analogy: production lines



Notes:

Sequential use of combinatorial stages ≃ pipelining (1)

An analogy: production lines



Notes:

Sequential use of combinatorial stages ≃ pipelining (2)

An analogy: production lines

Definition

The **latency** is the total time elapsed before a given input is operated on to produce an output; this is simply the sum of the latencies of each stage.

Definition

The **throughput** (or **bandwidth**) is the rate at which new inputs can be supplied (resp. outputs collected).

- Good news:
  - Based on the metrics above we find

    $$\begin{array}{lll} \text{without pipeline} & \leadsto & \text{latency} = 4, \quad \text{throughput} = 1/4 \\ \text{with pipeline} & \leadsto & \text{latency} = 4, \quad \text{throughput} = 1 \end{array}$$

    suggesting that an (idealised) n-stage yields an n-fold improvement in throughput.
  - The production of one car is *still* sequential, but we used parallelism to *hide* latency wrt. a given car.

Notes:

Sequential use of combinatorial stages ≃ pipelining (2)
An analogy: production lines

Definition

The **latency** is the total time elapsed before a given input is operated on to produce an output; this is simply the sum of the latencies of each stage.

Definition

The **throughput** (or **bandwidth**) is the rate at which new inputs can be supplied (resp. outputs collected).

- Bad news:
  - If we can't **advance** the production line for some reason (e.g., a stage is delayed), we say it has **stalled**; this is bad, since it reduces utilisation.
  - Advancement is limited by the slowest stage; to minimise idle time, balance is needed between the workload of stages.
  - Even then, there is overhead associated with *all* stages (e.g., communicating between stages takes some time).

Notes:

Sequential use of combinatorial stages ≃ pipelining (3)
Application to digital logic

- Idea: apply the production line approach to *circuits*, i.e.,

1. split some combinatorial logic X into a **pipeline** of n stages, say X_i for $0 \leq i < n$,
2. have each stage perform one step of the computation,
3. allow with multiple independent computations to overlap, with partial (or active) cases deemed **in-flight**.

Notes:

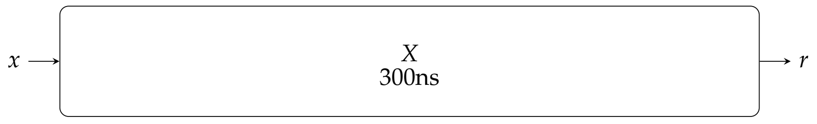
Sequential use of combinatorial stages $\simeq$ pipelining (3)
Application to digital logic

- Problem #1: how to split X into stages, or rather
  1. *where* should we split it (which depends heavily on what X *is*), and
  2. *how* should we split it?
- Problem #2: how do we control the resulting pipeline?

Notes:

Sequential use of combinatorial stages $\simeq$ pipelining (4)
Application to digital logic

- Solution #1: consider a (simplistic) example,

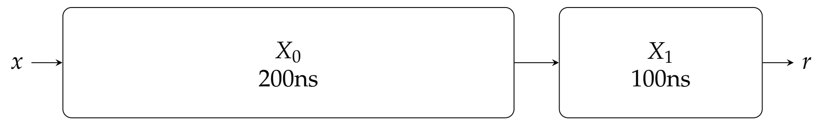


  from which we can infer that
  - the slowest stage dictates how fast we can advance the pipeline, so
  - we need to balance the stages st. idleness is minimised (i.e., we avoid one stage waiting for another).

Notes:

Sequential use of combinatorial stages $\simeq$ pipelining (4)
Application to digital logic

- **Solution #1**: consider a (simplistic) example,



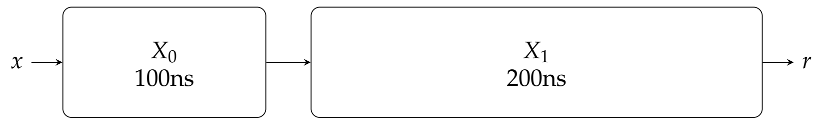
from which we can infer that

- the slowest stage dictates how fast we can advance the pipeline, so
- we need to balance the stages st. idleness is minimised (i.e., we avoid one stage waiting for another).

