Computer Architecture & Organization CS 520

Timothy N. Miller

(Course materials from Kanad Ghose)

About the Instructor

- Timothy Normand Miller
- 16 years of industry experience
  - Chip design, embedded systems, software
- PhD from Ohio State, 2012
  - Specialization: Computer Architecture

Prerequisites

- Coding skills
- Undergraduate logic design
- Assembly language programming
- Virtual memory and memory protection
- Undergraduate operating systems

Prerequisites

- Computer organization
- Instruction set design
- Opcode encoding, addressing modes
- RISC vs. CISC, load-store
- Data representation
- Computer arithmetic

Course Goals

- Thorough understanding of modern microprocessor design, primarily uniprocessor
- Performance, design tradeoffs, implementation
- Interaction/interface with compilers
- Influence on OS design

Course Materials

- Syllabus
- Homework assignments, project
- No text book
  - Online notes, papers, etc.
- Optional reading material:
  - *Computer Architecture, a Quantitative Approach*, Hennessey and Patterson
- Blackboard

Document password

C0rEKo0L16!

Homework

- 4 or so major assignments
- Design and coding assignments
- Some exam and quiz questions based on homework problems

Exams & Quizzes

- Pop quizzes to review specific topics
- Midterm
- Final

Class Project

- Coding assignment
- Two-phase
- Due around midterm and after final
- Typically involving some CPU simulator
  - Implementing a CPU yourself or
  - Making algorithm changes to msim

Class Participation

- Subjective
- Make sure I know you!
- Ask questions in class
- Show up for class
- Pop Quizzes
- Visiting during office hours

Grading

- Consider the grader's ability to understand your work
- Instructor will not override grader
- No give-away grade changes
- The grader's decision is final

Cheating

- Know the Academic Honesty Policy
- **Cheating will be reported to the school.**
- "Germany's defence minister has given up his doctoral title for good, after allegations that he had plagiarised sections of his thesis." -- BBC, 21 February 2011
  - http://www.bbc.co.uk/news/world-europe-12532877
- **DO YOUR OWN WORK**

What is cheating?

- Plagiarism: Representing someone else's work as your own.
- Improper sharing: Allowing another to represent your work as theirs.

Examples of cheating

- Copying answers from a classmate.
- Copying answers from the Internet or another course.
- Copying an algorithm.
- Sharing your answers with a classmate.
- Any case where you did not do your own work.
- All cheating is of the same magnitude.

Not cheating

- Discussing homework problems in class, with instructor, or with TA.
- Discussing homework *questions* with classmates (i.e. what do they mean?) — but doing the work on your own.

Why is cheating wrong?

- Grader cannot assess your performance.
- Penalizes those who do not cheat.
- Your unrecognized incompetence could cost millions of dollars or even kill people.
- HOWEVER:
- We are not looking for an excuse to fail you.

Contact Time

- Class:
  - TR 10:05AM to 11:30AM
  - Science Library 210

- Office hours:
  - TR 1PM to 2PM and By appointment
  - Engineering (EB) Q01

- Email: millerti@binghamton.edu
  - Short questions, arrange appointment

Part I: Basic Concepts

Outline

- Costs, economics, markets
- Architecture basics
  - Machine cycle
  - Instruction Set Architecture (ISA)
- Processor design & implementation
- Fabrication
- Circuits, technology trends, power
- Performance
  - RISC vs. CISC
  - Locality of reference
- Pipelining
- The "APEX" archetypical CPU
- Advanced Pipelining

Costs, Economics, Markets

Economics of Processor Design

- Cost of CPU =
  - High-level design (this class)
  - ***Testing***
  - Gate/transistor level implementation
  - ***Testing***
  - Masks, fabrication, packaging
  - ***Testing***
  - Business overhead & profit margin

Cost to design new high-end CPU

- ~ 600 man years (150 designers for 4 years)
- At $300K per person per year:
  - $180 million

CPU cost drivers beyond basic design

- Functional testing
- VLSI/Chip-level testing

- These increase with chip complexity

Economy of scale

- 10K units $18K / unit
- 1M units $180 / unit

- Requires high volumes

Market segments

- PCs are largest market in $ sales
- Embedded are largest in units sold
- Workstations, high-end
- Servers

Server processor market, 2006-2011

- 22 million server processor chips shipped
- 60% were x86-based
- $500 to $1000 retail per microprocessor

Desktop & Workstation markets

- Commodity
- Highly competitive
  - Features: multi-core, memory latency, graphics, encryption, performance per watt
  - Price: $200 in 2000, $150 in 2005, $105 in 2010
- High-end chips, well over $800 / chip

Cell Phones & Tablets

- Very competitive with many vendors and many sophisticated designs
- Most important: Performance, features, battery life
- Features: Graphics, video, encryption, speech recognition
- Evolution: Discrete chips toward highly-integrated multi-core at 1+ GHz clock
- Cost: Order of $1 to $10 per SoC

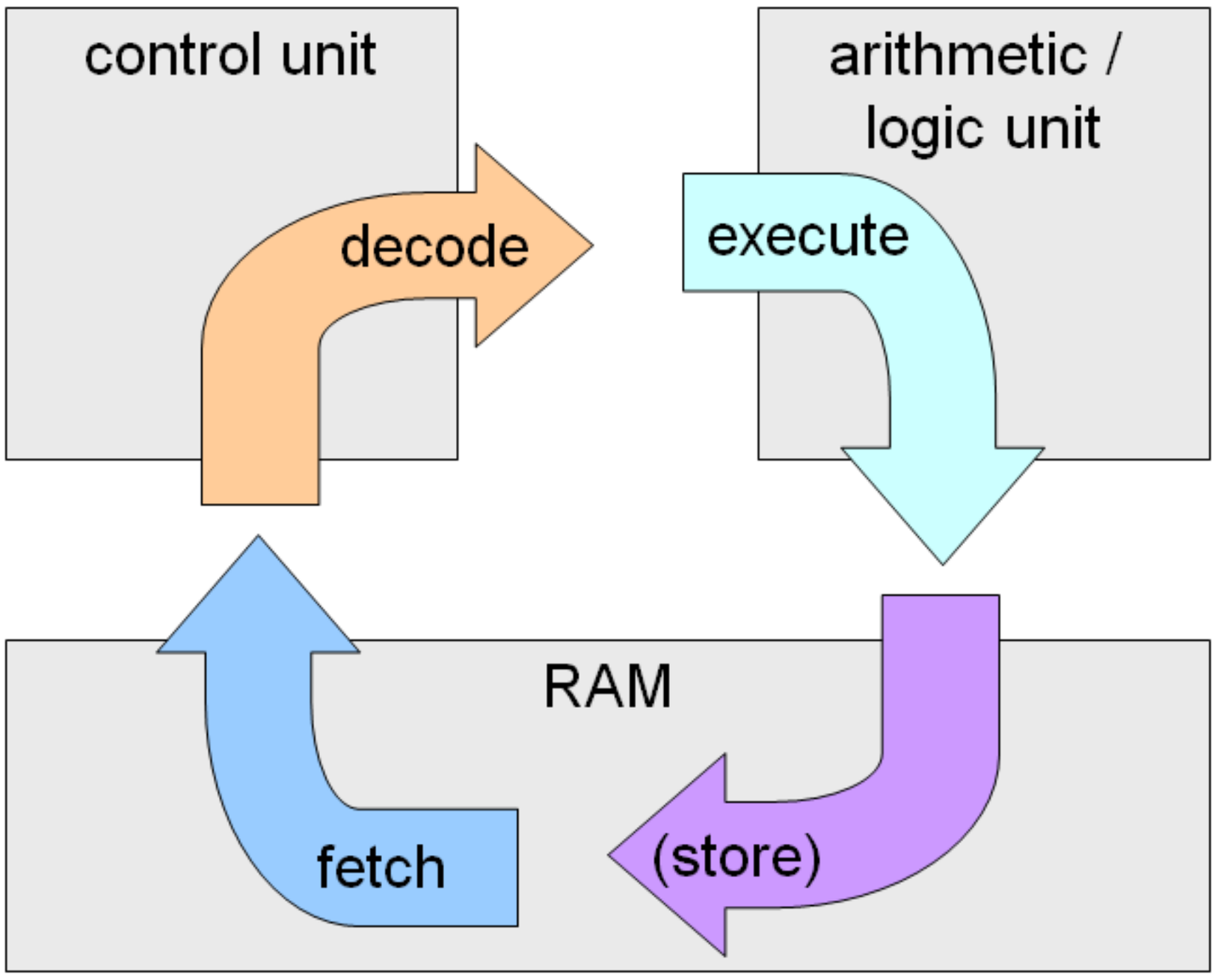
Intel x86 server/ desktop market

- Revenue: $26.5 B in 2001 to $43.6 B in 2010
- Breakdown for 2010: 60% PC processors, 17% servers, 18% chipsets
- Intel owns 80% of PC CPU market, AMD owns most of the rest

Architecture Basics

http://blog.shamess.info/2008/12/03/computer-tech-lecture-three/

Basic CPU Machine Cycle



Instruction Set Architecture (ISA)

- Functional behavior of processor
- Characterizes processor family
- Can have multiple implementations
- Produced by compiler
- Implemented by ***microarchitecture***

Characteristics of an ISA

- Specification provides:
  - Instruction types and semantics
  - Instruction formats
  - Addressing modes
  - Data formats (word sides, endianness)

Examples of ISAs

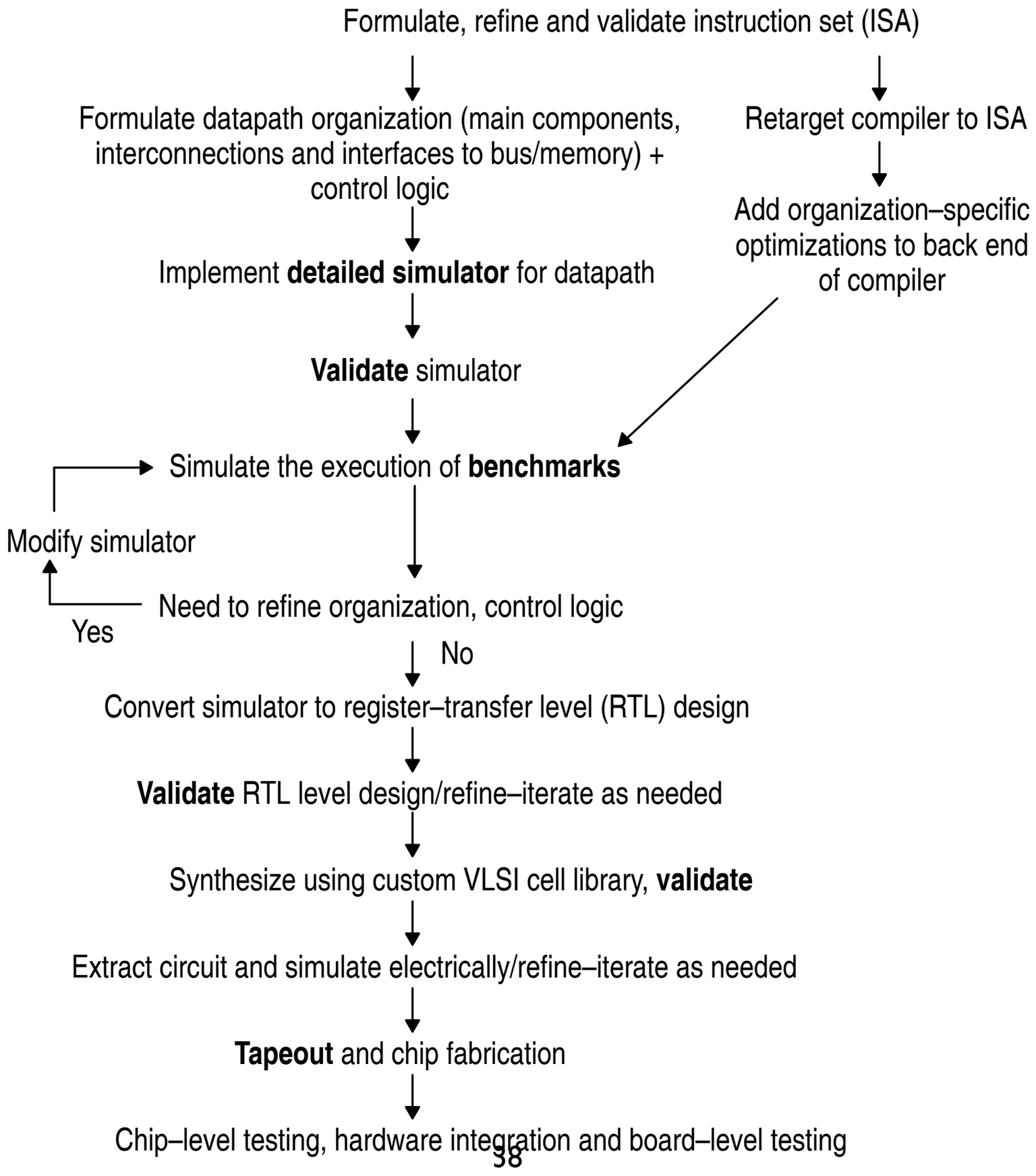
- x86 Intel, AMD, Via, Transmeta
- x64 Intel, AMD
- IA-64 Intel, HP
- SPARC Sun, Fujitsu
- PowerPC IBM, Motorola
- MIPS SGI, NEC

New ISA is very expensive

- ISA design
- New hardware
- New compilers
- New software
- Must have adequate base of applications
- Would take 5+ years to recompile x86 software for PowerPC

Designing and Implementing a Processor

Steps Involved in Designing and Implementing a Modern Processor



Typical Design & Implementation

- ISA already exists
- Design/validation team of 150 to over 200
- 2 validators for each designer
- Separate teams for interfaces, buses, chipsets
- Validate based on detailed simulation
- Discover bugs at every validation point

Typical Design & Implementation

- Bugs discovered after release, often not serious
- Trend towards using HDL like Verilog for simulator
- Simpler CPUs go through same steps

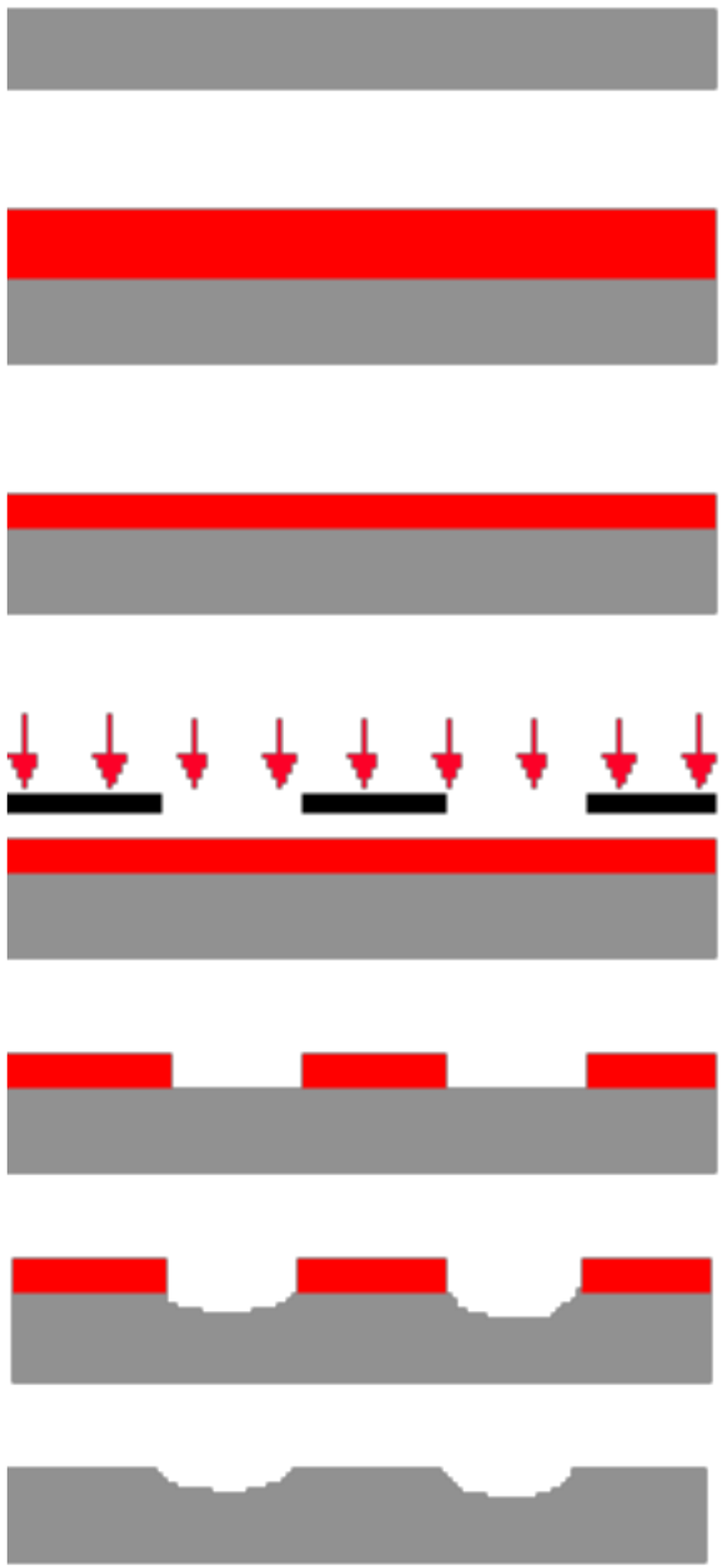
Fabrication

Contemporary VLSI Technology

- Virtually all CPUs implemented completely within a single chip
- Circuit components: Transistors, wires, limited capacitors, limited resistors

Microchip fabrication process

- Cut extremely pure silicon wafer
- Lithographically fabricate many identical copies of same design on wafer
- Series of steps that form layers of various materials:
  - Etching & deposition

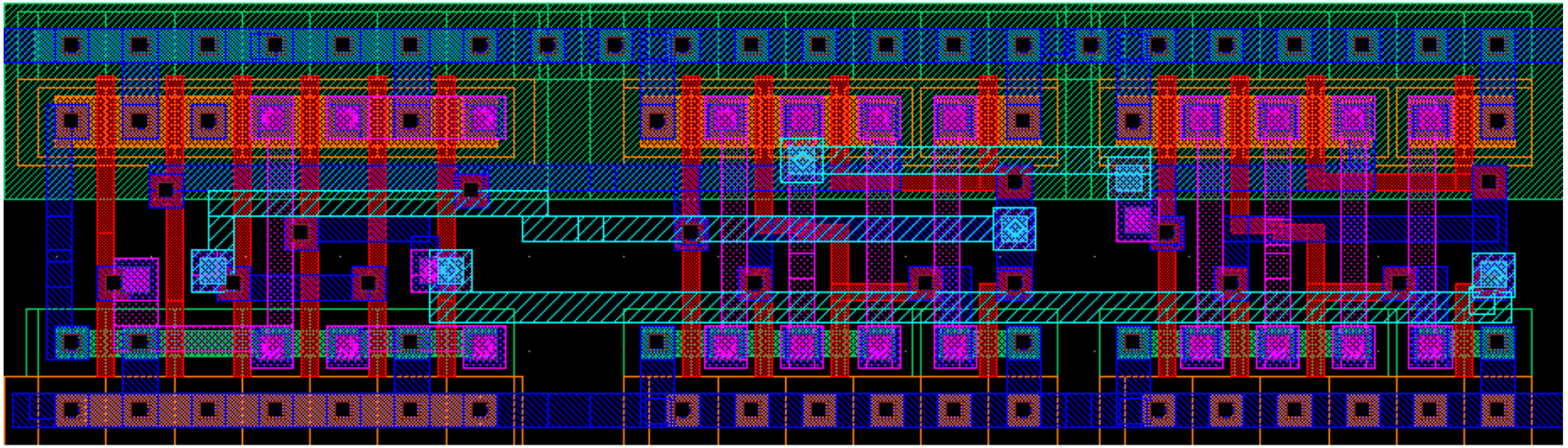


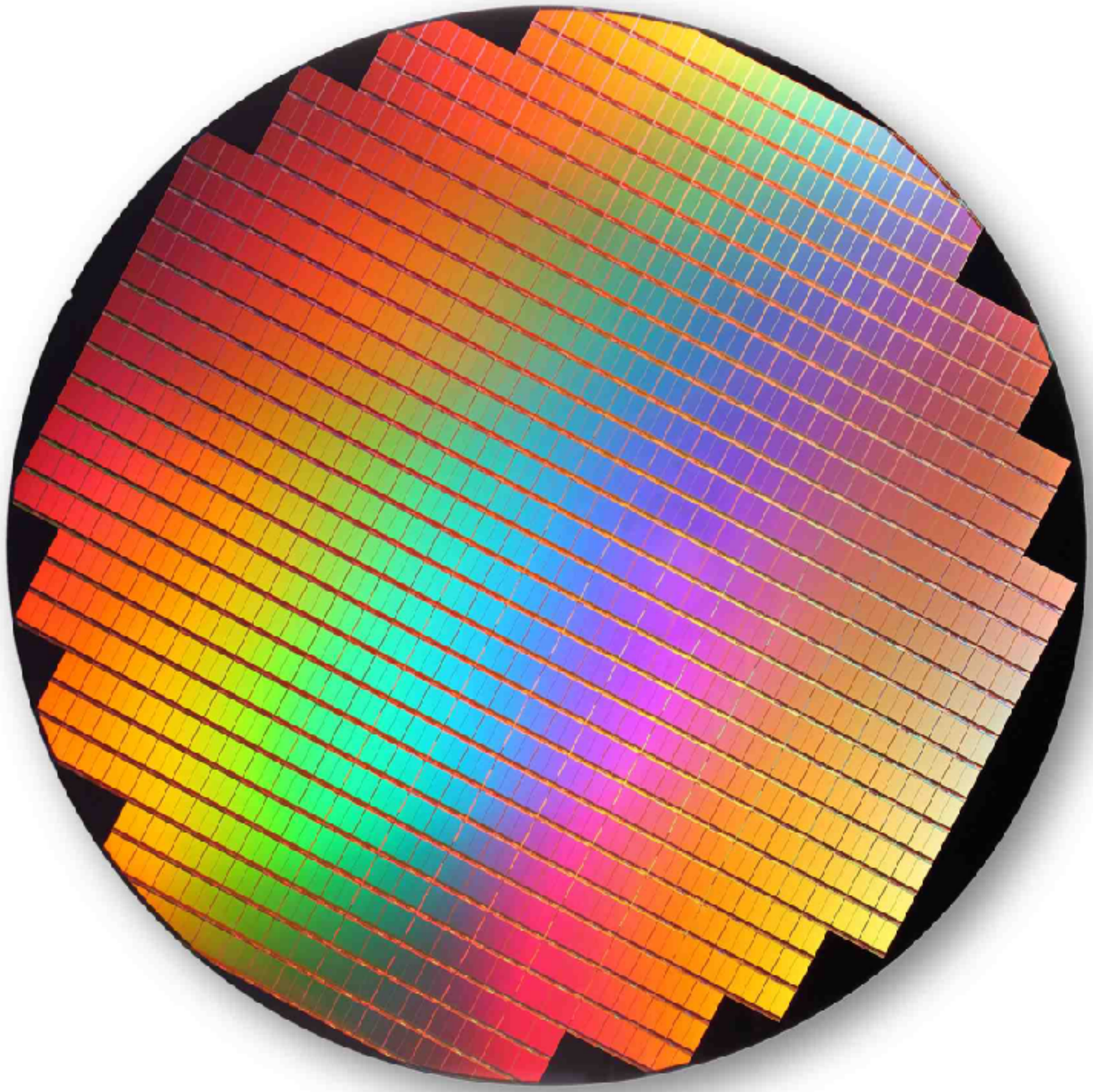
http://www.lithoguru.com/scientist/lithobasics.html

Types of microchip layers

- Semiconducting (doped silicon)
- Poor conductors (polysilicon)
- Insulators (SiO_2)
- Conductors / Interconnects (2 to 5 layers of metal, separated by insulators)

- Patterns specified by MASKs. Careful alignment required.
- 192 nm light, smaller features





Fabrication process

- Lithographically lay out layers of doping, insulators, and interconnects
- Cut dice from wafers
- Test & reject
- Package

Yield

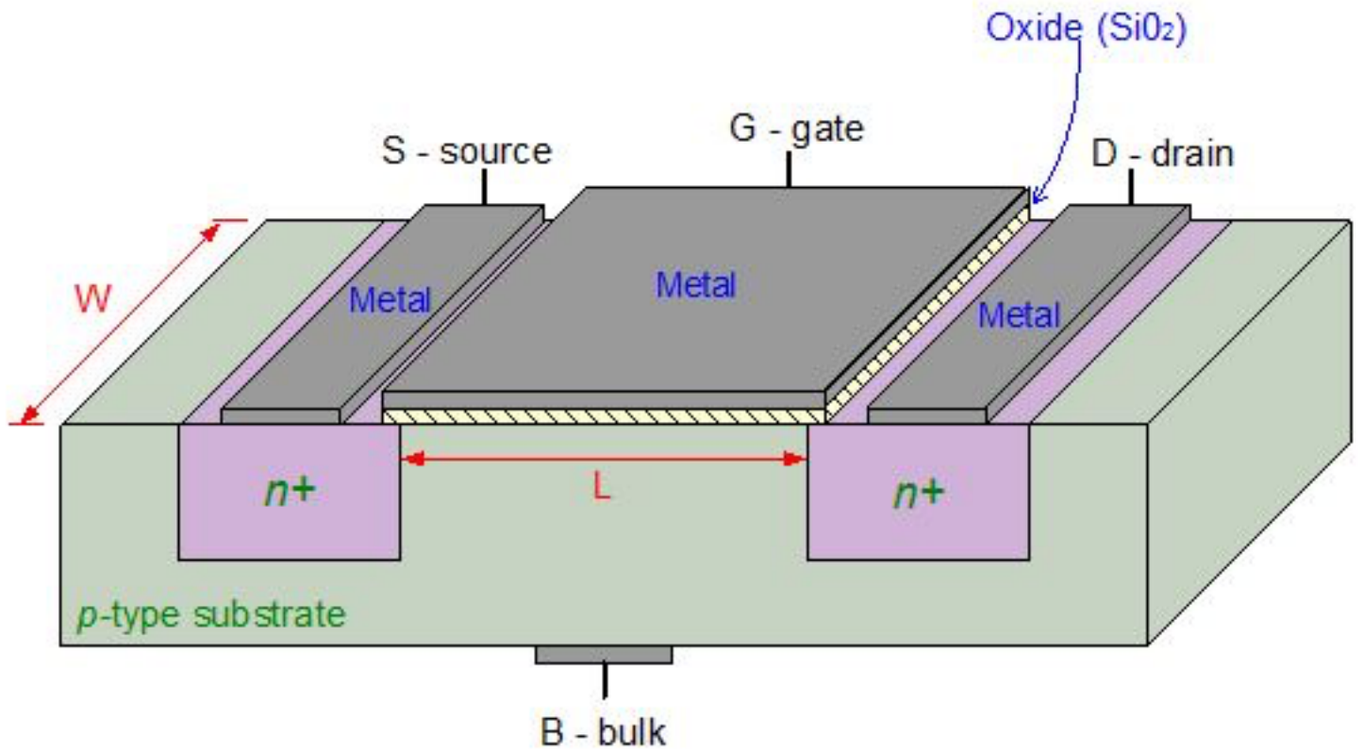
- Fraction of good dice to total
- Lower die area -- better yield
- More regularity, better mask alignment -- better yield

Circuitry, Tech Limitations, Power

Logic vs. DRAM

- CPU/logic/SRAM: Requires high speed, many layers of metal
- DRAM: Requires high density, few layers of metal, much cheaper

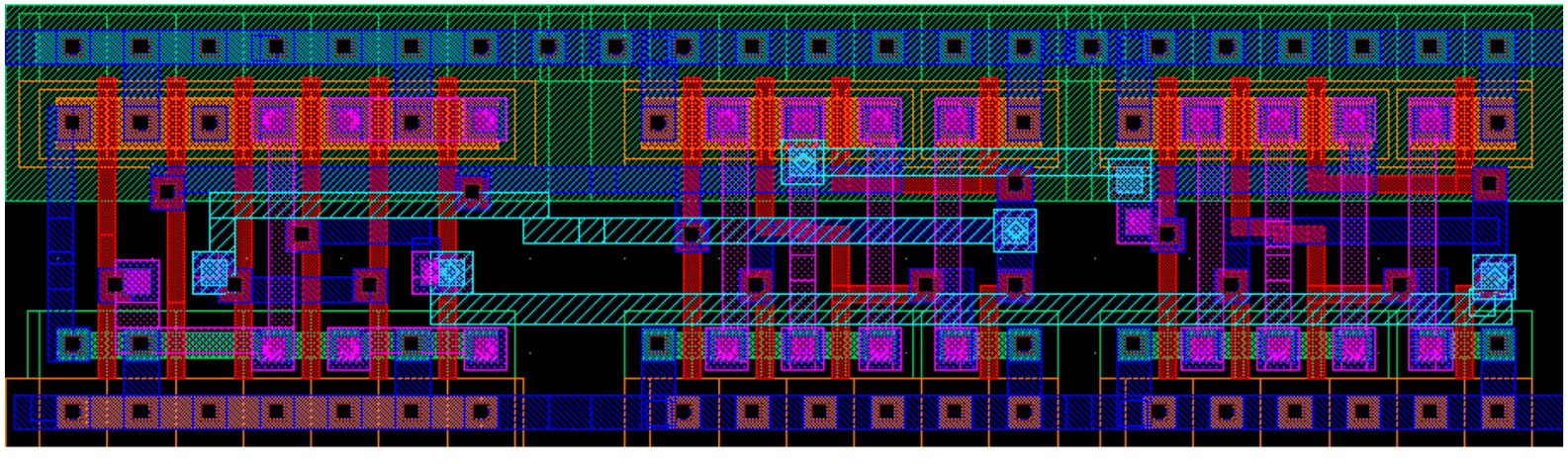
Circuit area & speed

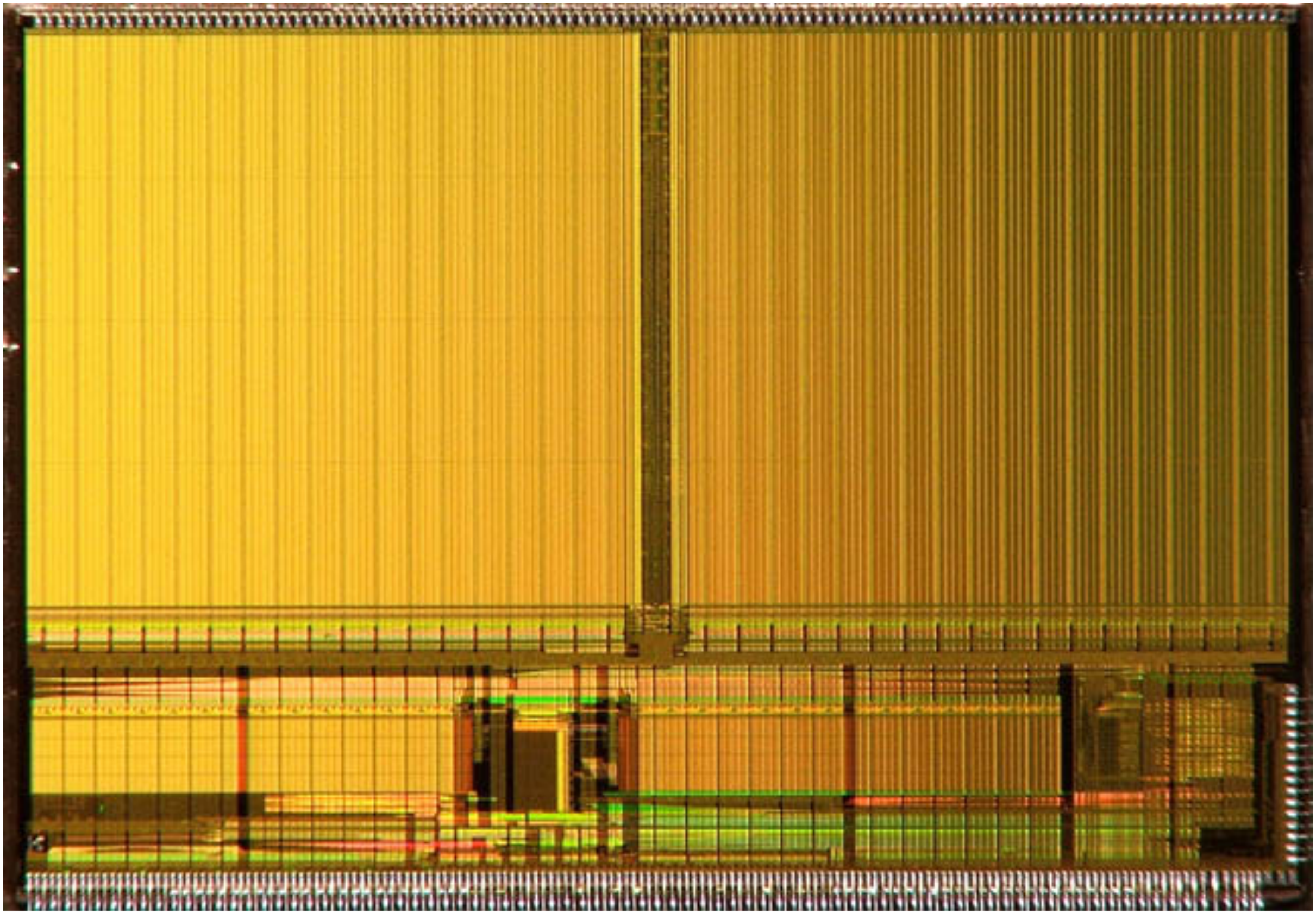


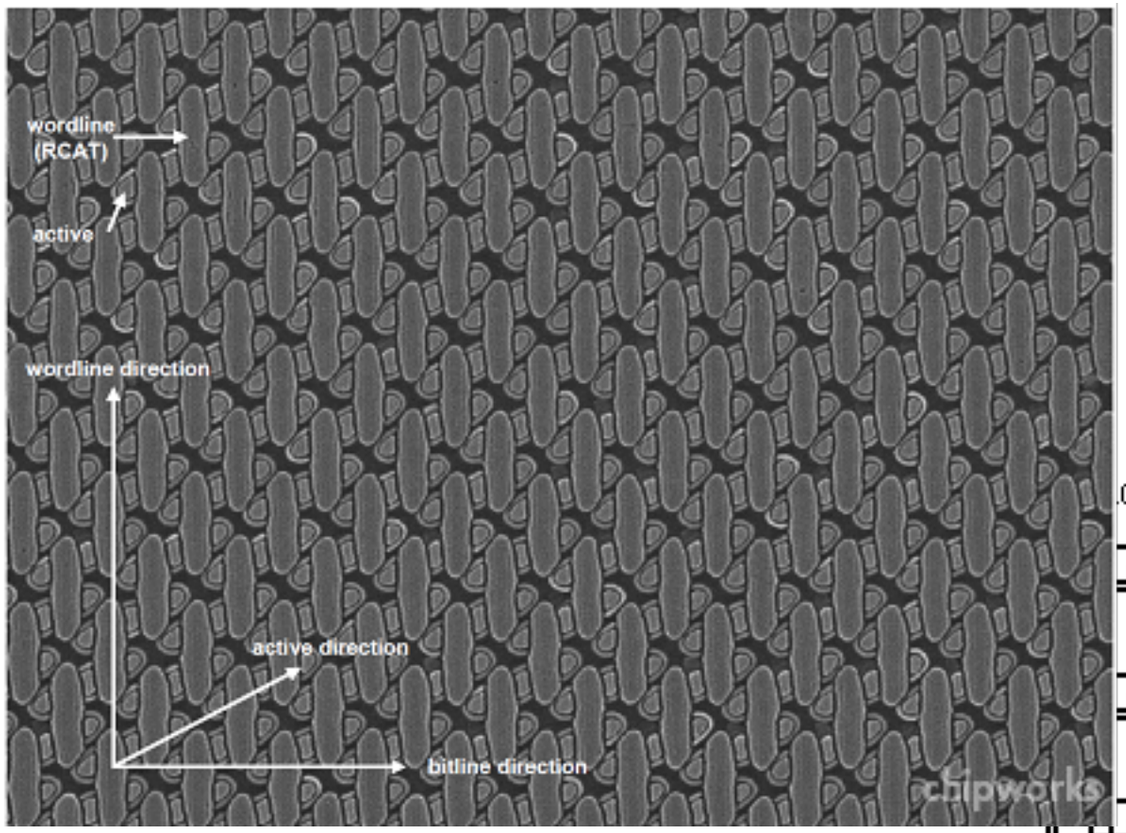
- Smaller transistors (channel length) are faster but use more energy
- Smaller circuits are faster, shorter wires
- Modern circuits dominated by wire delays

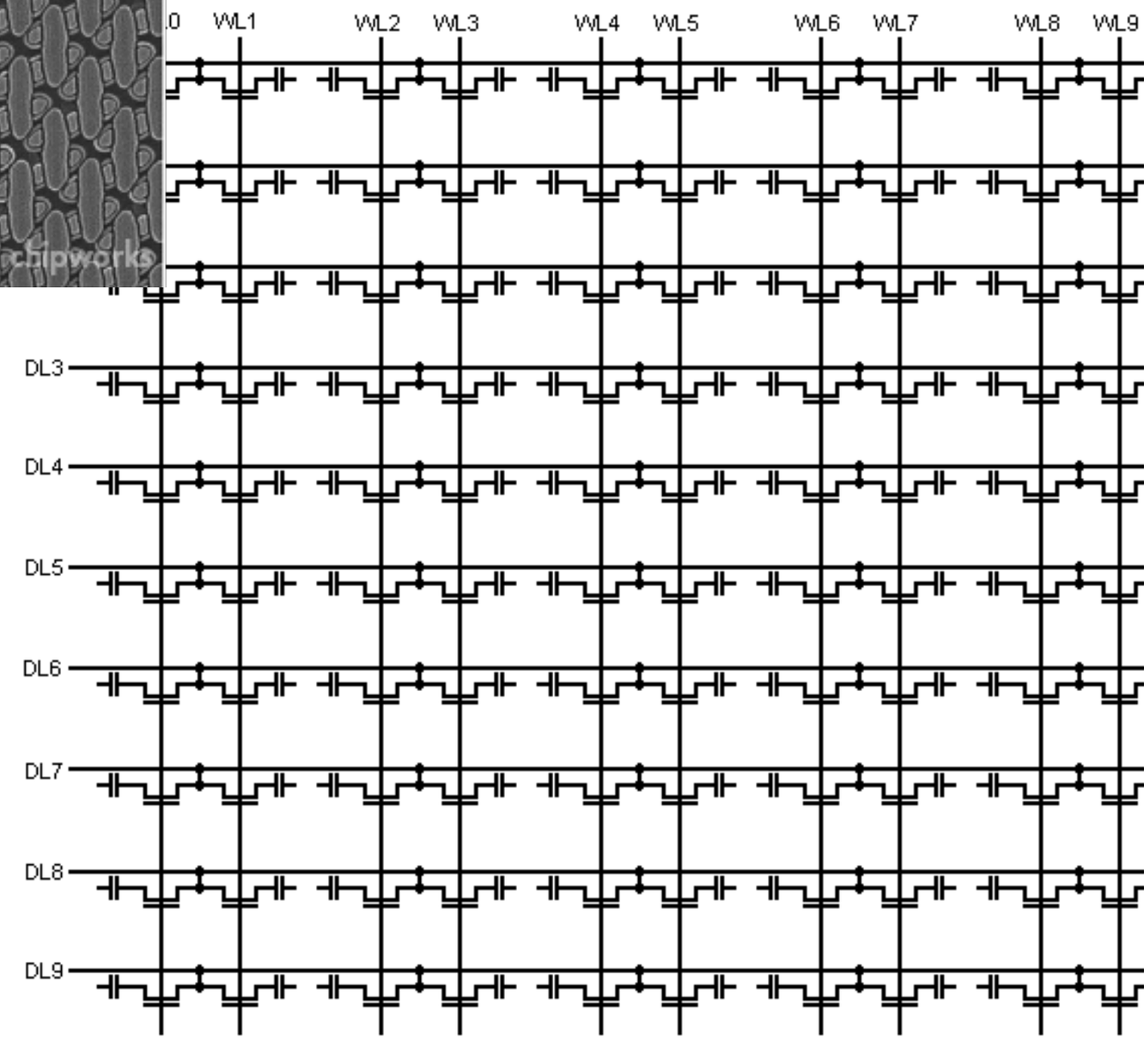
Technological Constraints

- Design irregularities in logic limit maximum number of transistors per chip
- SRAMs and FPGAs more dense due to regularity
- DRAMs smallest at 1.5 B transistors per die






wordline
(RCAT)
active
wordline direction
active direction
bitline direction
chipworks


.0
WL1
WL2
WL3
WL4
WL5
WL6
WL7
WL8
WL9
DL3
DL4
DL5
DL6
DL7
DL8
DL9

Technological Constraints

- Off-chip I/O delay 10x to 50x on-chip
- Limited I/Os: Pin count max at around 2000; about half are pwr & ground.

External I/O limitations

- Confine most activity to within chip
- Exploit spatial and temporal LOR
- Multiplexing of I/Os for multiple purposes
- Burst transfers, streaming, high-speed differential serial
- Clock multipliers

Transistor count limitations

- Put only the most essential functions on a single chip.
- 1971 -- Simple 4-bit CPU, Intel 4004, first microprocessor
- Late 70's -- 16-bit general purpose CPU on single chip, no floating point, cache or virtual memory

Transistor count limitations

- Early 80's -- 32-bit CPU, no cache or floating point. Some microcontrollers included limited RAM + communications. Some pipelined designs.
- Late 80's, early 90's -- 32-bit and 64-bit CPUs, on-chip floating point, single level of cache. Early superscalar.

Transistor count limitations

- 1993 -- DEC 21164: 64-bit, on-chip floating point, two level cache, superscalar
- Late 90's -- multiprocessors, high-speed cluster networking, MCMs, VLIW, etc.
- 2000's -- SMT, multi-core, on-chip media processors, on-chip graphics, etc.
- Future -- Greater integration of CPU, RAM, chipset. Single-chip multiprocessors. Power-aware designs.

Recent limitations

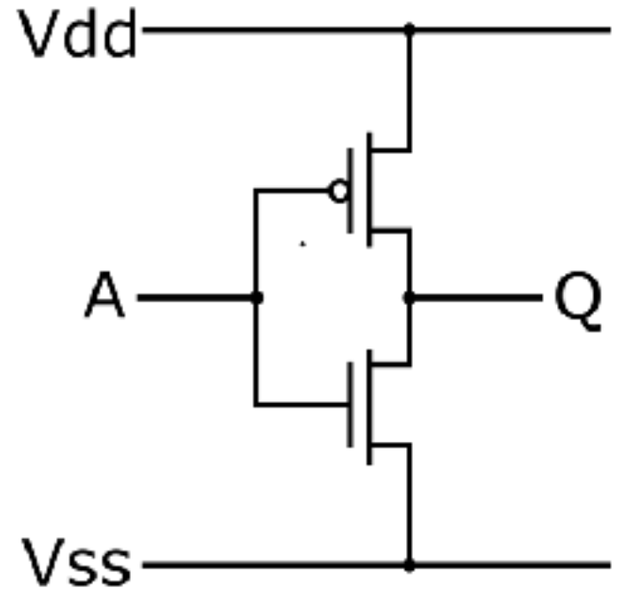
- Power wall -- performance limited by maximum heat dissipation
- Energy (charge) wall -- limited by battery life
- Dominance of interconnect delays, clock skew

Peak power dissipation

Processor	Peak Power (W)	Clock rate (MHz)
Alpha 21264C	95	1001
AMD Athlon XP	67	1800
AMD Athlon 64 3800+	89	2400
IBM Power 4	135	1300
Intel Pentium 4 (dual MT)	115	3600
Intel Itanium 2	130	1000
Intel Xeon 5680 (6–core)	130	3333
AMD FX–8170 Bulldozer (8–core, Sept 2011)	125+	3900

Power components

- Dynamic/switching power -- Energy is used every time a CMOS logic gate switches.
  - $P = \alpha f V^2$
- Static power -- Subthreshold leakage current
  - Increases as transistors shrink, increases with temperature, linear with voltage



Problems with high power dissipation

- Decreased processor lifetime
- Hotspots
- Limits performance
- Cooling costs

Trends in Fabrication and Processor Design

CPU/Logic/SRAM trends

- Logic transistor counts increase 60% to 80% per year
  - Moore's law -- transistor count doubles every 18 months
- Switching speeds double every 3 years
  - Trend has slowed since 2001

DRAM trends

- Densities increase 60% per year (4x in 3 years)
- Access times have increased by only about 30% in the last 10 years.

External buses on PCB

- Speed trends: 25MHz in 1990's to 1GHz+ in 2000's, DDR.
- 500Mbits/sec on parallel bus in late 1990's to 4 Gbits/sec in 2000's using point-to-point differential signaling.

Recent Trends high-end processors

- Multithreading (SMT)
- Virtualization
- Adaptations to power wall
  - Throttling, more simpler cores, slower clocks, clock/power gating
- Multi-core (CMP, SMP)

Recent Trends
Embedded processors

- Integration
  - Memory, DSPs, programmable logic
  - Multi-core
  - Array processors (video, signal processing, network, graphics)
  - Micro LMS3S101: $1 at 10K unit pricing
  - Dark silicon

Performance Considerations

Performance factors

$$T_{exec} = N \times \tau \times CPI$$

- N = Instructions executed
- τ = Clock period
- CPI = Average clocks per instruction

Reduced Instruction Set Computer (RISC)

- Reduce Texec by
  - Reducing CPI
  - Reducing τ

Complex Instruction Set Computer (CISC)

- Reduce Texec by:
  - Reducing N

Characteristics of RISC ISA

- Load-store architecture
- Simple instruction semantics
- Few addressing modes
- Few instruction formats

Op times are fast

Quick fetch and decode

Consequences of RISC

- Simpler hardware
- Smaller logic delays -- faster clock

- N goes up

- Ensure that growth in N is balanced by reduction in CPI and τ.