Notes:

Sequential use of combinatorial stages $\simeq$ pipelining (4)
Application to digital logic

- **Solution #1**: consider a (simplistic) example,



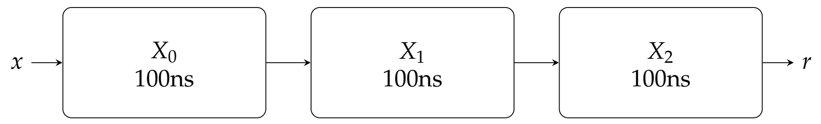
from which we can infer that

- the slowest stage dictates how fast we can advance the pipeline, so
- we need to balance the stages st. idleness is minimised (i.e., we avoid one stage waiting for another).

Notes:

Sequential use of combinatorial stages ≃ pipelining (4)
Application to digital logic

- Solution #1: consider a (simplistic) example,



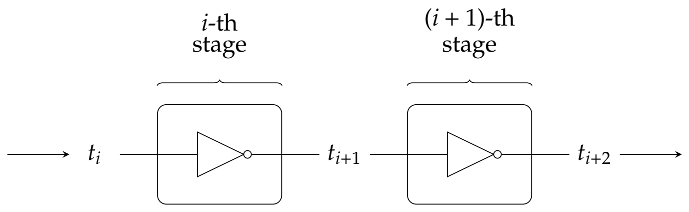
  from which we can infer that
  - the slowest stage dictates how fast we can advance the pipeline, so
  - we need to balance the stages st. idleness is minimised (i.e., we avoid one stage waiting for another).

Notes:

Sequential use of combinatorial stages ≃ pipelining (5)
Application to digital logic

- Solution #2:
  1. insert a **pipeline register** between adjacent stages, and
  2. advance the pipeline on each positive edge of some control signal,

  i.e.,



  where
  1. for a *synchronous* pipeline, *adv* is basically a global clock signal, but
  2. for an *asynchronous* pipeline, some form of protocol between the stages controls adv_i: in a sense the stages decide when to advance when they're ready.

Notes:

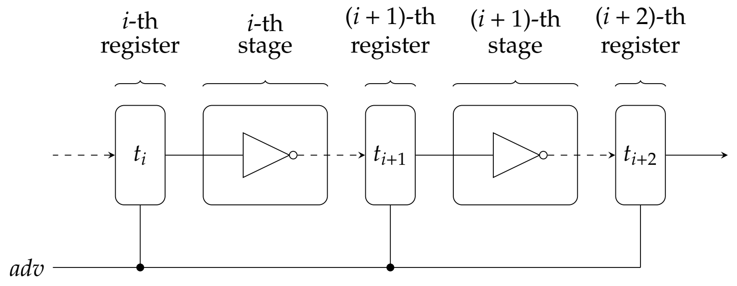
- You should view the solid lines as permanent connections, and the dashed lines as connections that are used during update: the dashed lines hence isolate the stages from each other, so they only interact at the point when some R_{i+1} is updated to hold t_i.

Sequential use of combinatorial stages ≃ pipelining (5)

Application to digital logic

- Solution #2:
  1. insert a **pipeline register** between adjacent stages, and
  2. advance the pipeline on each positive edge of some control signal,

  i.e.,



  where

  1. for a *synchronous* pipeline, *adv* is basically a global clock signal, but
  2. for an *asynchronous* pipeline, some form of protocol between the stages controls adv_i: in a sense the stages decide when to advance when they're ready.

Notes:

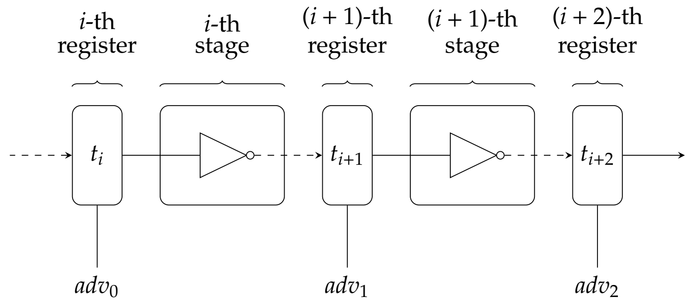
- You should view the solid lines as permanent connections, and the dashed lines as connections that are used during update: the dashed lines hence isolate the stages from each other, so they only interact at the point when some R_{i+1} is updated to hold t_i.

Sequential use of combinatorial stages ≃ pipelining (5)

Application to digital logic

- Solution #2:
  1. insert a **pipeline register** between adjacent stages, and
  2. advance the pipeline on each positive edge of some control signal,

  i.e.,



  where

  1. for a *synchronous* pipeline, *adv* is basically a global clock signal, but
  2. for an *asynchronous* pipeline, some form of protocol between the stages controls adv_i: in a sense the stages decide when to advance when they're ready.