Characteristics of CISC CPU

- ALU instructions can reference memory
- High-level semantics
- Many addressing modes
- Non-uniform instruction formats

Op times are slow

Large decode times

Large fetch times -- memory alignment

Consequences of CISC

- Complex hardware
- Larger logic delays
- Higher CPI

- Philosophy: Amortize cost of fetch & decode by performing more work per instruction
- Challenge: Ensure increase in CPI and τ is compensated by drop in N

Locality of Reference (LOR)

- Temporal LOR: Repeated access of same address
- Spatial LOR: Likely access of nearby addresses
- Basis for most performance optimization techniques

Examples of LOR

Mechanism	*Type of LOR Exploited*
Registers	Temporal
Ins. Cache	Temporal & Spatial
Data Cache	Temporal & Spatial
Pipelining	Spatial
Interleaved mem	Spatial

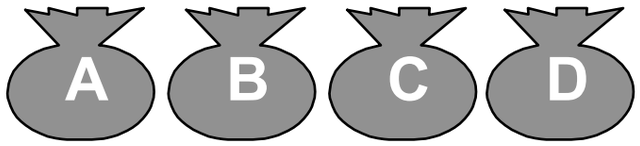
Pipelining and the "APEX" processor

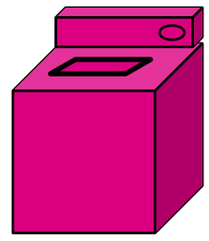
Pipelining

- Divide labor among multiple hardware functional units
- Route work through chain of units
- Analogy: Assembly line or fluid flow through pipeline
- Up to K consecutive tasks can be in various states of processing simultaneously
- Logic for each step is called a **stage**.

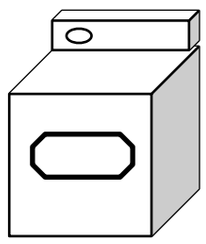
Pipelining is Natural!

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes

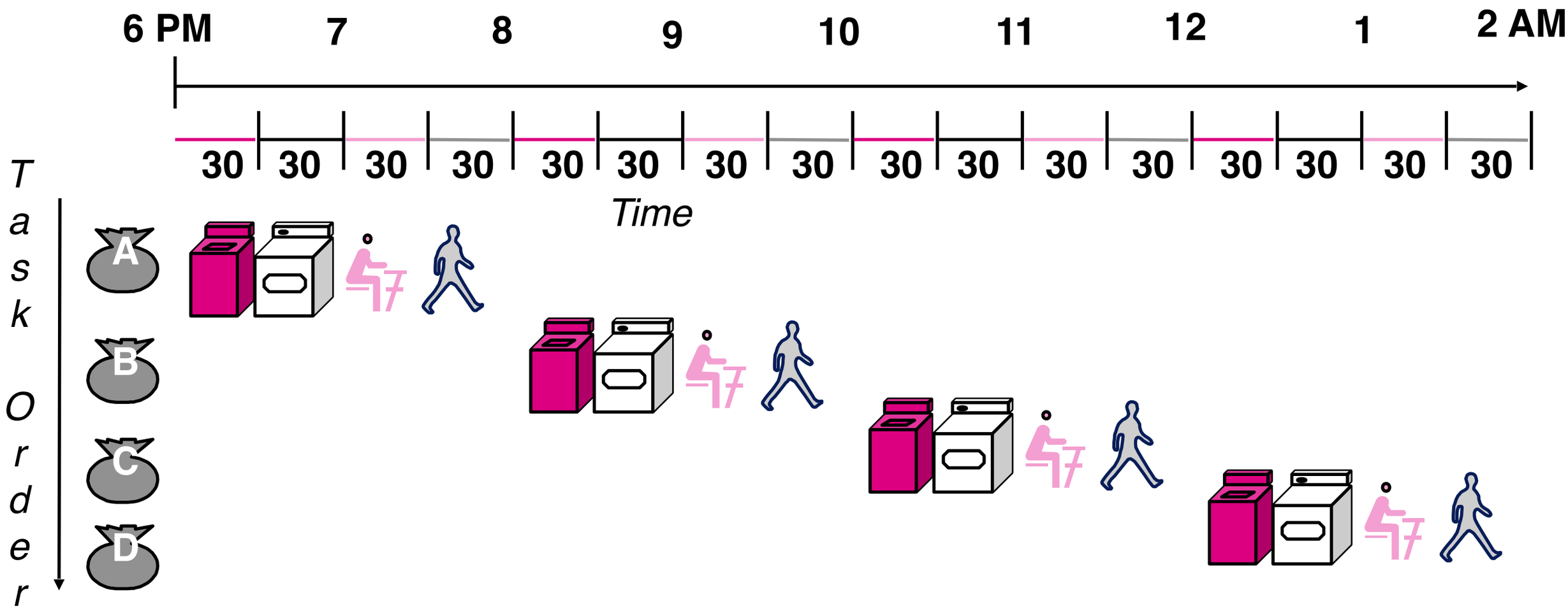




- Dryer takes 30 minutes
- "Folder" takes 30 minutes
- "Stasher" takes 30 minutes to put clothes into drawers



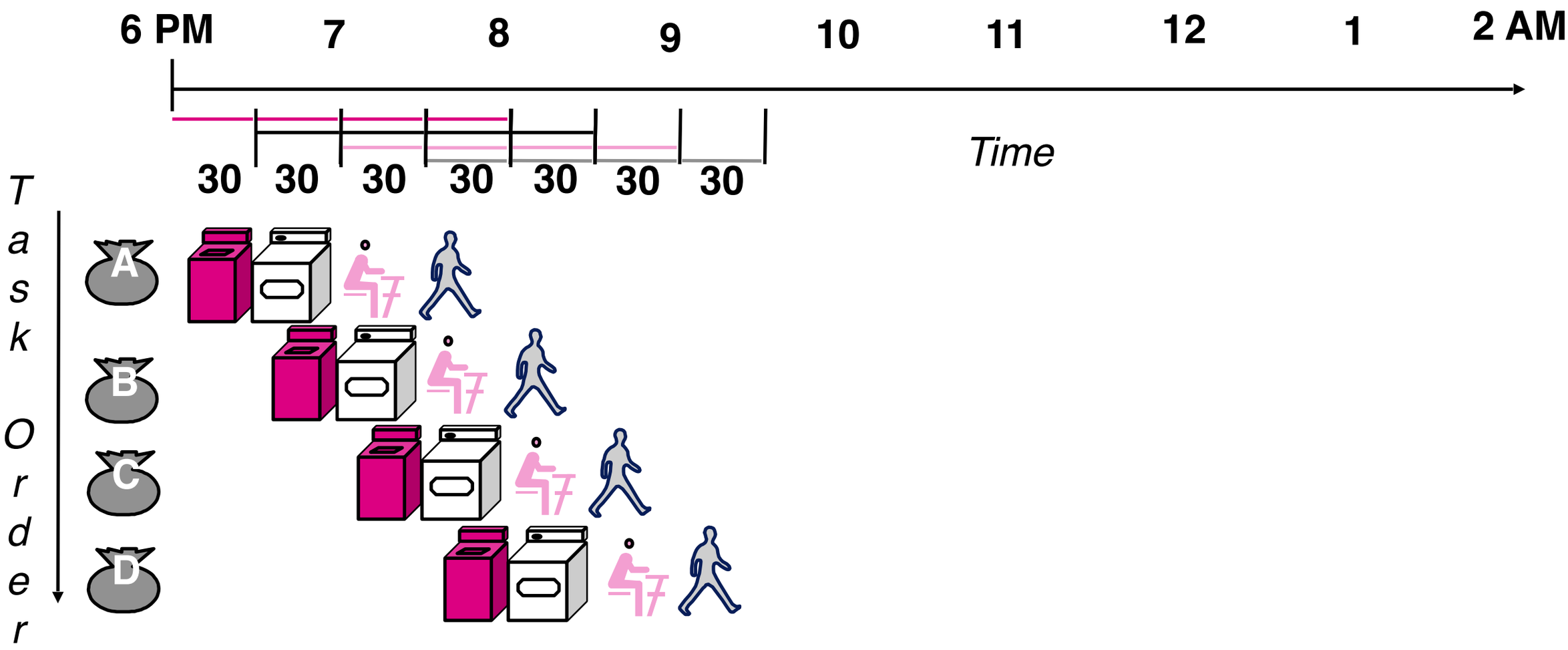
Sequential Laundry



° Sequential laundry takes 8 hours for 4 loads

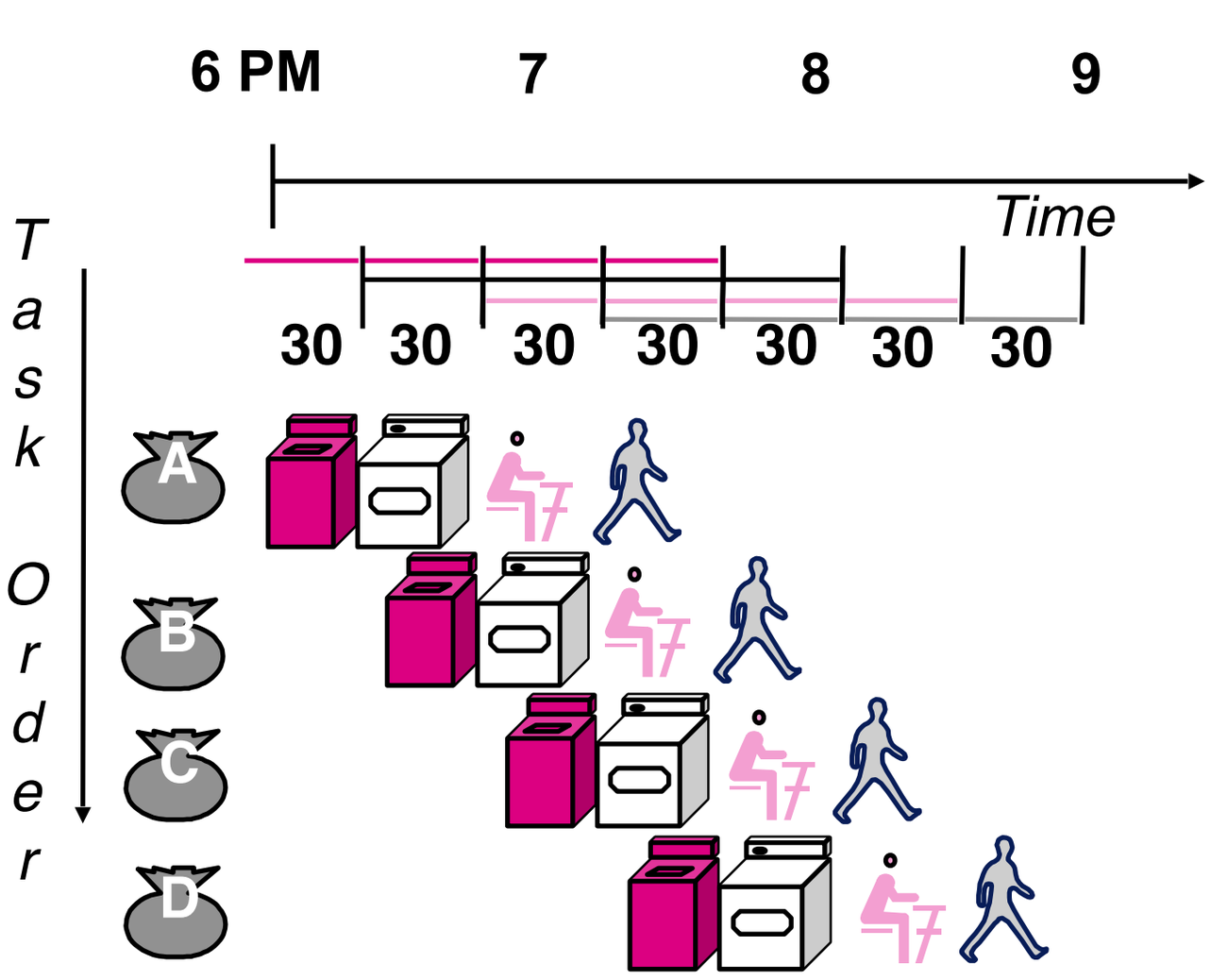
° If they learned pipelining, how long would laundry take?

Pipelined Laundry: Start work ASAP



° Pipelined laundry takes 3.5 hours for 4 loads!

Pipelining Lessons



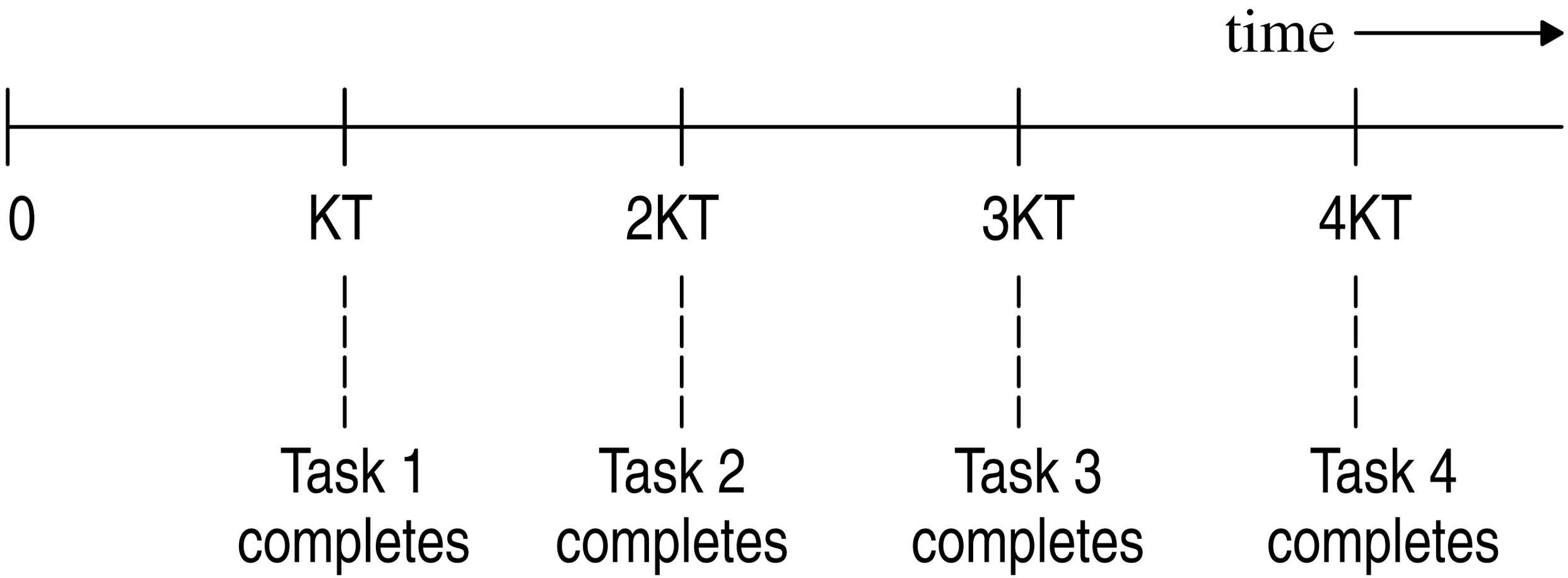
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Stall for Dependences

Improving throughput by Pipelining

- N processing tasks
- K steps per task
- T duration each step

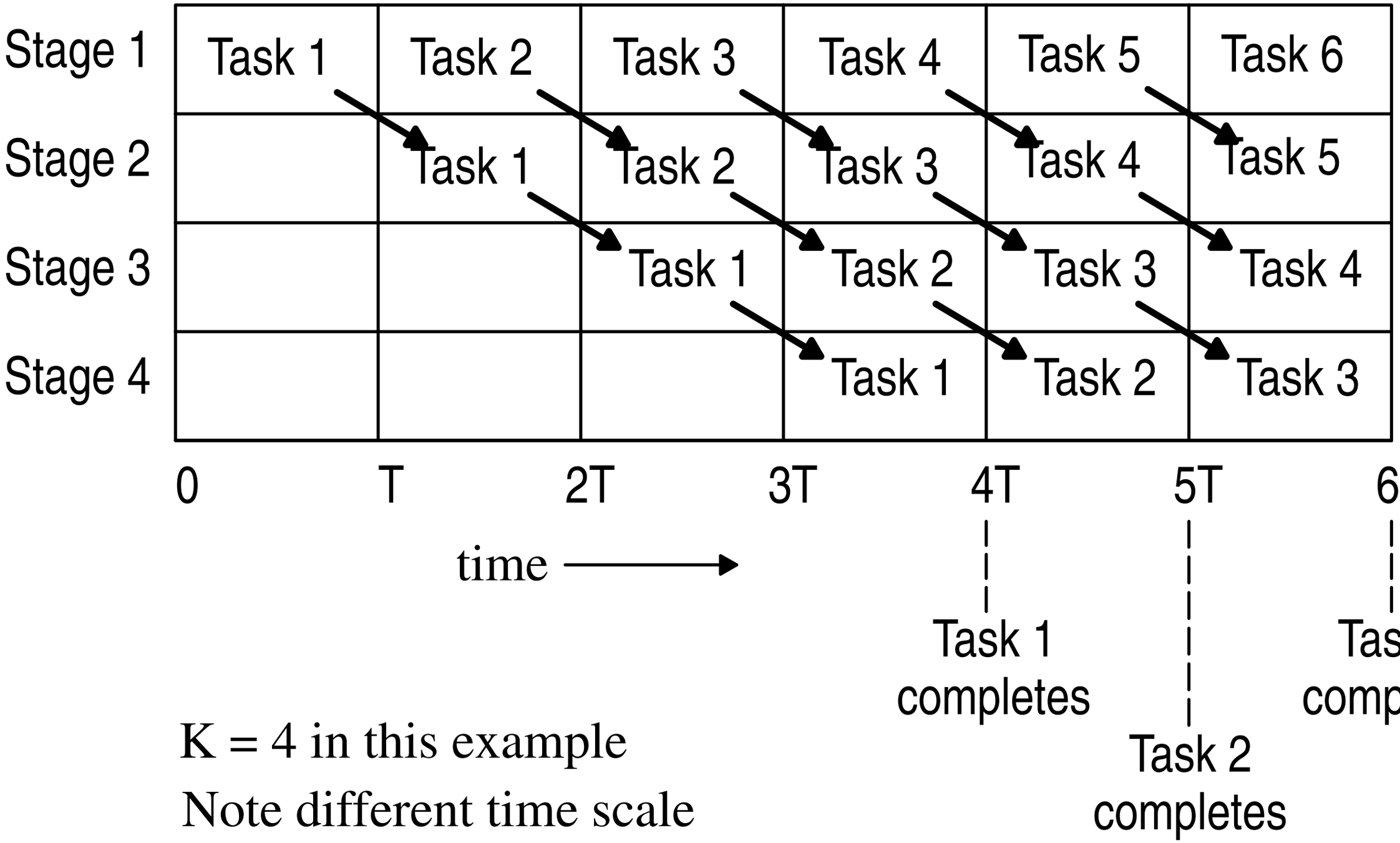
- Non-pipelined: N * (K * T)

$$T_{np} = N * (K * T)$$


time
0
KT
2KT
3KT
4KT
Task 1 completes
Task 2 completes
Task 3 completes
Task 4 completes

Without pipelining

- No overlap in processing of consecutive tasks
- Throughput = completion rate = one task every K*T time units
- Processing latency per task = time between start and end = K*T



K = 4 in this example

Note different time scale

Pipelining

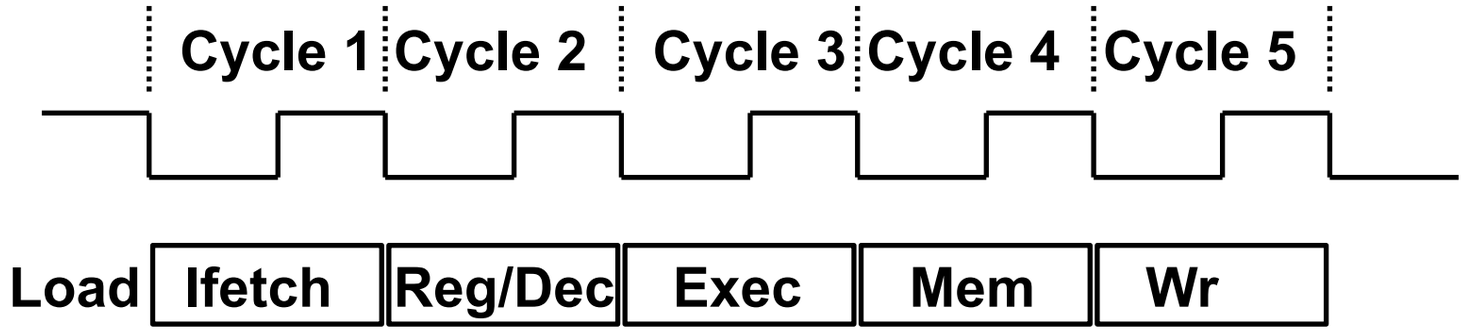
- Total processing time
  - $T_p = K * T + (N-1)*T$
- First task takes K*T, subsequent complete every T
- Throughput = one task per T time units, improved by a factor of K
- Latency = K*T (unchanged)

Instruction Pipelining

- Stages:
  - F: Instruction fetch
  - D/RF: Decode, register fetch
  - EX: Execute
  - MEM: Memory access
  - WB: Write-back

- Typical of load-store architecture

The Five Stages of Load



° Ifetch: Instruction Fetch
  • Fetch the instruction from the Instruction Memory

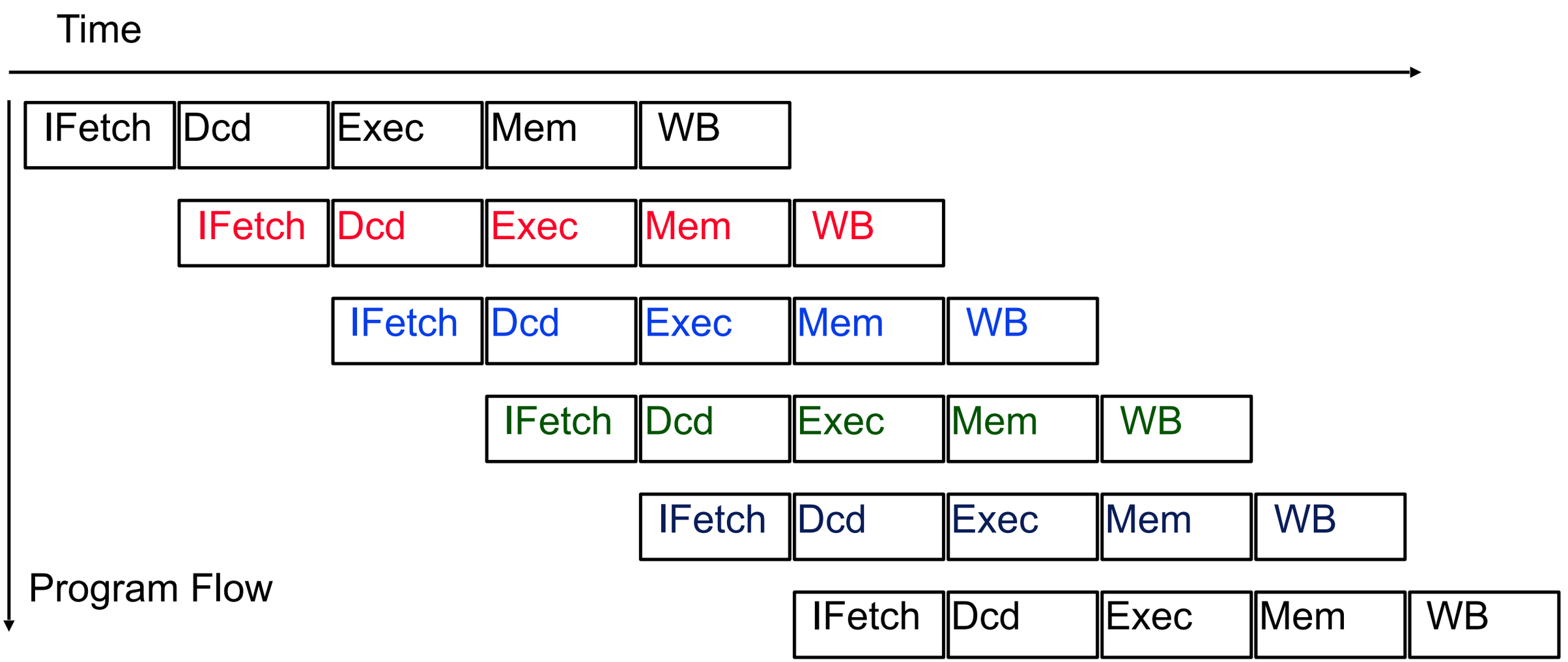
° Reg/Dec: Registers Fetch and Instruction Decode