Notes:

- You should view the solid lines as permanent connections, and the dashed lines as connections that are used during update: the dashed lines hence isolate the stages from each other, so they only interact at the point when some R_{i+1} is updated to hold t_i.

Sequential use of combinatorial stages ≃ pipelining (6)
Application to digital logic

- *Why*?!
  - Imagine in the j-th clock cycle the i-th stage X_i computes partial result t_i required by the $(i+1)$-th stage.
  - If the stages are connected directly via a wire,
    1. when the i-th stage changes t_i, this disrupts computation by the $(i+1)$-th stage,
    2. the i-th and $(i+1)$-th stages cannot perform different in-flight computations, because there is nowhere to store the associated (partial) inputs.
- So,

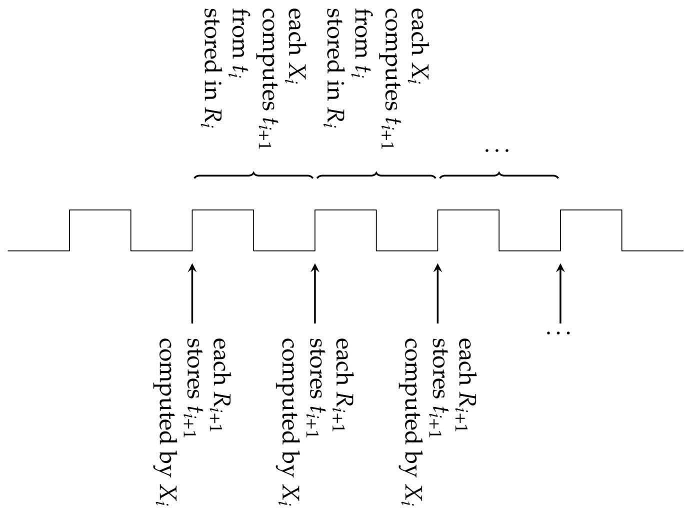
  +ve: If the stages are connected indirectly via a pipeline register, say R_i, the $(i+1)$-th stage can have a separate, stable input.

  −ve: Each pipeline register takes time to operate, so adds to the total latency: latency and throughput become a trade-off.

Notes:

Sequential use of combinatorial stages ≃ pipelining (7)
Application to digital logic

Example


stored in R_i from t_i each X_i computes t_{i+1}
stored in R_i from t_i each X_i computes t_{i+1}
. . .
each R_{i+1} stores t_{i+1} computed by X_i
each R_{i+1} stores t_{i+1} computed by X_i
each R_{i+1} stores t_{i+1} computed by X_i
. . .

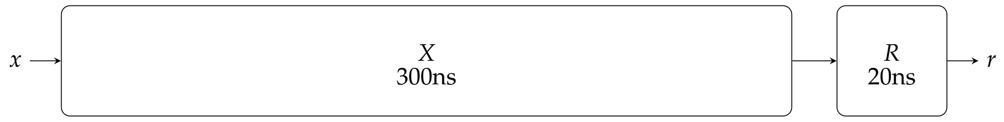
Notes:

Sequential use of combinatorial stages ≃ pipelining (8)

Example: a generic component

Example

Consider some generic, combinatorial logic X:



1. For the standard design
   - the latency is 300 + 20ns = 320ns, while
   - the throughput is 1/320ns = 3.12×10^6operations/s

   if we measure the latency of computing X and storing the result.

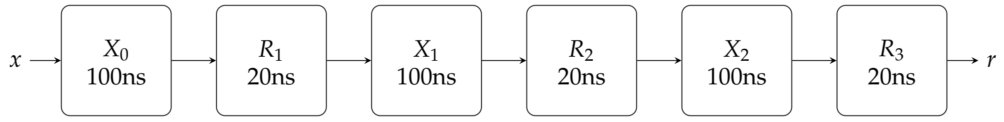
Notes:

Sequential use of combinatorial stages ≃ pipelining (8)

Example: a generic component

Example

Consider some generic, combinatorial logic X:

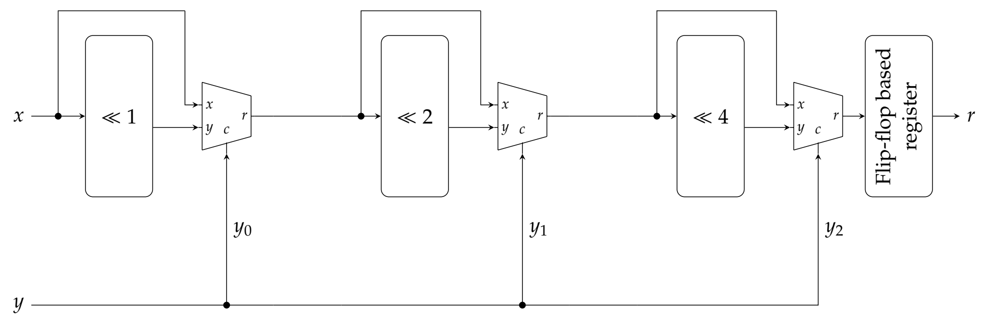


2. *But*, for a 3-stage pipeline (using the same measure)
   - the latency is 100 + 20 + 100 + 20 + 100 + 20ns = 360ns, while
   - the throughput is 1/120ns = 8.33×10^6operations/s.

Notes:

Sequential use of combinatorial stages ≃ pipelining (9)
Example: a pipelined (logarithmic) shift

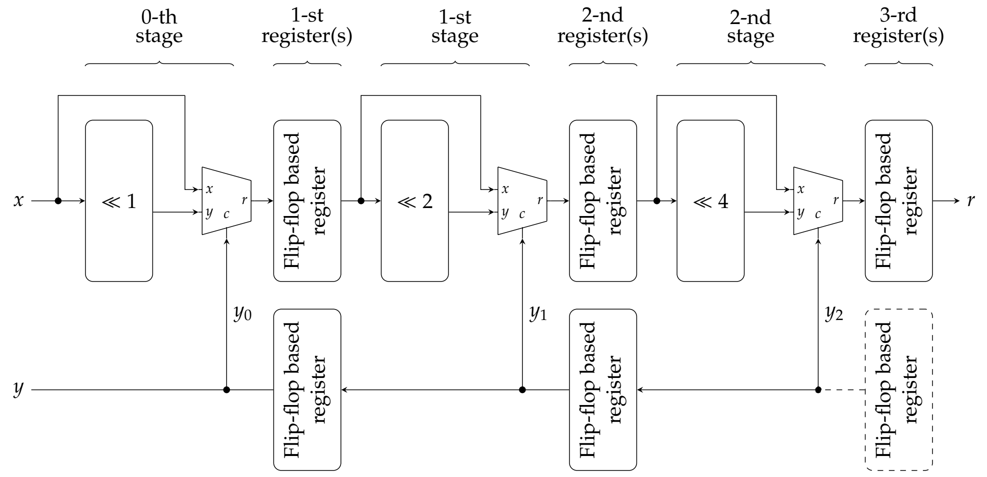
Circuit



Notes:

Sequential use of combinatorial stages ≃ pipelining (9)
Example: a pipelined (logarithmic) shift

Circuit



Notes:

Conclusions

- Take away points:
  1. In a sense, pipelining is a classic trade-off, i.e.,

     −ve: : higher area
     −ve: : higher latency
     +ve: : higher throughput

  2. Modulo some caveats, e.g.,
     - need to balance stages, and
     - need to keep the pipeline full,

     pipelining is a *general-purpose* technique: it can be applied, for example, to execution of *instructions* by a micro-processor ...

Notes:

Conclusions

- Example: **instruction pipelining** [1].
  - instruction execution follows a **fetch-decode-execute** cycle,
  - but *various* approaches can implement this behaviour, e.g.,

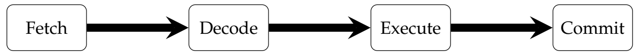
Fetch, Decode, Execute, Commit

  1. a non-pipelined design.

Notes:

Conclusions

- Example: **instruction pipelining** [1].
  - instruction execution follows a **fetch-decode-execute** cycle,
  - but *various* approaches can implement this behaviour, e.g.,

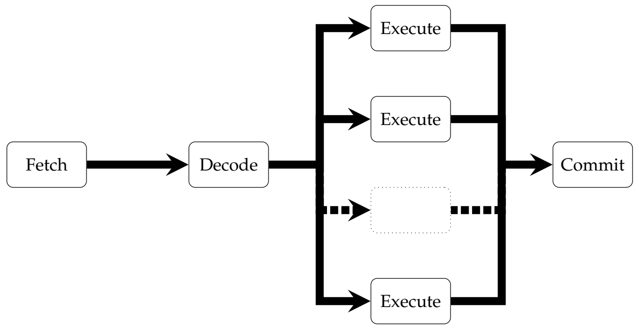


1. a non-pipelined design,
2. an **in-order** pipelined design.