° Exec: Calculate the memory address

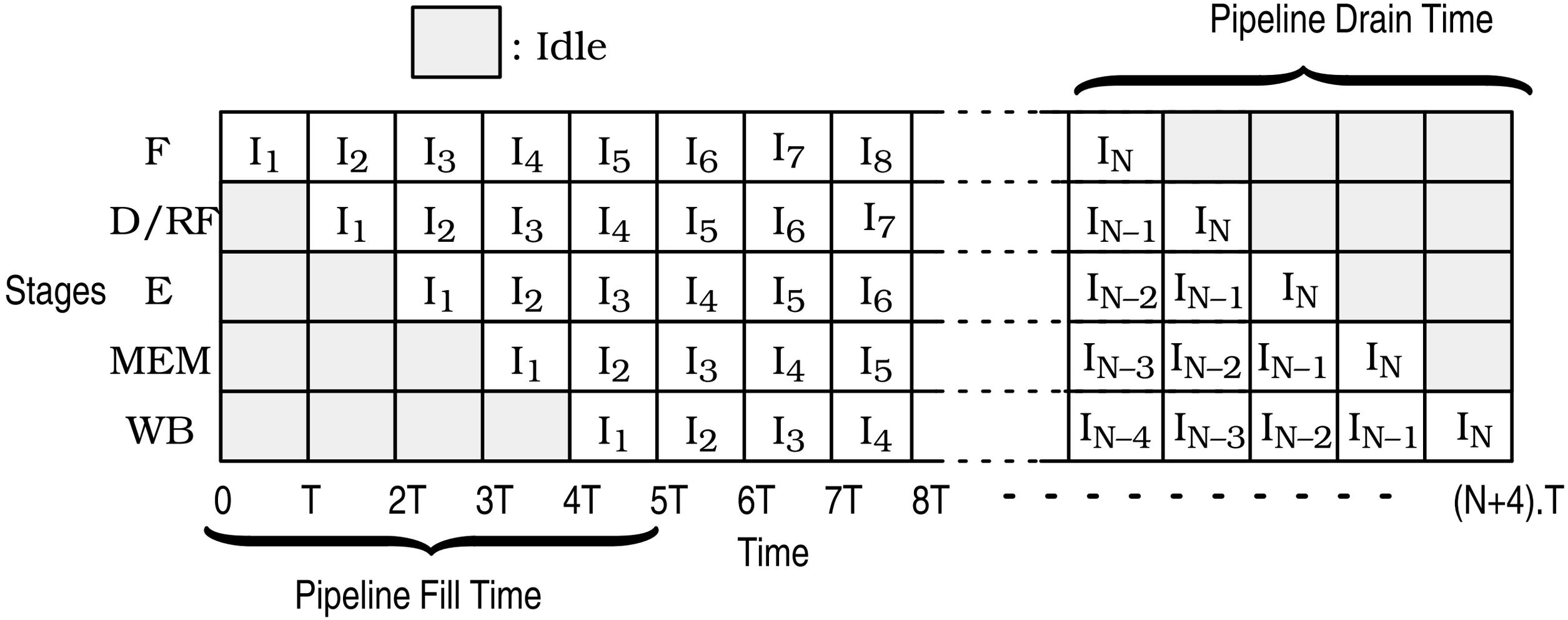
° Mem: Read the data from the Data Memory

° Wr: Write the data back to the register file

Conventional Pipelined Execution Representation

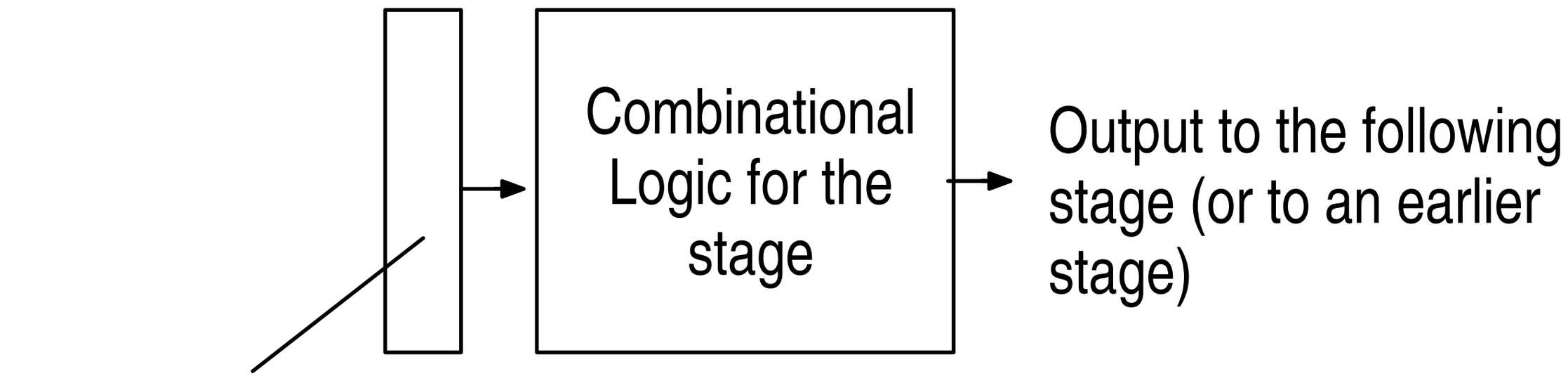


N instructions to be processed

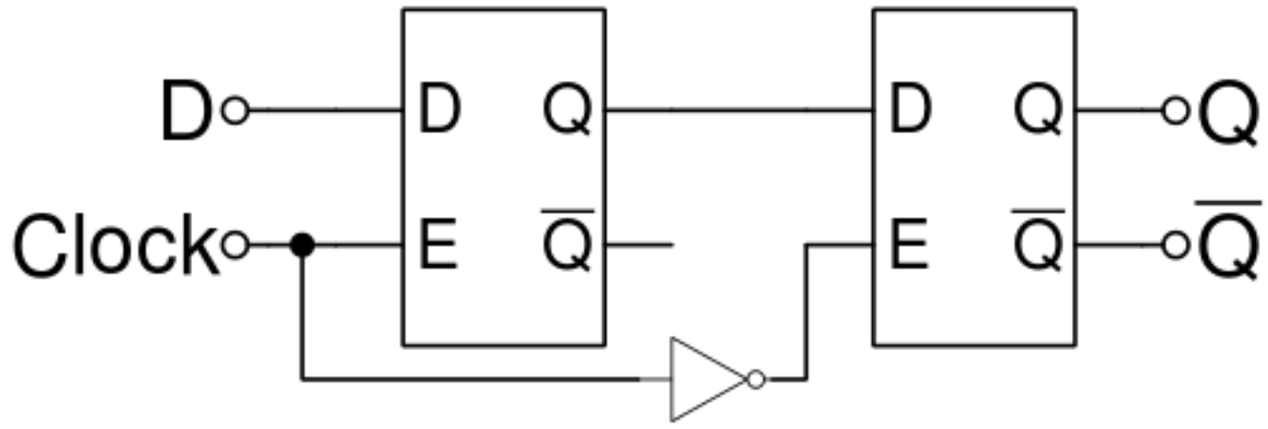


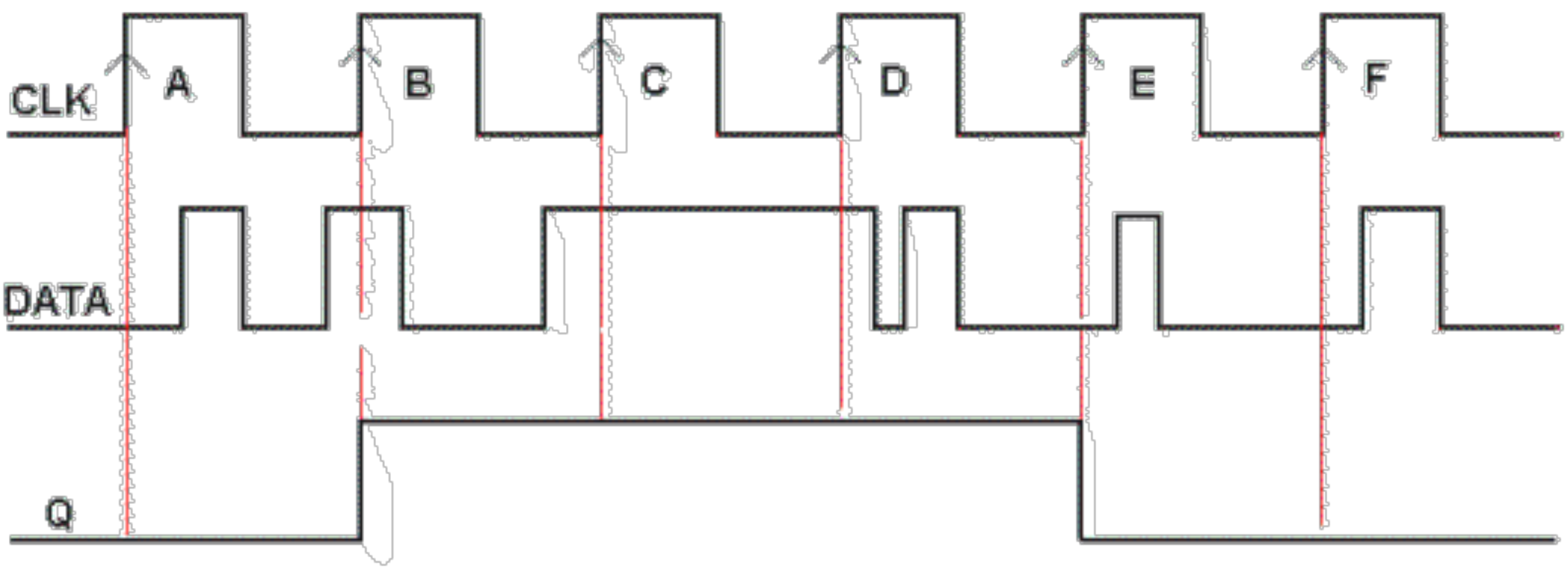
The Execution of a Sequence of N Instructions on APEX

Single stage of a pipeline



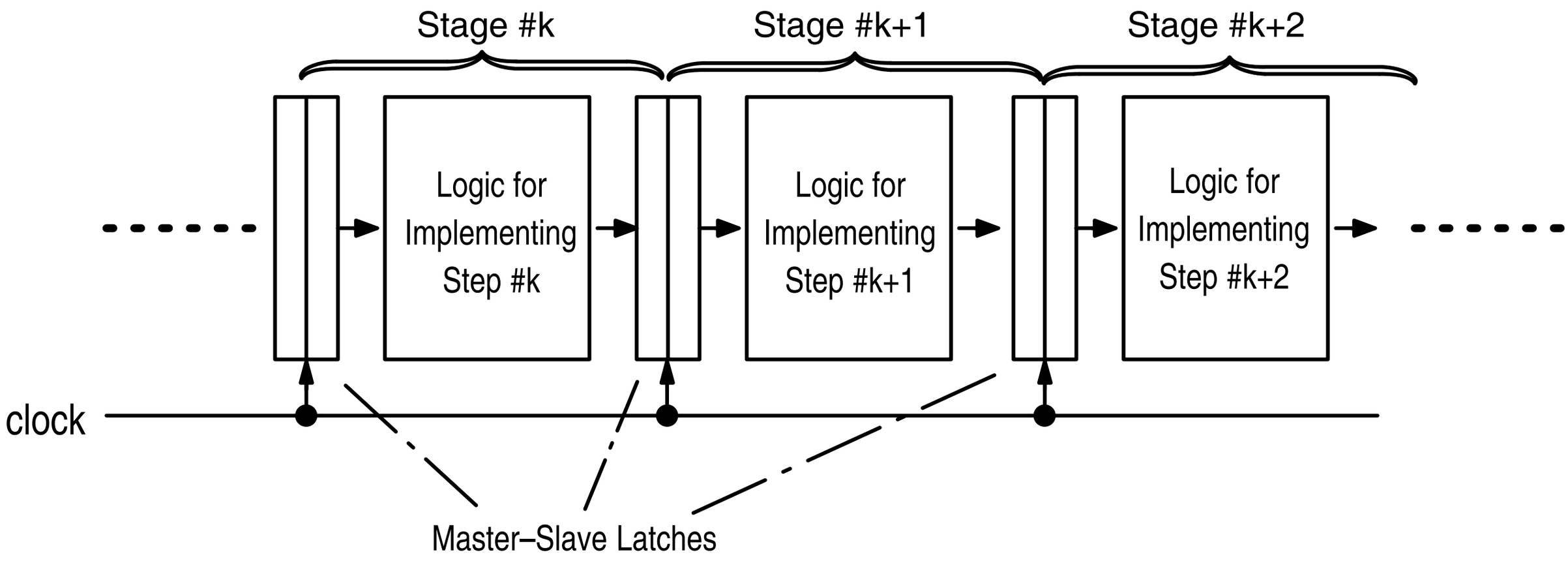
Edge Triggered Flip-flop



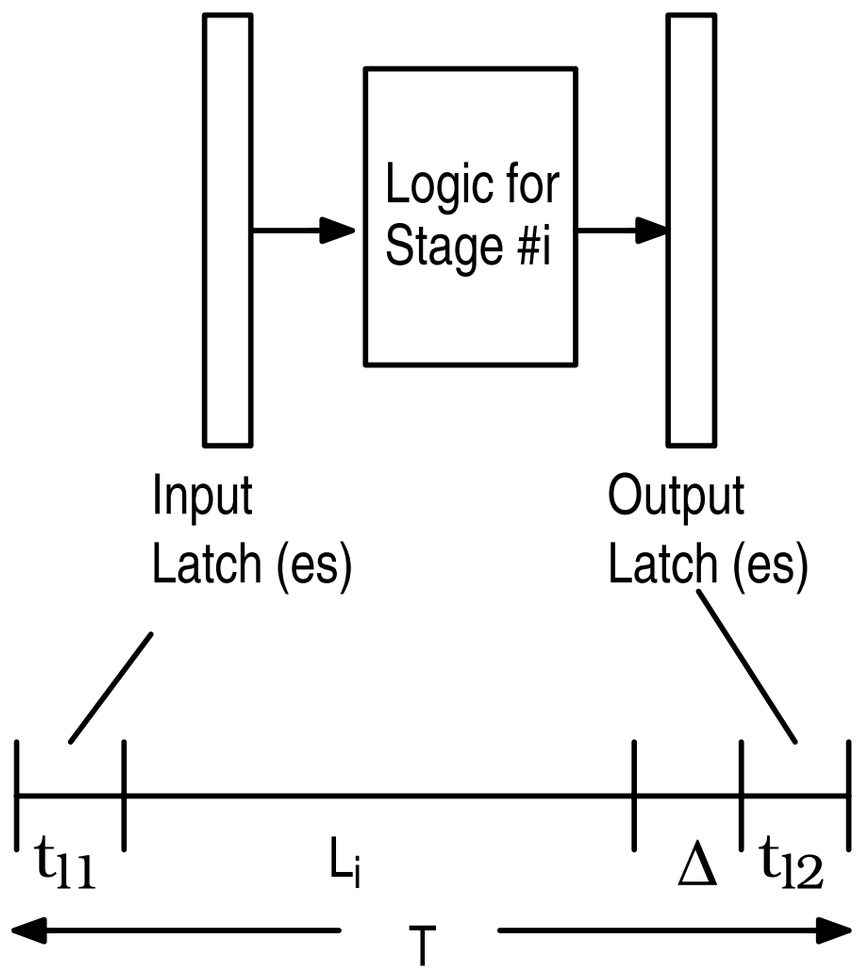


http://www.hobbyprojects.com/flip_flop/edge_triggered_flip-flops.html

Pipeline as a whole



Synchronous pipeline,
common clock



t_{l1} = input latching delay (delay of slave latch)

t_{l2} = output latching delay (delay of master latch)

$t_{latch} = t_{l1} + t_{l2}$ = overall latching delay: roughly the same for all stages

L_i = processing delay for the logic of stage #i

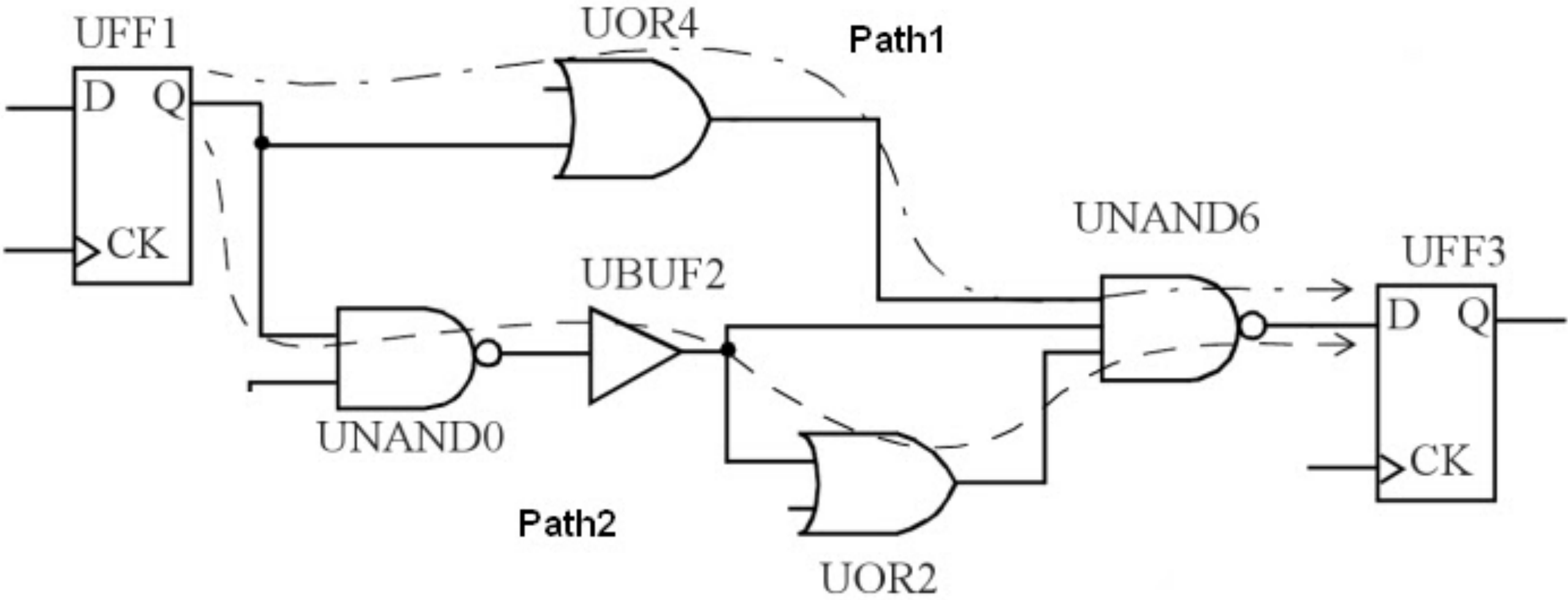
L_{max} = processing delay for the slowest logic associated with any stage = max $\{L_i\}$

T = pipeline cycle time
$= t_l + L_{max} + t_{skew}$

t_{skew} = allowance for clock skew

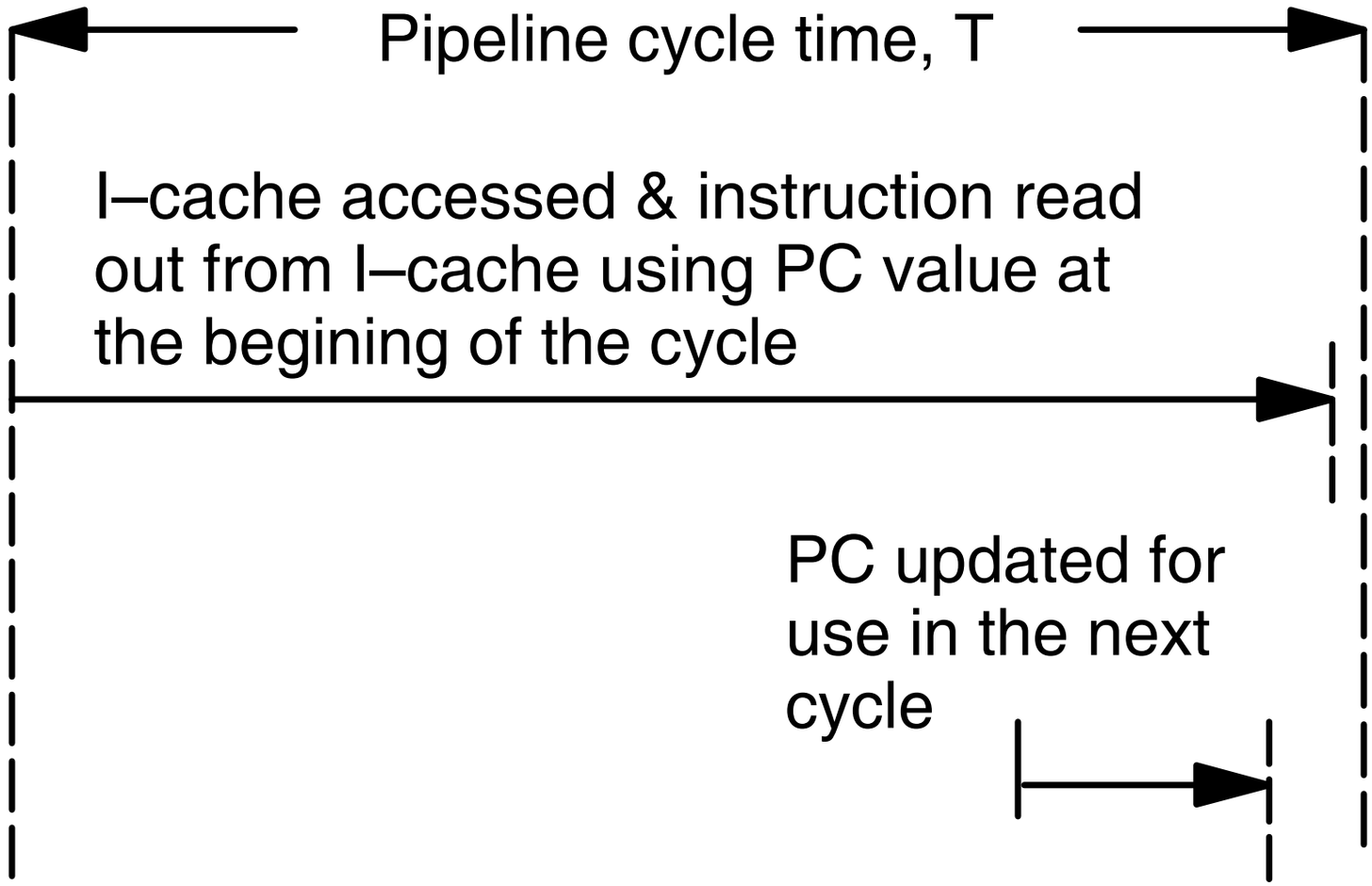
$\Delta = L_{max} - L_i + t_{skew}$

Circuit Path

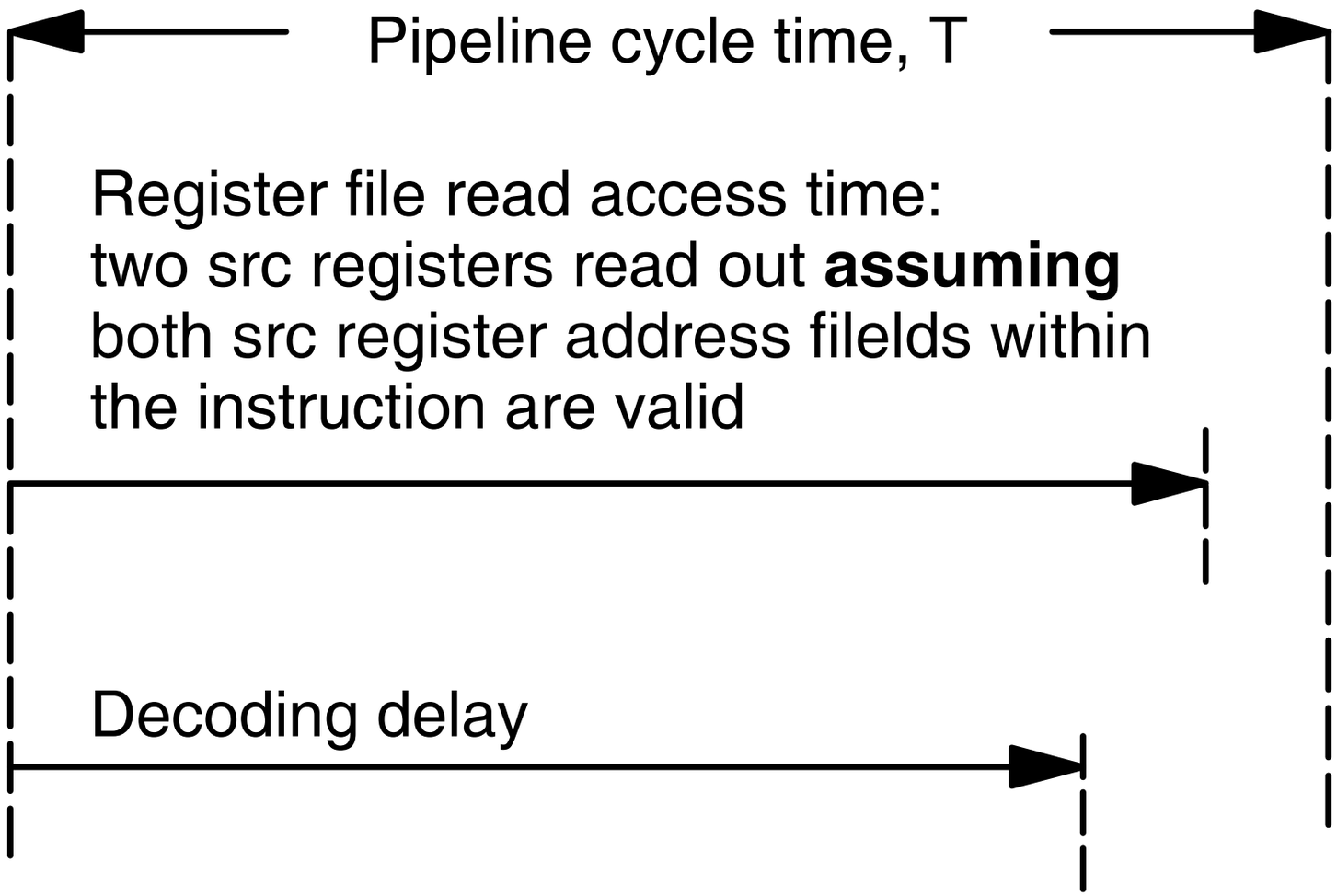


http://www.vlsi-expert.com/2012/09/maximum-clock-frequency-static-timing_30.html#.VA3NZ0s_1PY

Reading I-cache



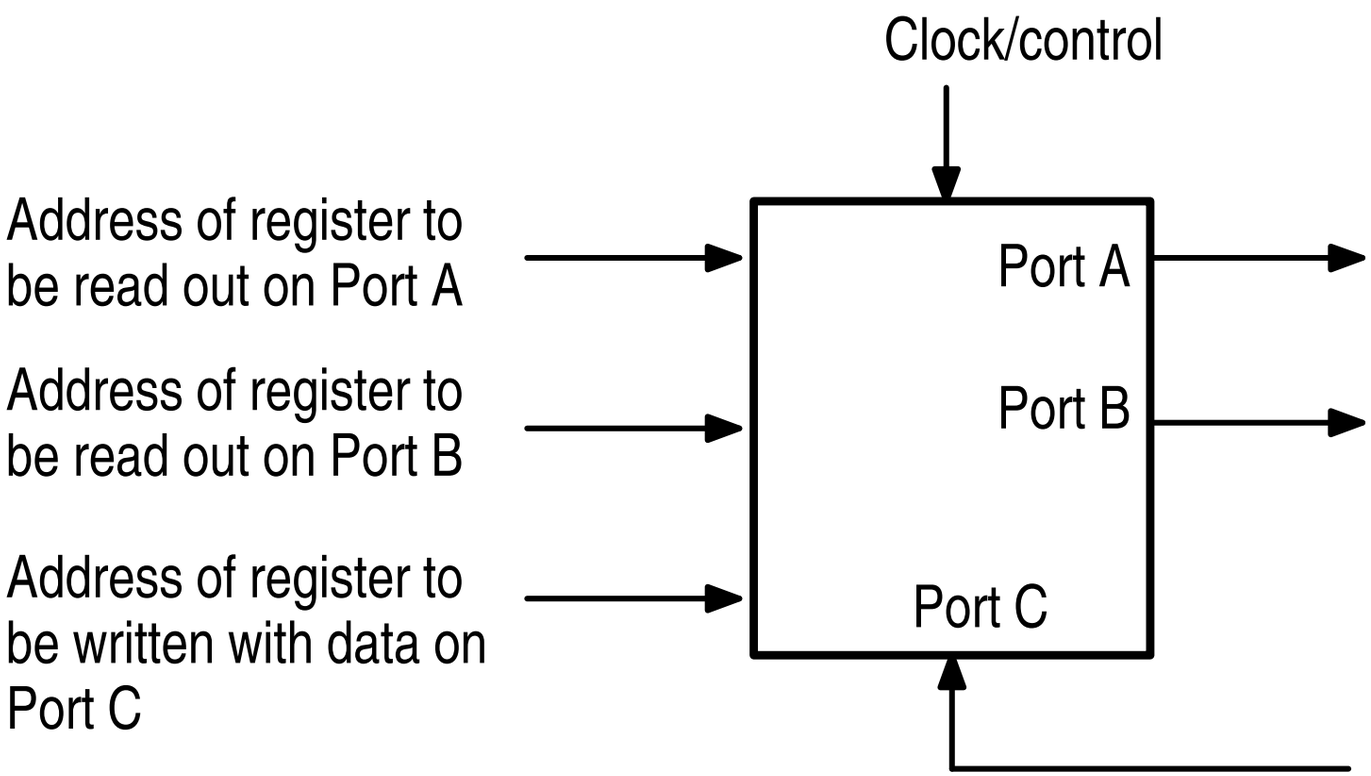
Decoding and register file read



At this point, decoded info is available to discard data read from register(s) whose address field(s) was/were incorrectly assumed to be valid

Register file

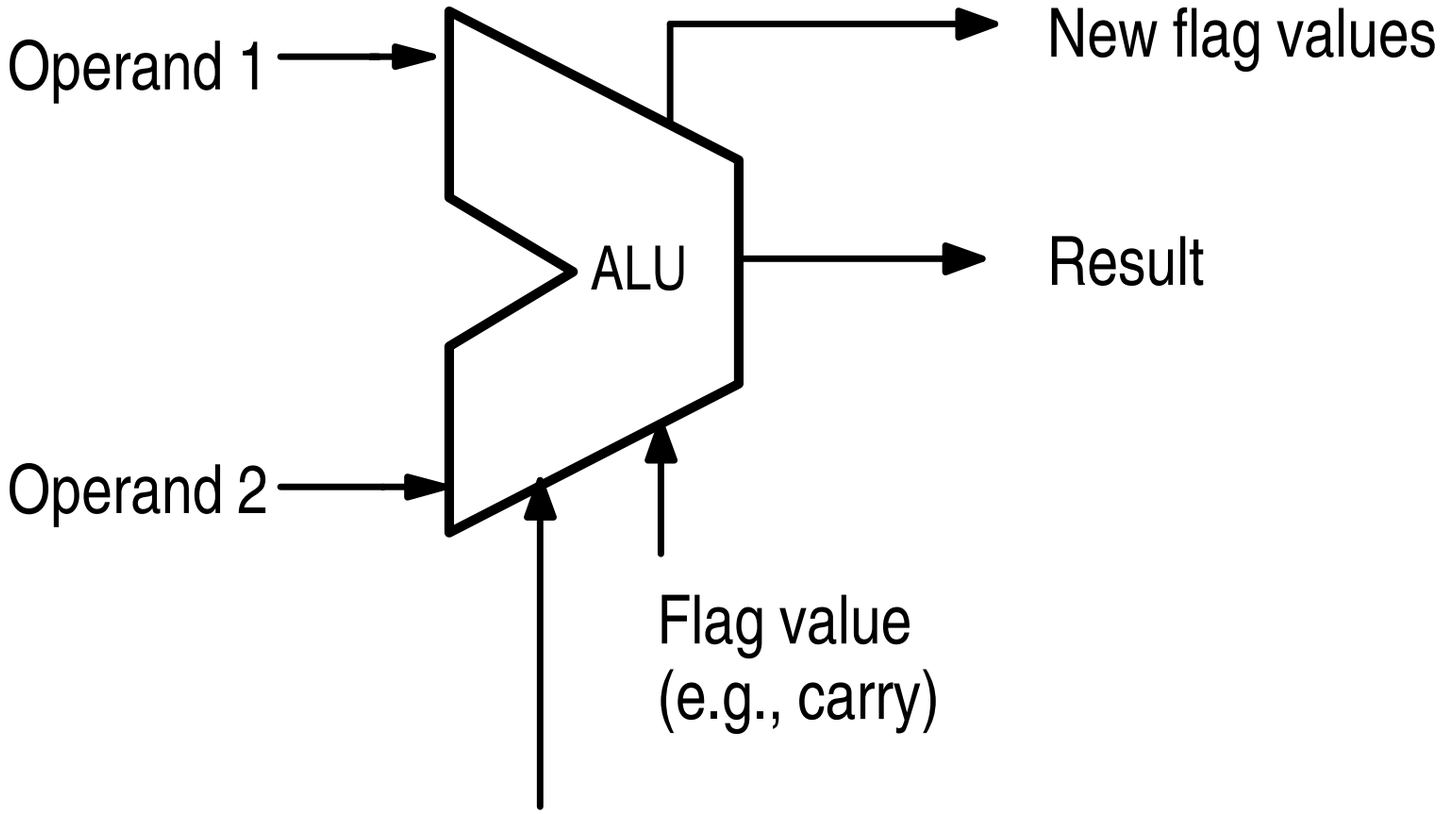
- Small RAM
  - Multiple ports: 2 read, 1 write simultaneously
  - Each location is a register



Combinational for reads

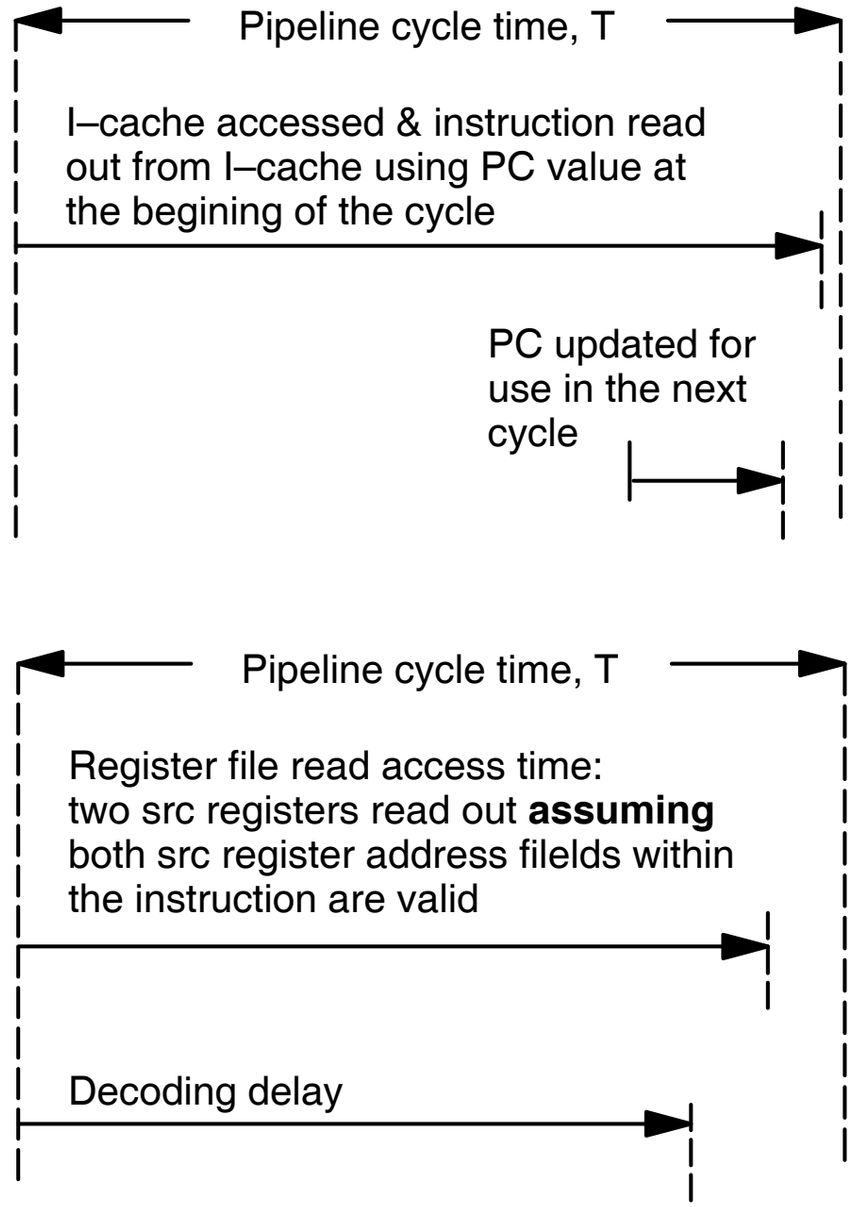
ALU

- Combinational circuit for arithmetic and logic ops



Logic utilization

- Logic delays of stages are not uniform
- Fastest logic waits for slowest logic to catch up (utilization loss)
- Solution: Normalize logic delays across stages



Dependencies and Parallelism

- ADD R1,R2,R3
- ADD R4,R5,R6

- ADD R1,R2,R3
- ADD R4,R5,R1

Dependencies and Parallelism

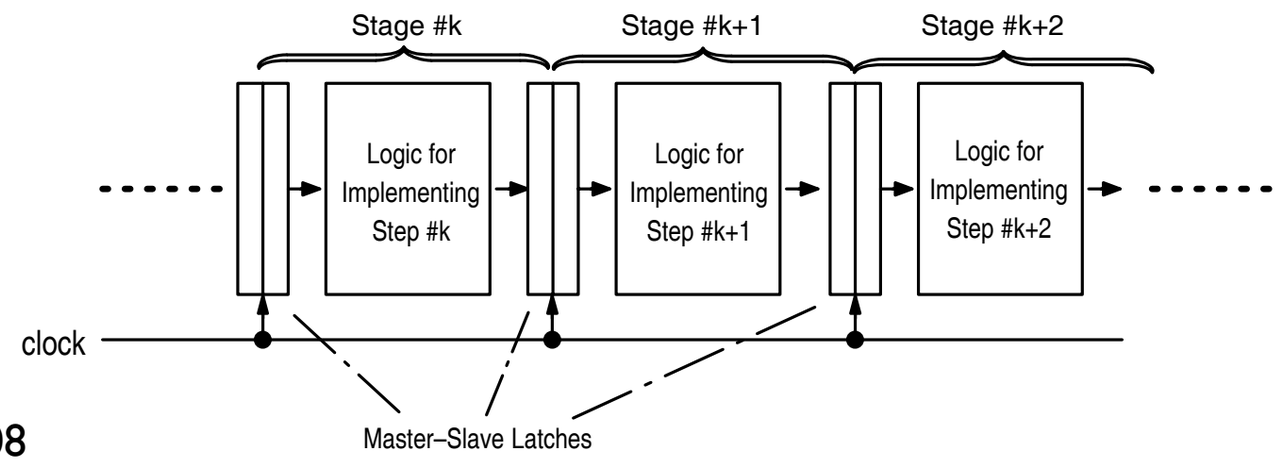
- ADD R1,R2,R3
- ADD R4,R5,R6

- ADD R1,R2,R3
- ADD R4,R5,R1

More on pipelining

- Pipelines exploit ILP -- Instruction-Level Parallelism
- Pipeline latency = number of cycles needed to process instruction

- Pipeline clock = common clock to drive interstage latches
- Clock rate = 1/T



Clocking issues

- Clock period must be long enough for:
  - Critical paths
  - Latch overhead
  - Clock skew

Pipeline efficiency

- Tp = Total cycles when any stages are committed
- K * Tp = Total resources committed over this duration (stage cycles)
- K * N = Actual usage of stages, each stage used once by N instructions

Pipeline efficiency

- Efficiency = Actual resources used/ committed
- = (K * N) / (K * Tp) = N / (K + N - 1)
- < 100% because some stages are unused during filling and flushing
- < 100% because dependencies cause stalls

Ideal CPI with pipelining

- Texec = Tp = K * T + (N - 1) * T
- = N * CPI * T
- Best possible CPI approaches 1 while N >> K and K > 1
- Throughput = N / Texec
- IPC = 1/CPI

Stalls

- If an instruction spends more than one cycle in a stage, prior stages *stall*.
  - Delays in fetching instruction, I-cache miss
  - Delays in accessing memory in MEM, D-cache miss
  - Waiting on dependencies
- Stalls cause "bubbles"

Real CPI

- Higher than 1 due to:
  - Data dependencies (interlocking)

```
LOAD   R1, R3, #50      /* R1 <- Mem [R3 + 50] */
ADD    R2, R1, R6       /* R2 <- R1 + R6 */
```

  - Branches
  - Slow memory
  - Resource contention

Real CPI for pipelines

- Design goal: CPI as close to 1 as possible
- f = average number of cycles per instruction lost to bubbles
  - CPI = (K - 1 + N*(1+f))/N = ~ 1+f
- Hardware and software solutions used to reduce CPI.

The APEX processor

ISA

APEX ISA

load: LOAD <dest> <src1> <literal>

store: STORE <src1> <src2> <literal>

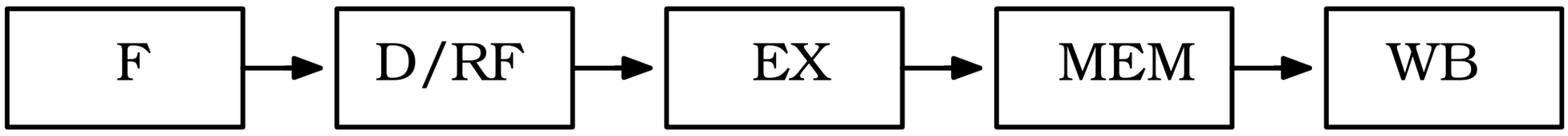
op: op <dest> <src1> <src2>

opL: op <dest> <src1> <literal>

APEX architecture

- Load-store architecture
- All instructions 32-bits
- 32 general-purpose registers, 0-31
- Registers and memory 32-bits
- Source operands: *rsrc*
- Destination operands: *rdest*
- *Mem[X]* is memory address at addr X.

APEX pipeline



The APEX Pipeline

Processing Step	Associated Stage Name	Abbreviation
Fetch Instruction	Fetch stage	F
Decode and fetch register operands	Decode, register fetch	D/RF
Execute register–to–register operation	Execute	EX
Perform memory operation, if any	Memory	MEM
Write results back	Writeback	WB

Instruction flow

- Instruction, operands, and addresses flow down pipeline as necessary.
- Program counter also: PC-relative addressing, return address for subroutines and interrupts

Example instructions

```
ADD rdest rsrc1 rsrc2 :     rdest <- rsrc1 + rsrc 2

ADDL rdest rsrc1 literal : rdest <- rsrc1 + literal

LDI rdest rsrc1 literal :  rdest <- Mem [rsrc1 + literal]

LDIR rdest rsrc1 rsrc2 :    rdest <- Mem [rsrc1 + rsrc2]

STI rsrc1 rsrc2 literal :  Mem [rsrc2 + literal] <- rsrc1
```

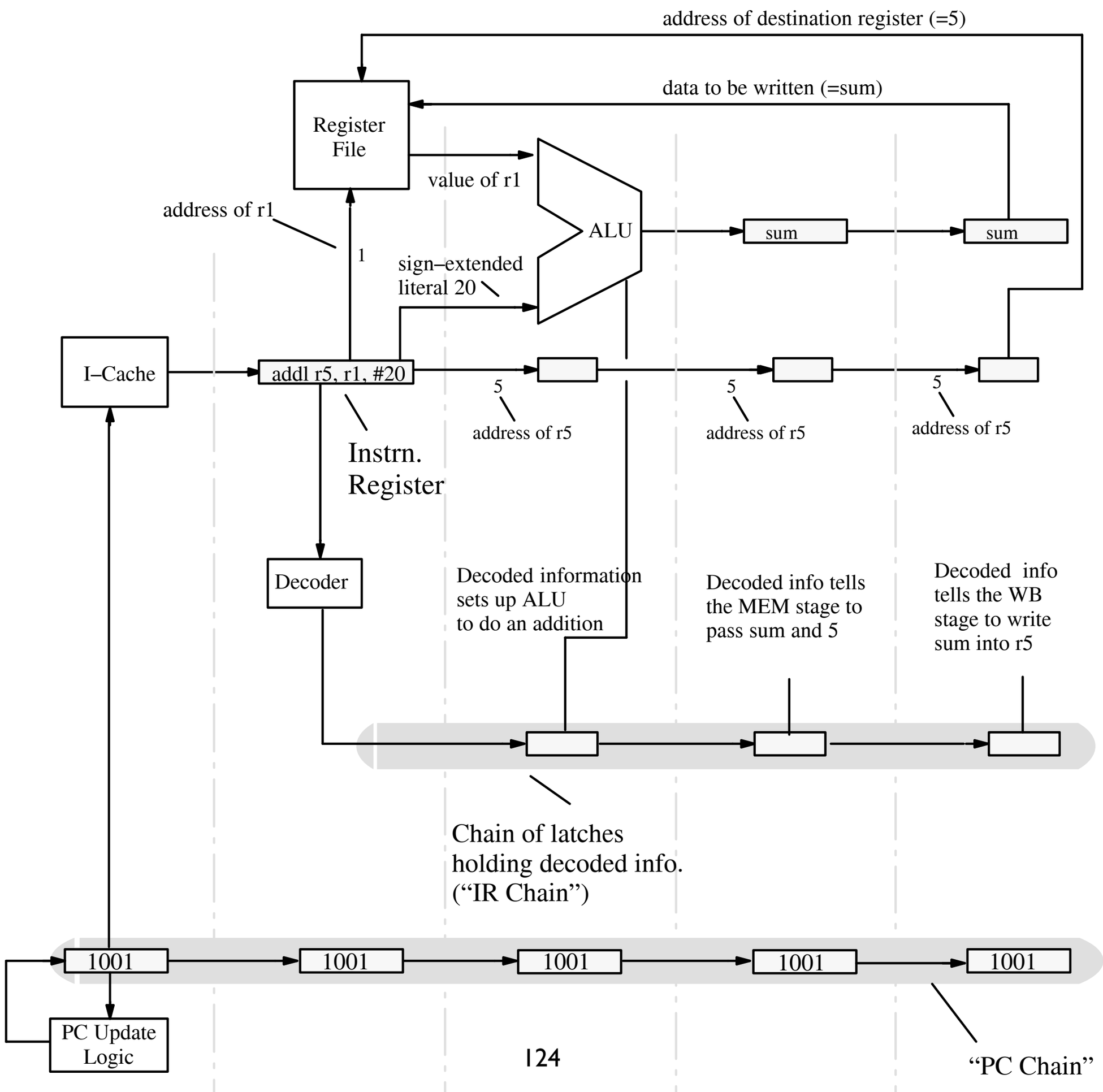
INSTRUCTION \ Stage	F stage	D/RF stage	EX stage	MEM stage	WB stage
Reg–to–reg: ADD rdest, rsrc1, rsrc2	Read instruction from memory	Decode instruction; Read rsrc1 and rsrc2 from the RF	Add contents of rsrc1 and rsrc2; Set condition code flags	Hold on to the result produced in the EX stage in the previous cycle;	Write result to register rdest
Load indexed–literal offset: LDI rdest, rsrc1, <literal>	Read instruction from memory	Decode instruction; Read rsrc1; sign extend <literal>	Add contents of rsrc1 and sign extended literal; Set condition code flags;	Read the contents of the memory location whose address was computed in the previous cycle in the EX stage	Write the data read out from memory to the register rdest
Load indexed–reg. offset: LDIR rdest, rsrc1, rsrc2	Read instruction from memory	Decode instruction; Read rsrc1 and rsrc2 from the RF	Add contents of rsrc1 and rsrc2; Set condition code flags	Read the contents of the memory location whose address was computed in the previous cycle in the EX stage	Write the data read out from memory to the register rdest
Store indexed–literal offset: STI rsrc1, rsrc2, <literal>	Read instruction from memory	Decode instruction; Read rsrc1 and rsrc2; sign extend <literal>	Add contents of rsrc2 and sign extended literal; Set condition code flags	Write the contents of rsrc1 (read out in the D/RF stage) to the memory location whose address was com-puted in the EX stage in the previous cycle	No action

Use of the stages of APEX by register–to–register, load and store instructions

INSTRUCTION \ Between Stages	F and D/RF	D/RF and EX	EX and MEM	MEM and WB
Reg–to–reg: ADD rdest, rsrc1, rsrc2	– The instruction read from memory – The PC value	– Instrn. info; – The PC value – Contents of rsrc1 and rsrc2; – Address of rdest	– Instrn. info; – The PC value – Address of rdest; – Result of operation in the EX stage	– Instrn. info; – The PC value – Address of rdest; – Result of operation in the E stage
Load indexed–literal offset: LDI rdest, rsrc1, <literal>	– The instruction read from memory – The PC value	– Instrn. info; – The PC value – Contents of rsrc1; – Sign extended literal – Address of rdest	– Instrn. info; – The PC value – Address of rdest; – Result of operation in the EX stage	– Instrn. info; – The PC value – Address of rdest; (Data being loaded comes from memory)
Load indexed–reg. offset: LDIR rdest, rsrc1, rsrc2	– The instruction read from memory – The PC value	– Instrn. info; – The PC value – Contents of rsrc1 and rsrc2; – Address of rdest	– Instrn. info; – The PC value – Address of rdest; – Result of operation in the EX stage	– Instrn. info; – The PC value – Address of rdest; (Data being loaded comes from memory)
Store indexed–literal offset: STI rsrc1, rsrc2, <literal>	– The instruction read from memory – The PC value	– Instrn. info; – The PC value – Contents of rsrc1 and rsrc2; – Sign extended literal	– Instrn. info; – The PC value – Contents of rsrc1; – Result of operation in the EX stage	None

* “Instrn. Info” indicates information about the instruction, such as the instruction itself or its decoded form – see text

Information flow among the stages of APEX for register–to–register, load and store instructions


address of destination register (=5)
data to be written (=sum)
Register File
value of r1
address of r1
1
ALU
sum
sum
sign–extended literal 20
I–Cache
addl r5, r1, #20
5
address of r5
5
address of r5
5
address of r5
Instrn. Register
Decoder
Decoded information sets up ALU to do an addition
Decoded info tells the MEM stage to pass sum and 5
Decoded info tells the WB stage to write sum into r5
Chain of latches holding decoded info. (“IR Chain”)
1001
1001
1001
1001
1001
PC Update Logic
“PC Chain”

Features of APEX pipeline

- Not all instructions use all stages -- data forwarded
  - Reg-to-reg: No processing in MEM
  - Store: WB not used.
- Registers read only in D/RF
  - Minimize number of ports
- Registers written only in WB
  - Even if result is available early
- EX state implements ALU and effective address calculation for LOAD and STORE

Dependencies in Pipelines, long pipelines, multifunction pipelines

Functional Pipelines

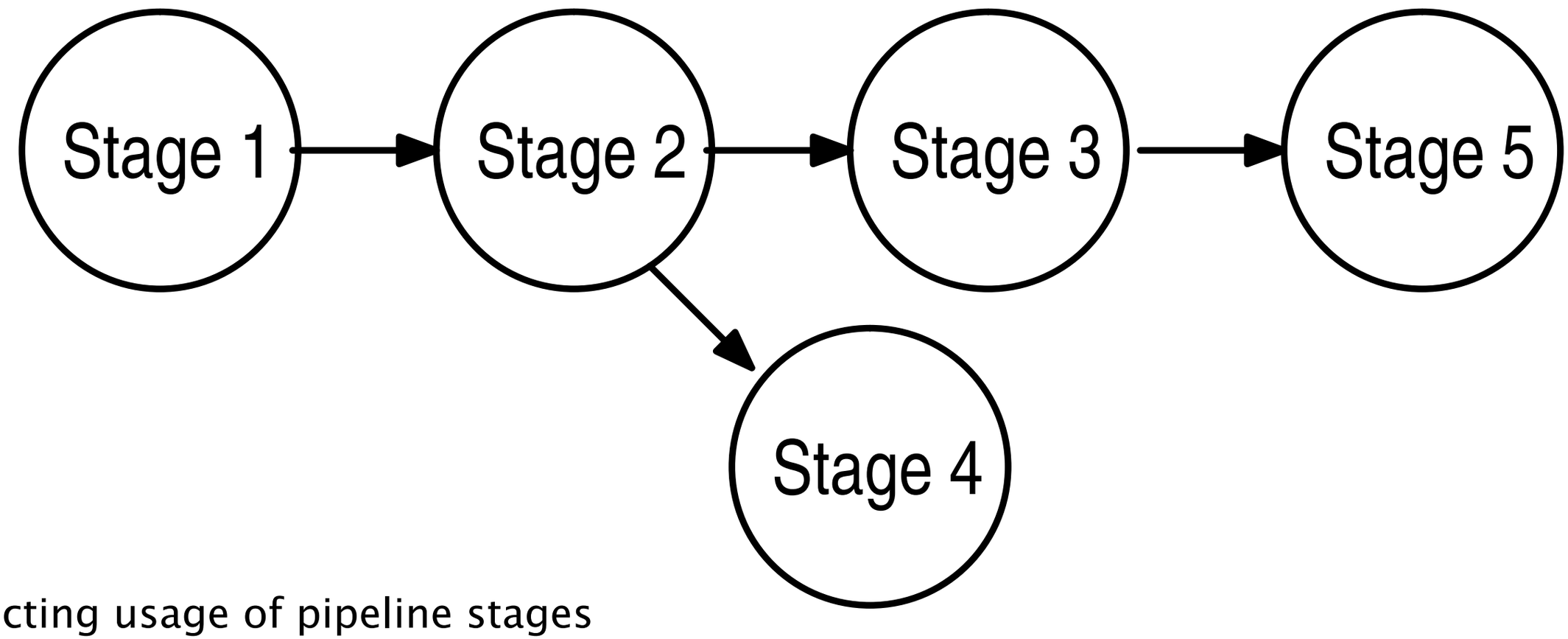
- Why pipeline?
- Pipelining to divide work among functional units
- Complex operations can be pipelined to increase throughput.

Optimizing number of pipeline stages

- Calculating Speedup (S)
- S = N * K * T / (K * T + (N-1) * T)
- S = Tnp / Tp
- Smax = K (N >> K, K > 1)

Precedence Graphs

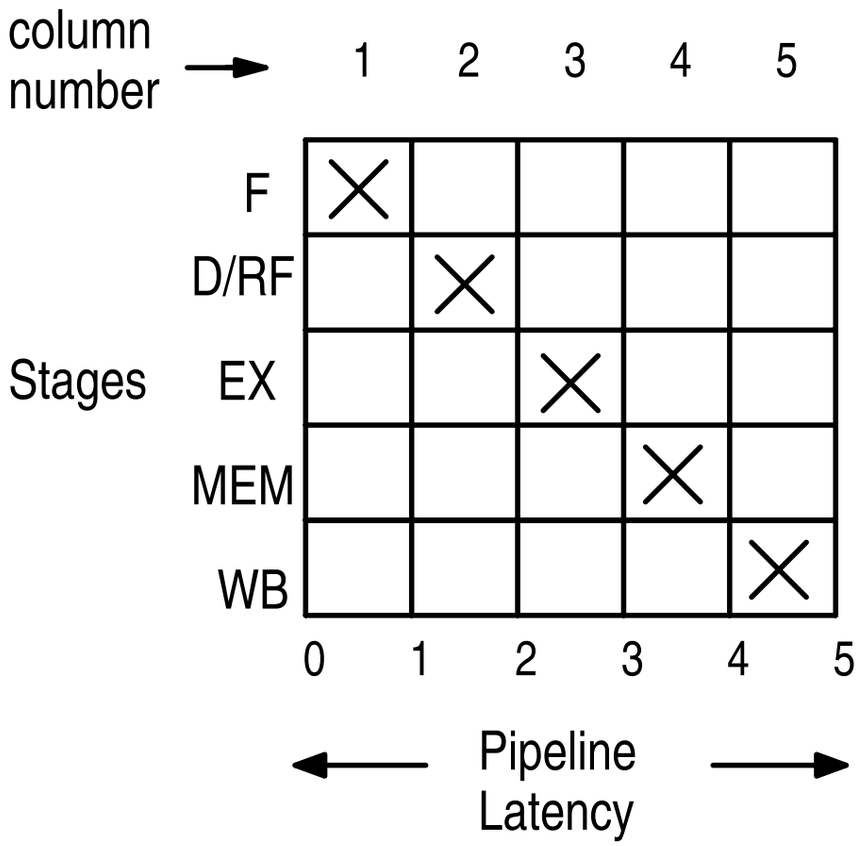
- nodes = stages
- directed arcs = input-output order



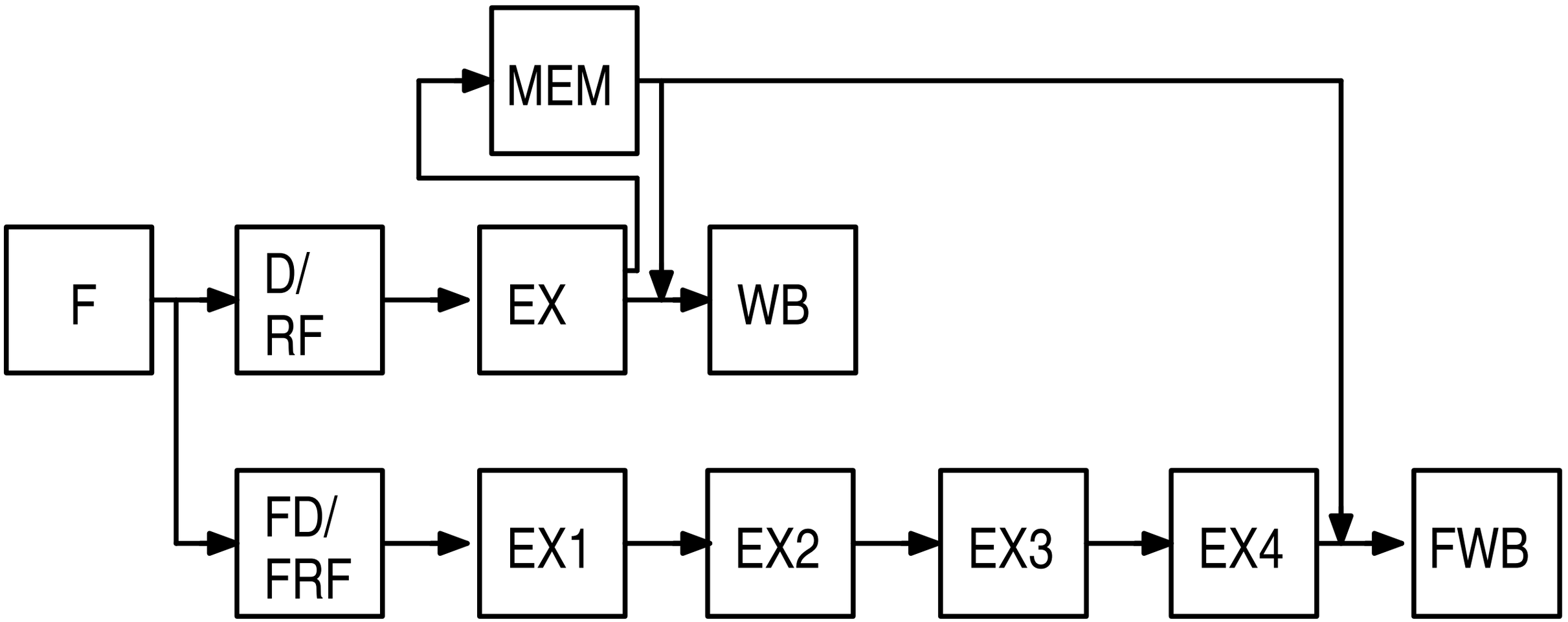
Depicting usage of pipeline stages

Reservation tables

- One row per stage
- One column per use by instruction (time slot)
- Total columns = pipeline latency in cycles
- X marks usage of stage S in slot m.



More realistic pipeline



Features of realistic pipeline

- Separate registers for floating point and integer data.
- Separate execution units for
  - Integer (IU)
  - Floating point (FPU)
  - Load/store unit (LSU)
- Floating point execution units more deeply pipelined.
- FPU and MEM may be further pipelined
- Potential resource conflict in using WB and FWB to write RF and FRF.

Pipeline Classifications

- Linear vs. nonlinear (feedback)
  - For nonlinear, some rows in reservation table can have two or more X marks
- Synchronous vs. asynchronous
  - Synchronous: Common clock
  - Asynchronous: Clockless, handshake signals

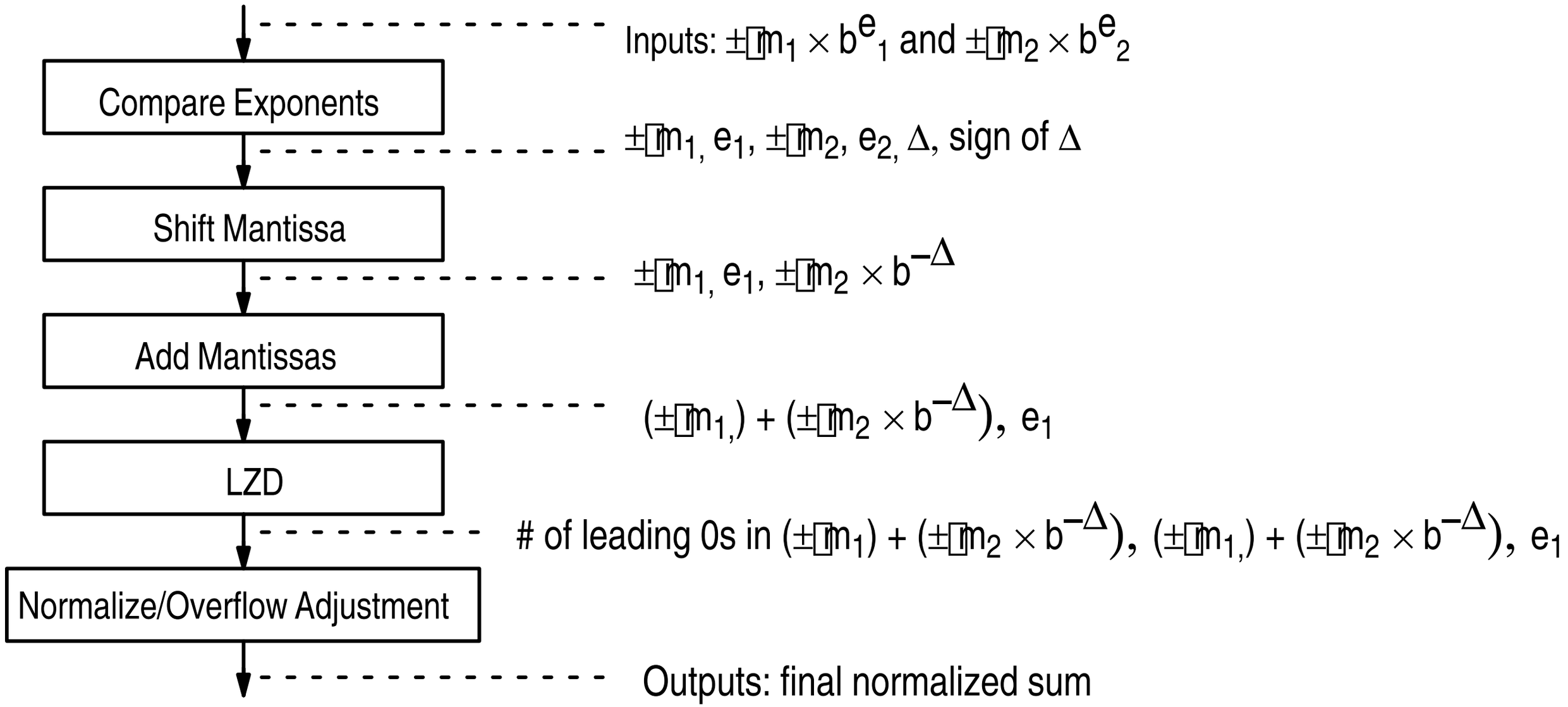
Floating Point addition

- Input operands
  $$m_1 \times b^{e}{}_1 \text{ and } m_2 \times b^{e}{}_2$$
- Base b usually power of 2, assume 2
- Normalized format
  - Leading 1 of mantissa needn't be stored
- Logic blocks:
  - Adders, comparators, barrel shifters, count leading zeros

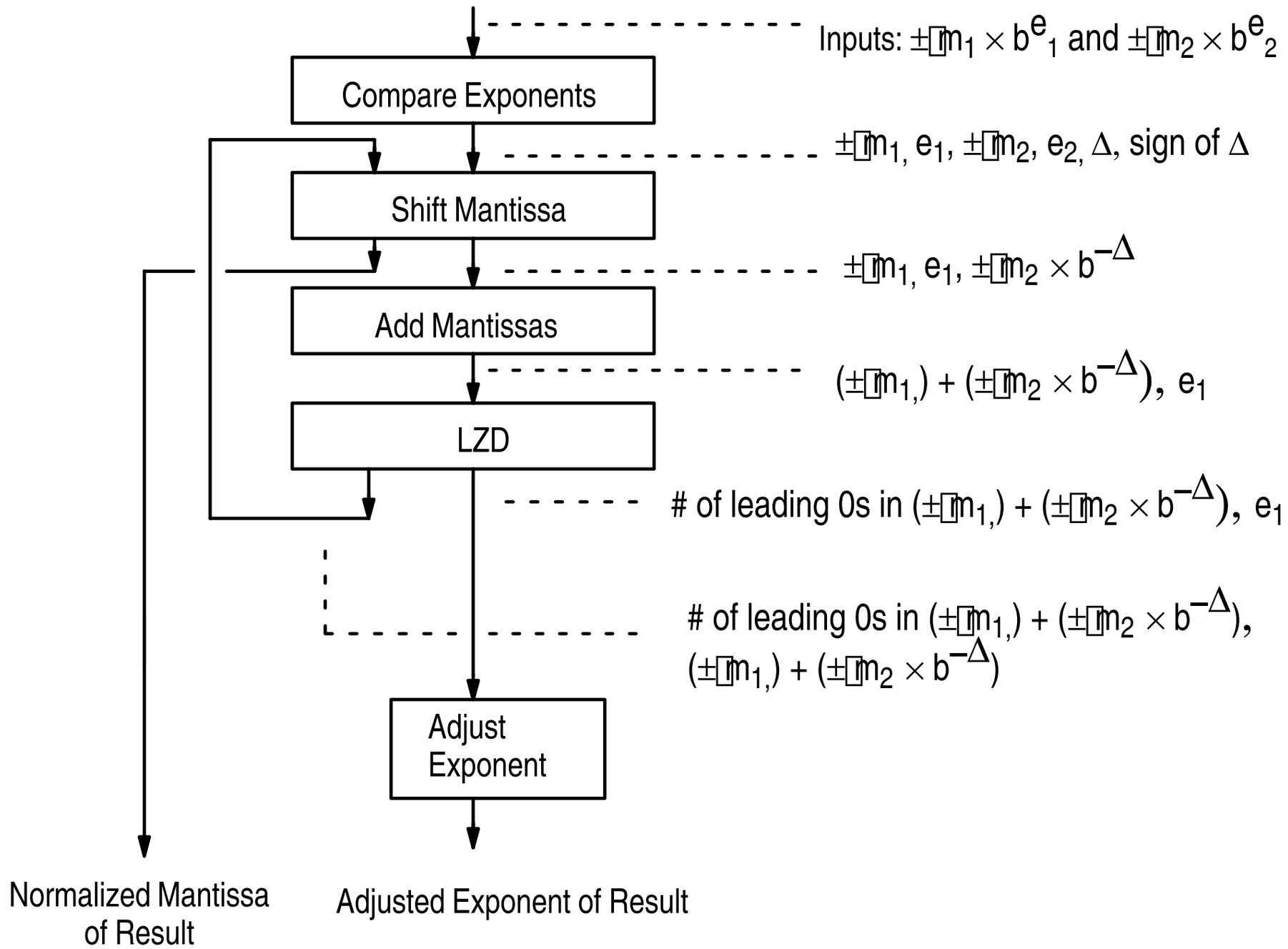
FP ADD Algorithm

- 1: Compare exponents
  - Δ = | e1 - e2 |
- 2: Shift m1 right to so e1 = e2
- 3: Add m3 = m1 + m2
  - new exponent e3 = e1
- 4: Normalize m3
  - Shift m3 left and dec e3 until MSB is 1

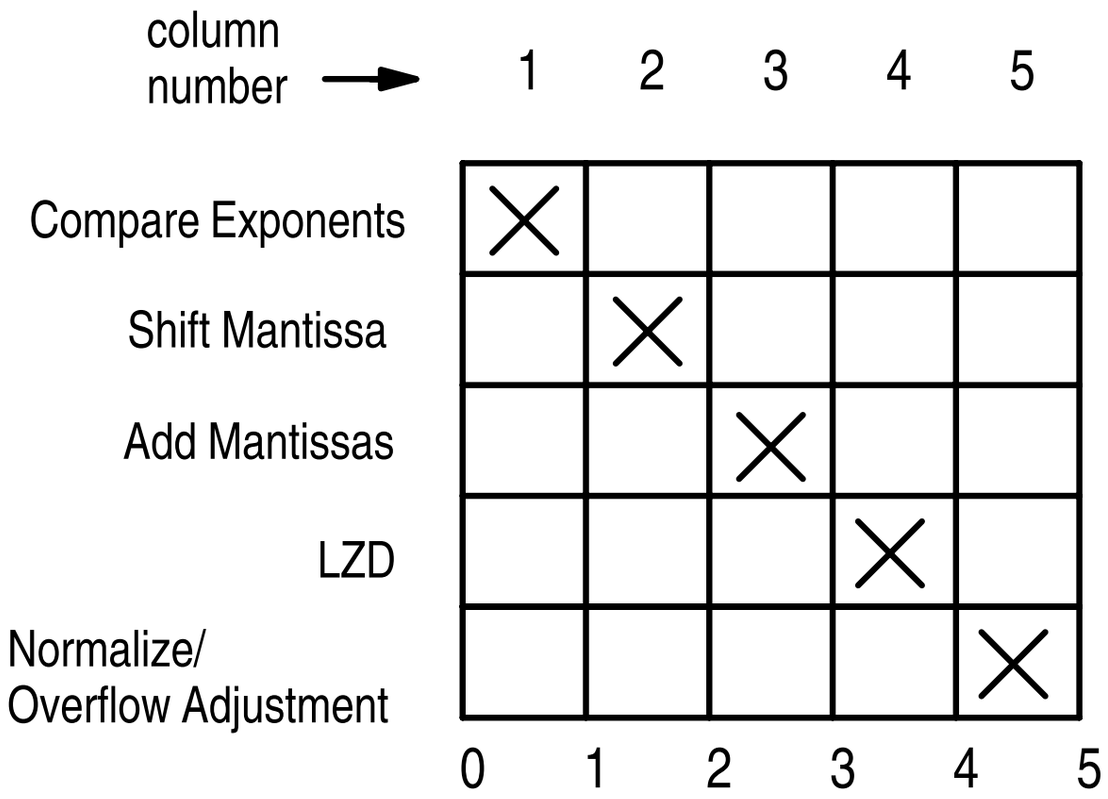
FP Adder


Inputs: $\pm m_1 \times b^{e_1}$ and $\pm m_2 \times b^{e_2}$
Compare Exponents
$\pm m_1, e_1, \pm m_2, e_2, \Delta$, sign of Δ
Shift Mantissa
$\pm m_1, e_1, \pm m_2 \times b^{-\Delta}$
Add Mantissas
$(\pm m_1) + (\pm m_2 \times b^{-\Delta}), e_1$
LZD
# of leading 0s in $(\pm m_1) + (\pm m_2 \times b^{-\Delta}), (\pm m_1) + (\pm m_2 \times b^{-\Delta}), e_1$
Normalize/Overflow Adjustment
Outputs: final normalized sum

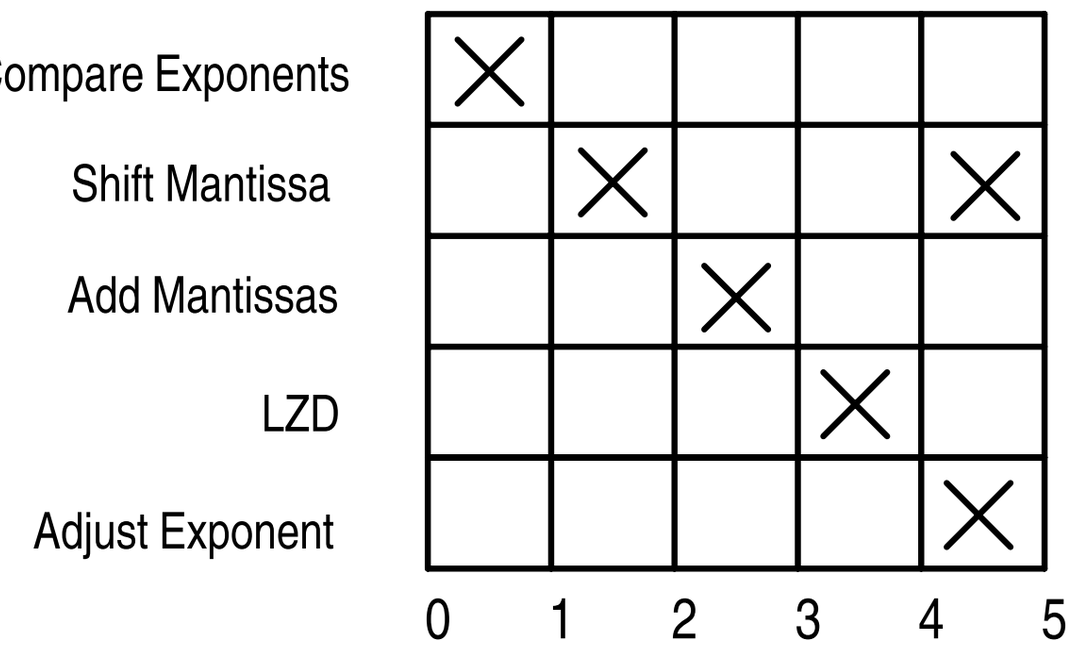
FP Adder (alternate)



FP Add reservation tables



Reservation table for the floating point adder of Figure 2.7(a)

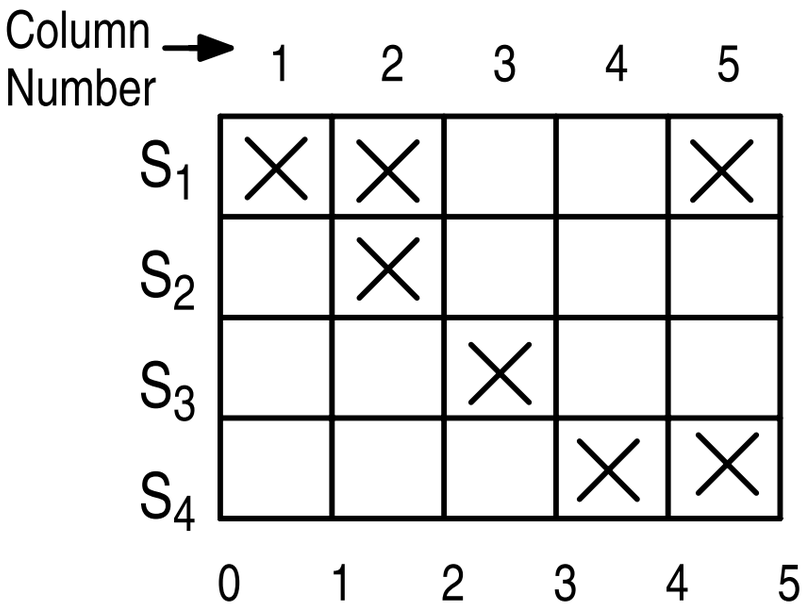


Reservation table for the floating point adder of Figure 2.7(b)

Nonlinear pipeline throughput

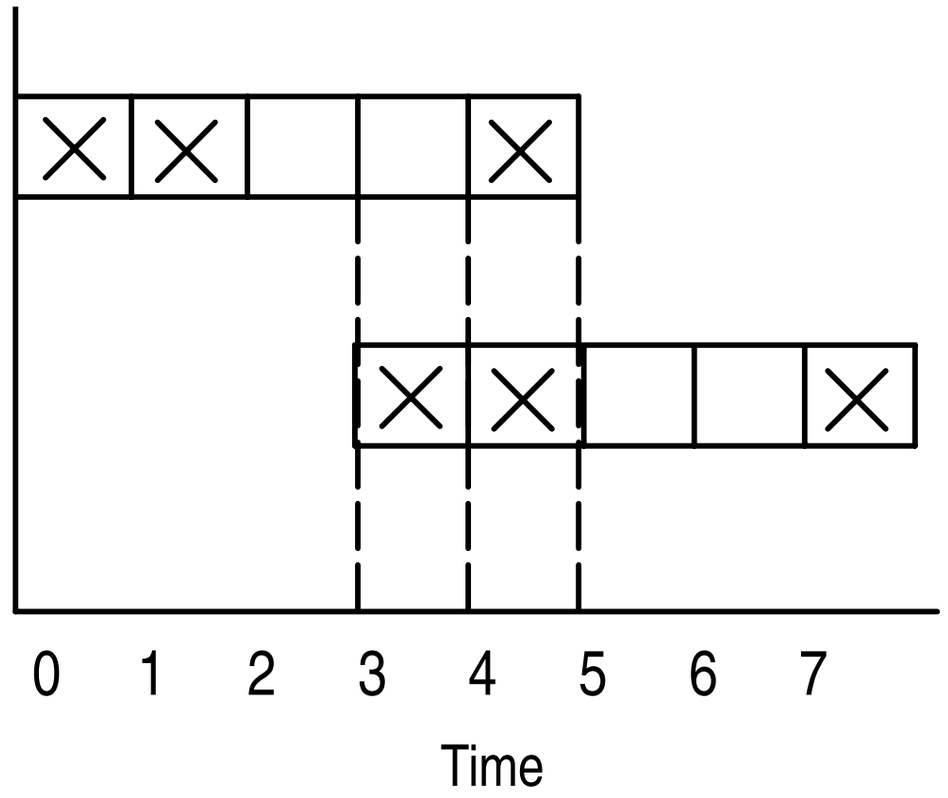
- Two or more X's in a row of res. table
  - Stage used more than once by same op.
- Already started op may contend for use of this stage with an op started later.
  - Collision
  - Cannot initiate new op every cycle

Nonlinear pipeline throughput



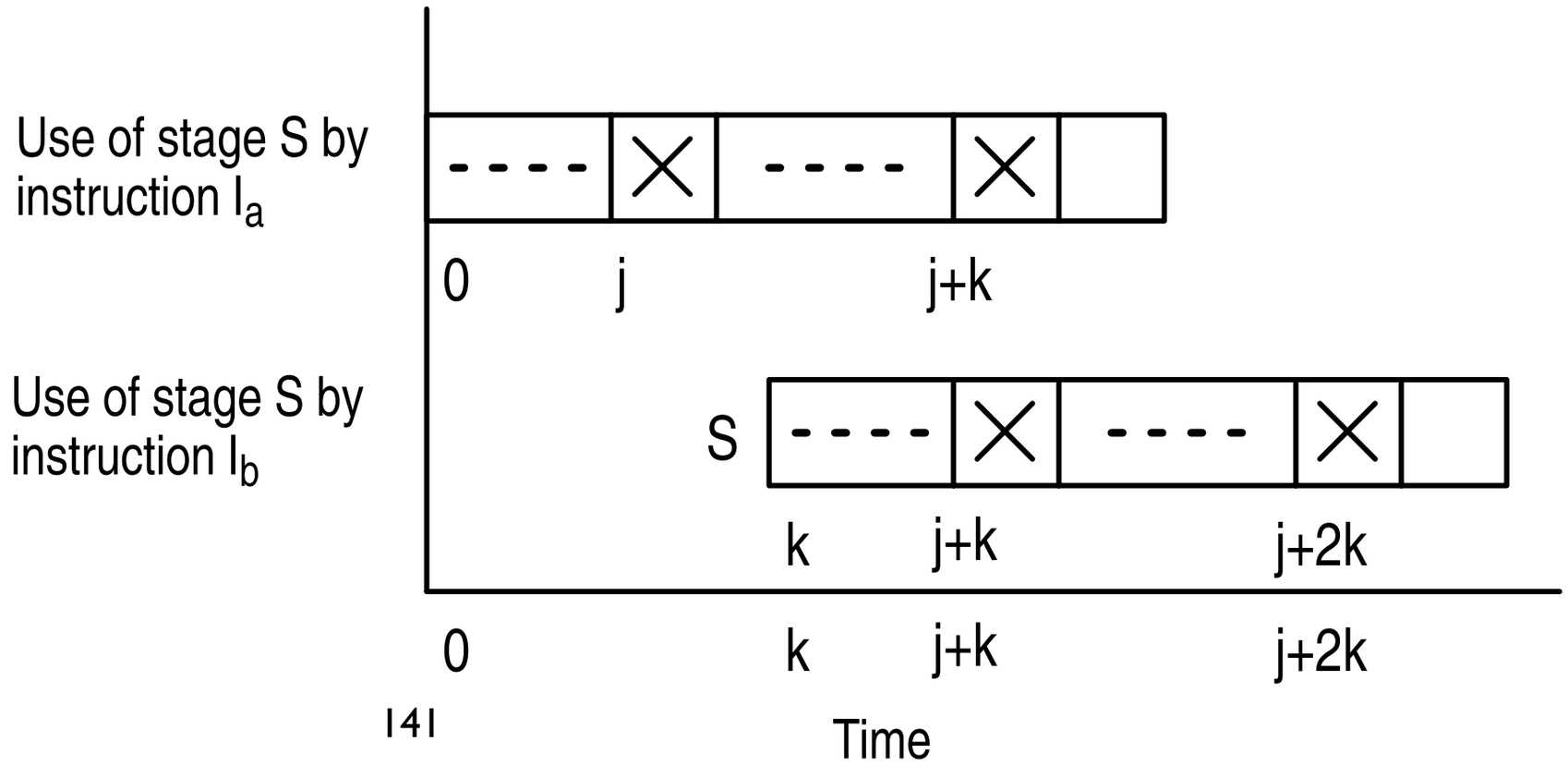
Use of stage S1 by instruction I_a

Use of stage S1 by instruction I_b initiated 3 cycles after I_a was initiated



Nonlinear pipeline throughput

- In general:
  - If the res. table has a row where any two X marks are separated by distance k, the two ops initiated at an interval k will collide.



Collision avoidance: Static

- Scheduling interval not equal to k
- Can be done by compiler
- Trivial:
  - Any distance higher than pipeline latency
  - Any distance higher than max of all distances between any two X's in same row.
- Better:
  - More complex schedules that produce best throughput while avoiding collisions.

Collision avoidance: Dynamic

- Use hardware to avoid collisions -- delay initiation of new operation to avoid collision with already-issued ops.

Terminology

- ***Initiation***: Starting of new op in pipeline
- ***Initiation latency***: Time between two successive initiations
- ***Initiation cycle***: Repetitive sequence of initiations
- ***Forbidden latency***: Initiation latency that causes collisions
- ***Collision-free initiation***: Initiation that does not collide with already-initiated ops.

Consequence of feedback pipeline

- M = number of X's in row with most X's
- Tp = Runtime on feedback architecture
- N = Number of instructions
- Busiest stage uses N*M cycles.
- Efficiency: $\varepsilon = N * M / Tp \leq 1$
- Throughput: $\tau = N / Tp$
- Implication: $\tau \leq 1 / M$

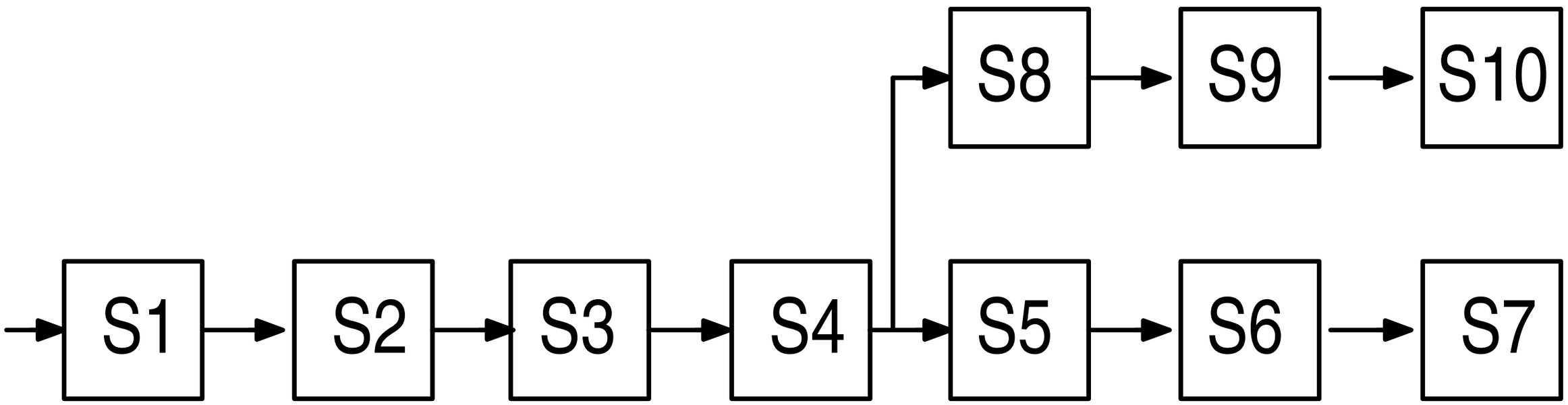
$$MAT \leq MPRT = 1/M$$

Limitations on pipelining

- Factors
  - Clock skew
  - Latch overhead
  - Pipeline bubbles, data dependencies
  - State info to save/restore on interrupt
- Consequences
  - Limits pipelines to 4 to 8 stages
  - Longer pipelines are *decoupled*, using queues.

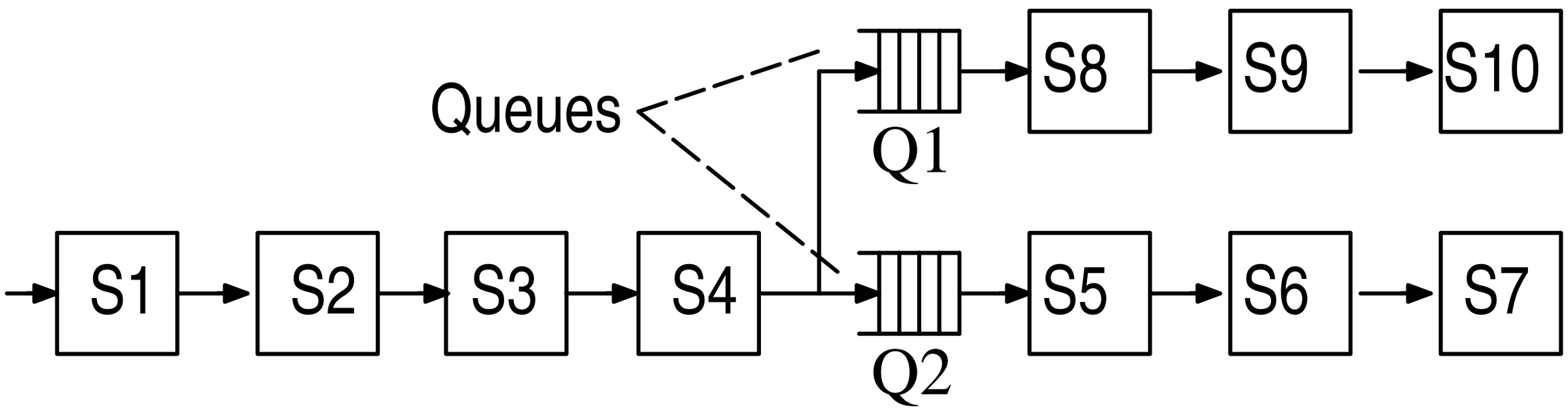
Non-decoupled pipeline

- Clock rate accounts for larger clock skew
- Blocking stage can stall whole pipeline



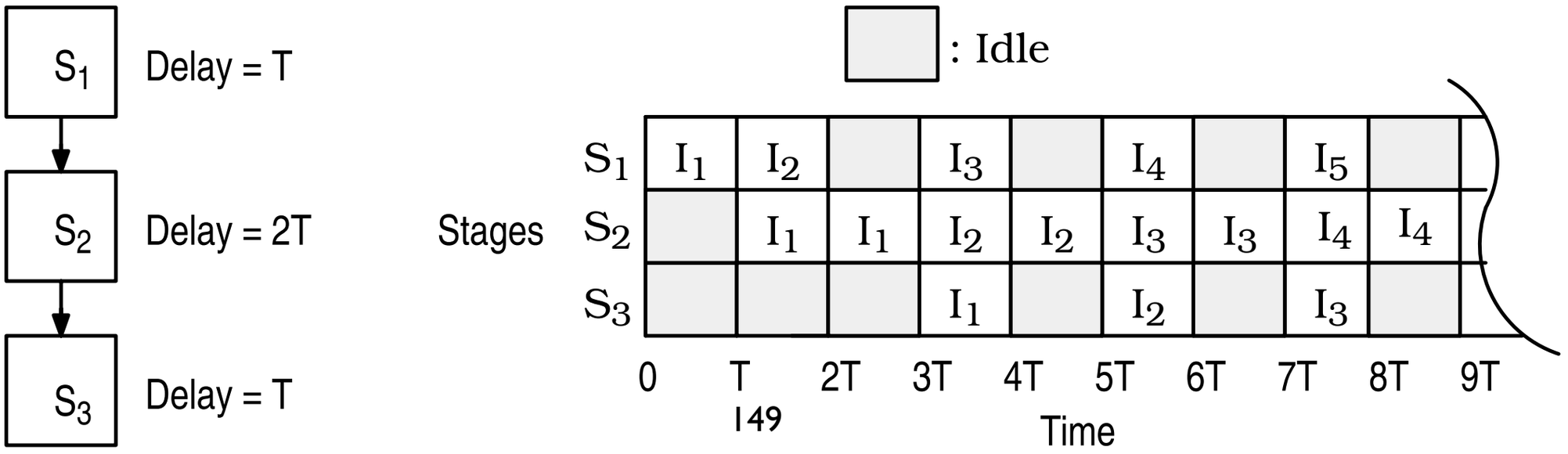
Decoupled pipeline

- Three independent pipeline segments
- Independently clocked (same frequency)
- Queues help hide stalls



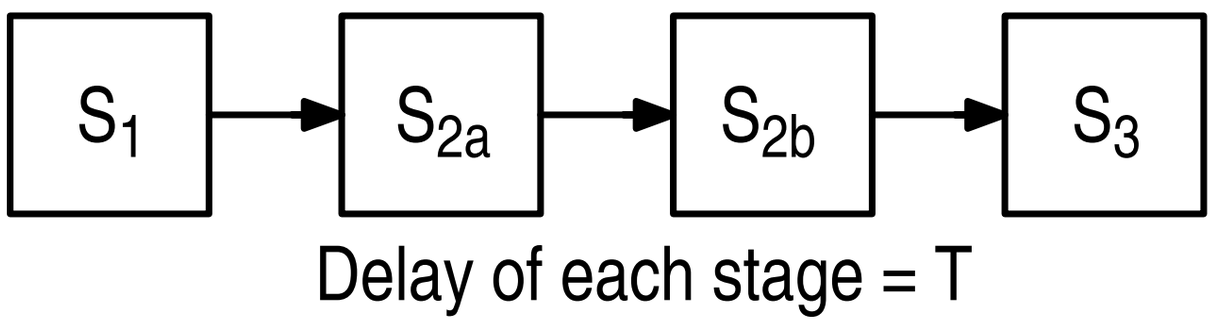
Impact of Pipeline Bottlenecks

- K-stage pipeline with non-uniform stage delays: $T_p = \Sigma_i T_i + (N–1).T_s$
- $T_s = \max \{ T_i \}$
- Instructions complete one every Ts
- Slowest stage is performance bottleneck.
- Stage with delay Ti unutilized for (Ts - Ti)



Segmentation

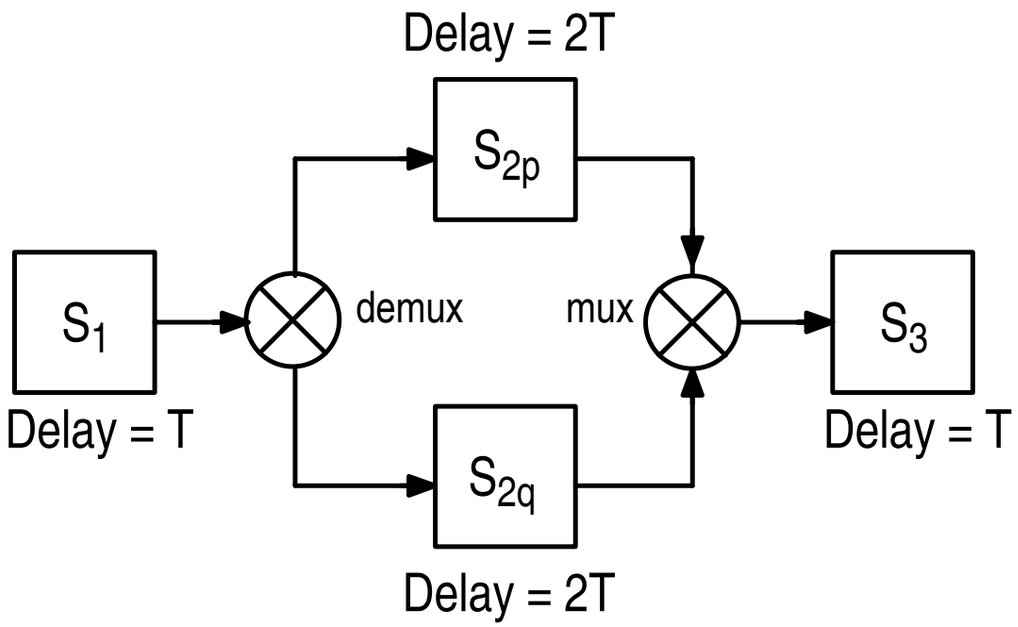
- Split slower stages into sequences of simpler stages.



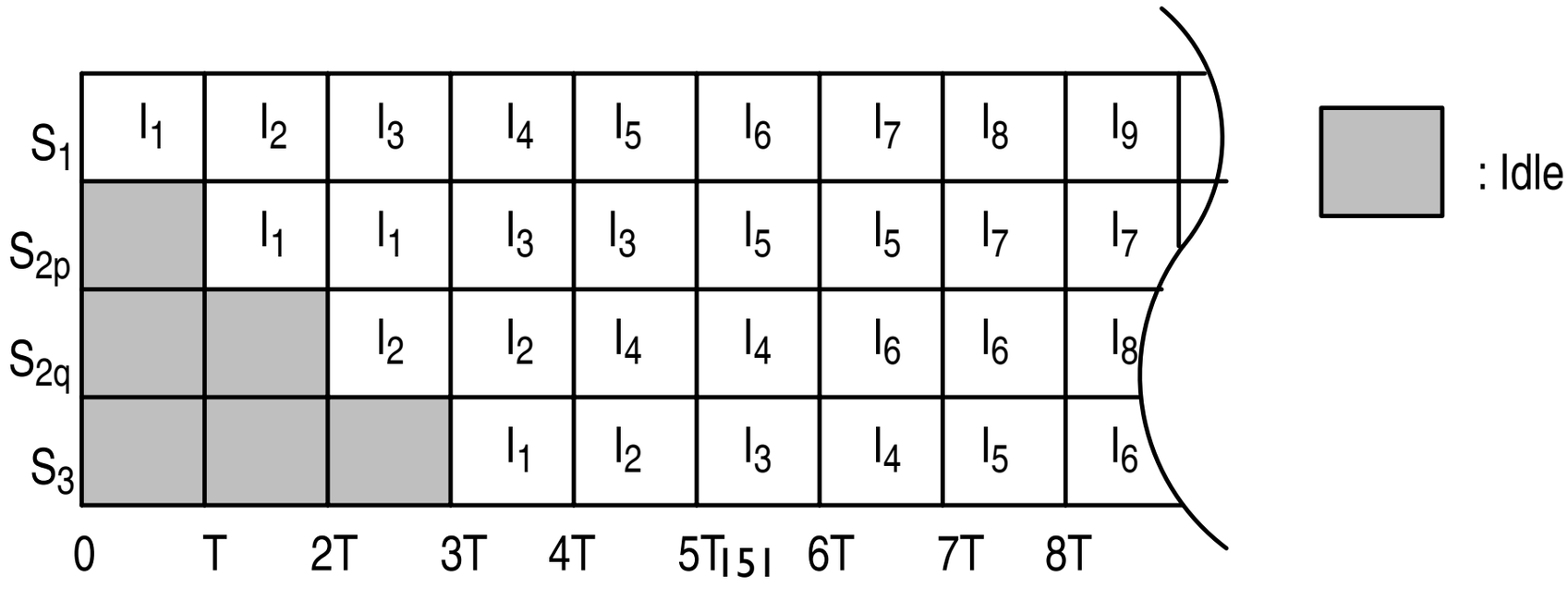
Latency = 4.T
Maximum Throughput = 1/T

Replication

- Replicate slower stages and alternate



Latency = 4.T
Maximum Throughput = 1/T

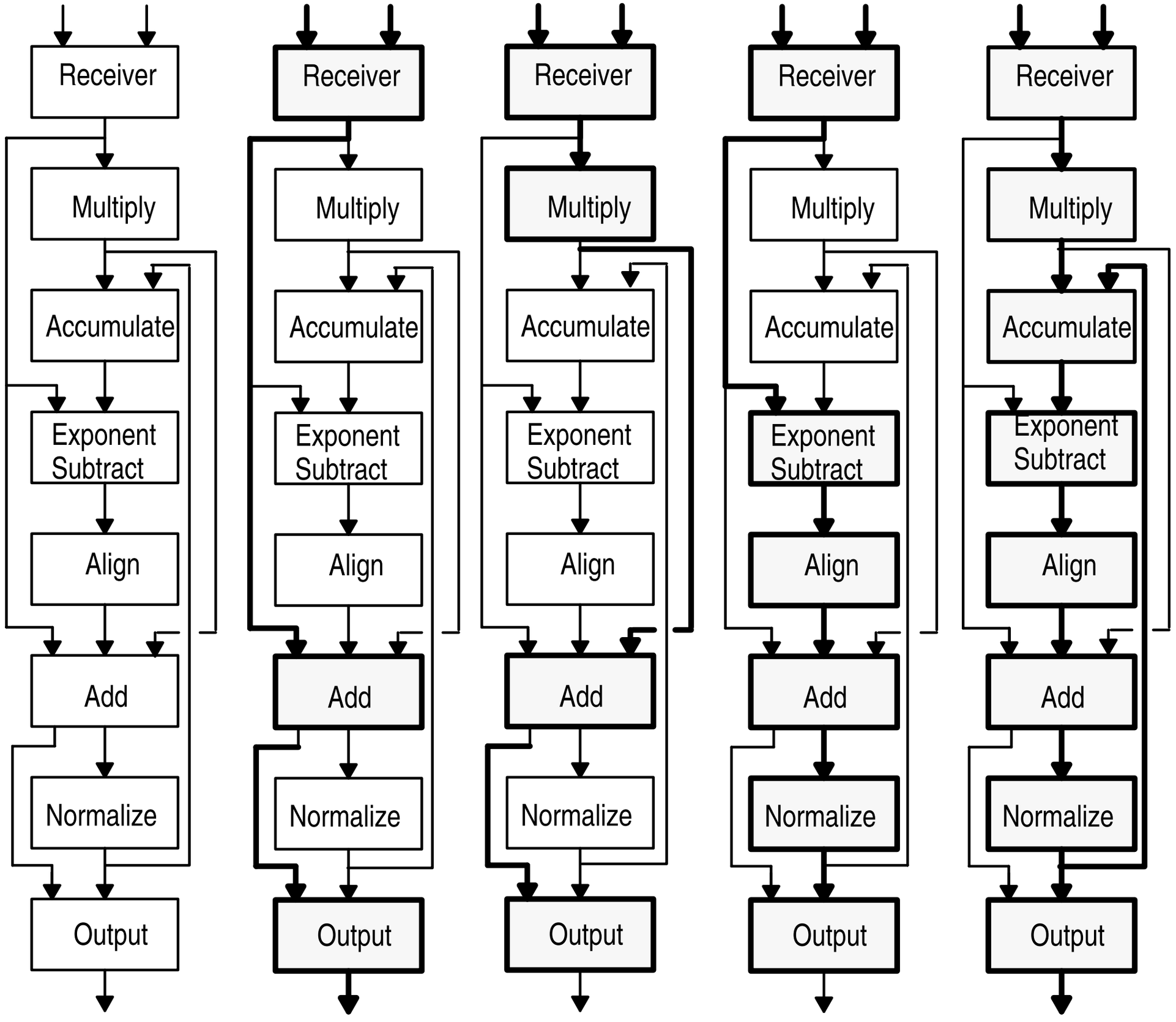


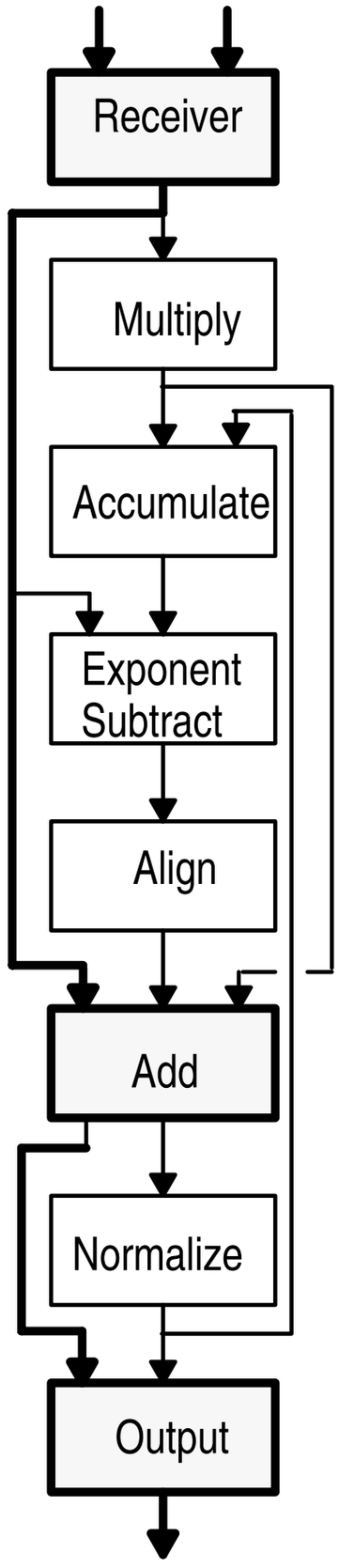
Pipeline Classifications

- Unifunction vs. multifunction
  - Unifunction: Pipeline implements one function
  - Multifunction: Common pipeline for multiple operations
- Static vs. Dynamic
  - Static: Fixed function
  - Dynamic: Reconfigurable on the fly

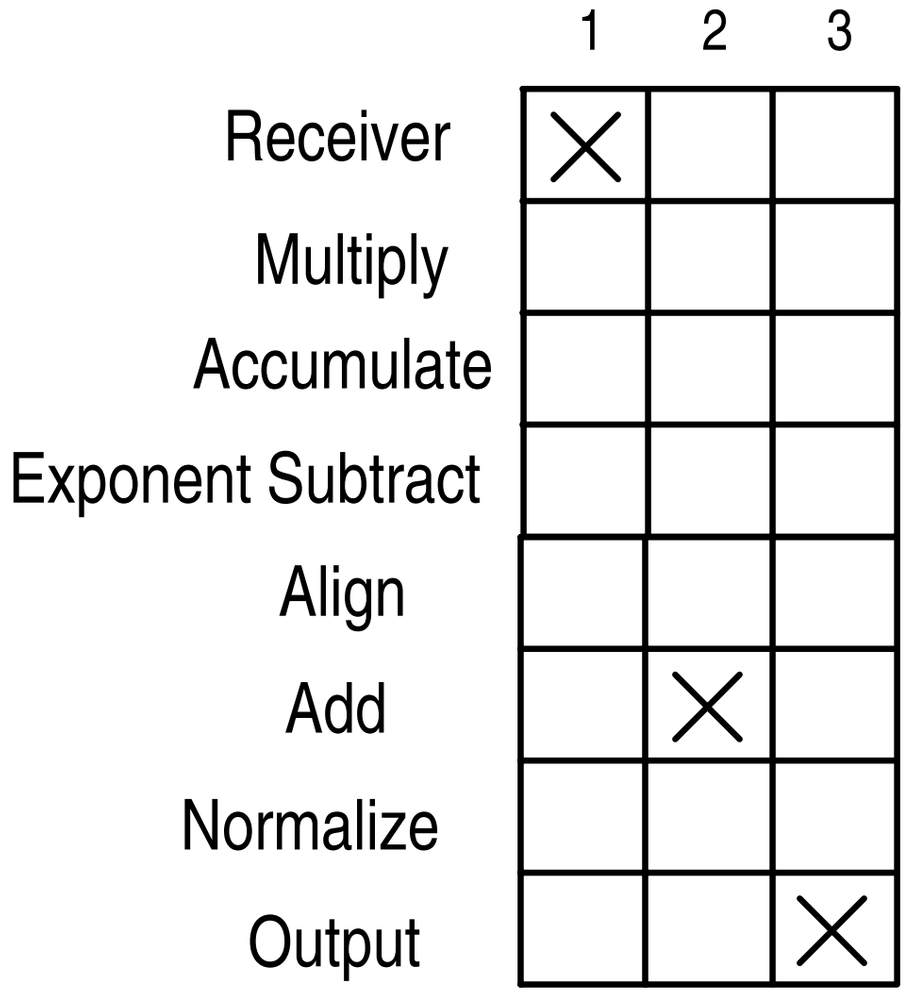
Multifunction pipelines

- Example: The arithmetic unit pipeline of TI ASC (Advanced Scientific Computer)
- All reconfiguration done by microcode.
- Cycle time is 60ns.
- Inputs and outputs are vectors.

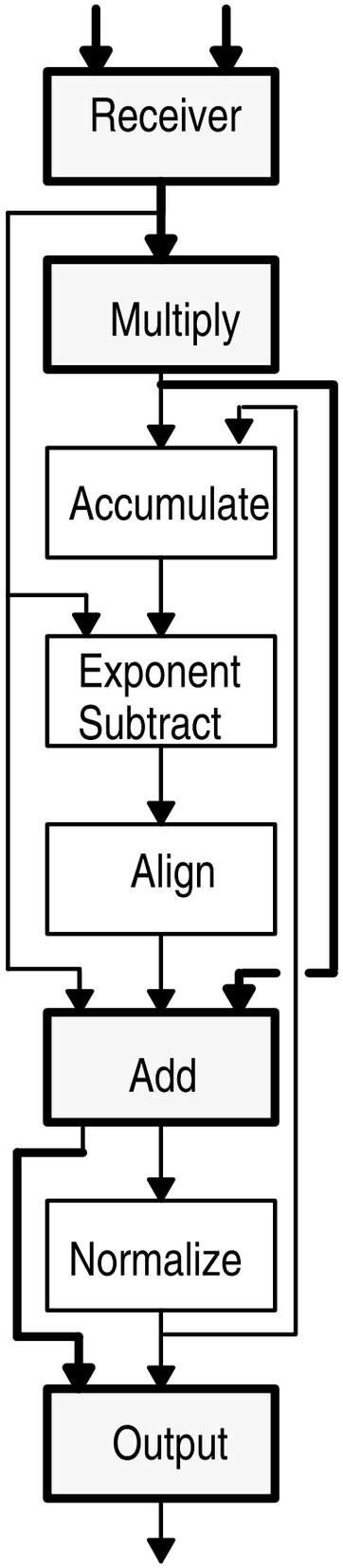

Receiver
Multiply
Accumulate
Exponent Subtract
Align
Add
Normalize
Output
Basic Pipeline
Integer Add
Integer Multiply
Floating Point Add
Floating Point Vector Dot Product



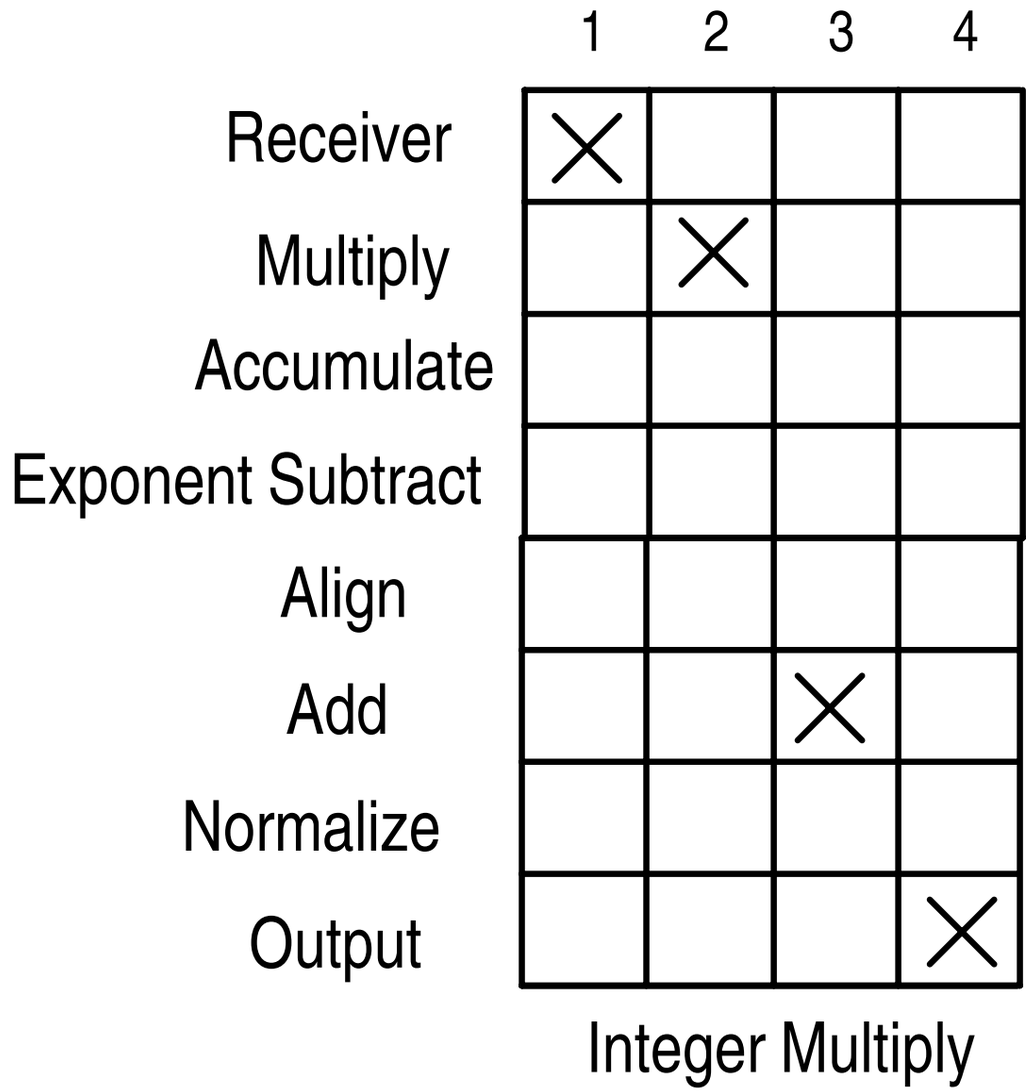
Integer Add



Integer Add

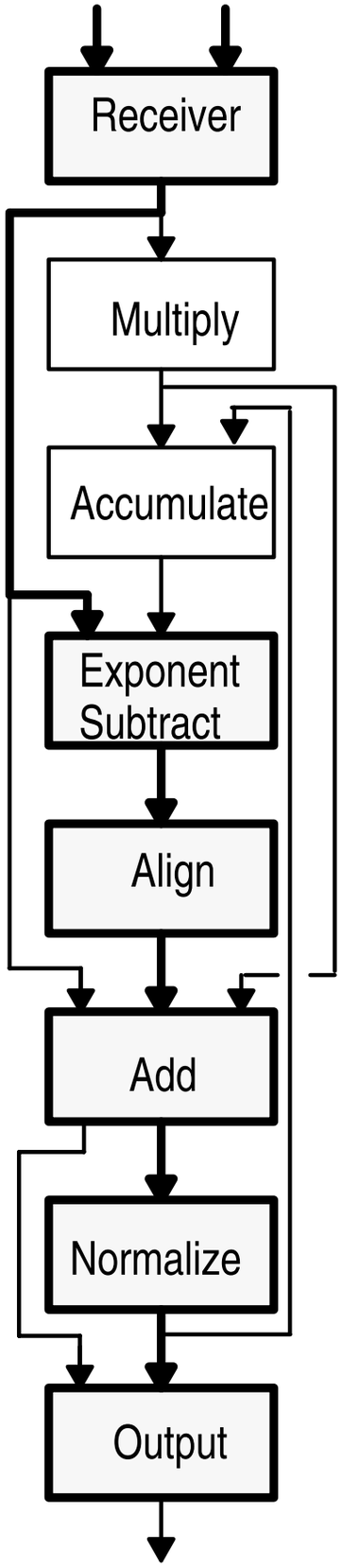


Integer Multiply