Notes:

Conclusions

- Example: **instruction pipelining** [1].
  - instruction execution follows a **fetch-decode-execute** cycle,
  - but *various* approaches can implement this behaviour, e.g.,

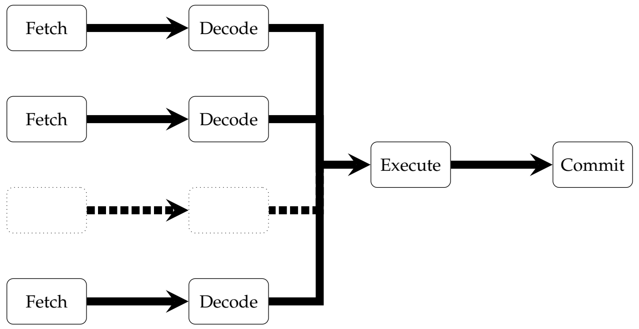


1. a non-pipelined design,
2. an **in-order** pipelined design,
3. an **out-of-order** pipelined design.

Notes:

Conclusions

- Example: **instruction pipelining** [1].
  - instruction execution follows a **fetch-decode-execute** cycle,
  - but *various* approaches can implement this behaviour, e.g.,

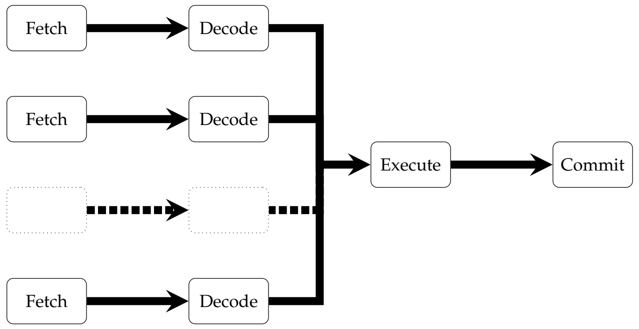


1. a non-pipelined design,
2. an **in-order** pipelined design,
3. an **out-of-order** pipelined design,
4. a **Simultaneous MultiThreaded (SMT)** [3] pipelined design.

Notes:

Conclusions

- Example: **instruction pipelining** [1].
  - instruction execution follows a **fetch-decode-execute** cycle,
  - but *various* approaches can implement this behaviour, e.g.,



1. a non-pipelined design,
2. an **in-order** pipelined design,
3. an **out-of-order** pipelined design,
4. a **Simultaneous MultiThreaded (SMT)** [3] pipelined design,

or combinations thereof (e.g., out-of-order *and* SMT).

Notes:

Additional Reading

- *Wikipedia: Throughput.* URL: http://en.wikipedia.org/wiki/Throughput.
- *Wikipedia: Pipeline.* URL: http://en.wikipedia.org/wiki/Pipeline_(computing).

Notes:

References

[1] *Wikipedia: Instruction pipeline.* URL: http://en.wikipedia.org/wiki/Instruction_pipelining (see pp. 63, 65, 67, 69, 71).

[2] *Wikipedia: Pipeline.* URL: http://en.wikipedia.org/wiki/Pipeline_(computing) (see p. 73).

[3] *Wikipedia: Simultaneous multithreading.* URL: http://en.wikipedia.org/wiki/Simultaneous_multithreading (see pp. 63, 65, 67, 69, 71).

[4] *Wikipedia: Throughput.* URL: http://en.wikipedia.org/wiki/Throughput (see p. 73).

[1] *Wikipedia: Instruction pipeline.* URL: http://en.wikipedia.org/wiki/Instruction_pipelining (see pp. 63, 65, 67, 69, 71).

[2] *Wikipedia: Pipeline.* URL: http://en.wikipedia.org/wiki/Pipeline_(computing) (see p. 73).

[3] *Wikipedia: Simultaneous multithreading.* URL: http://en.wikipedia.org/wiki/Simultaneous_multithreading (see pp. 63, 65, 67, 69, 71).

[4] *Wikipedia: Throughput.* URL: http://en.wikipedia.org/wiki/Throughput (see p. 73).

Notes: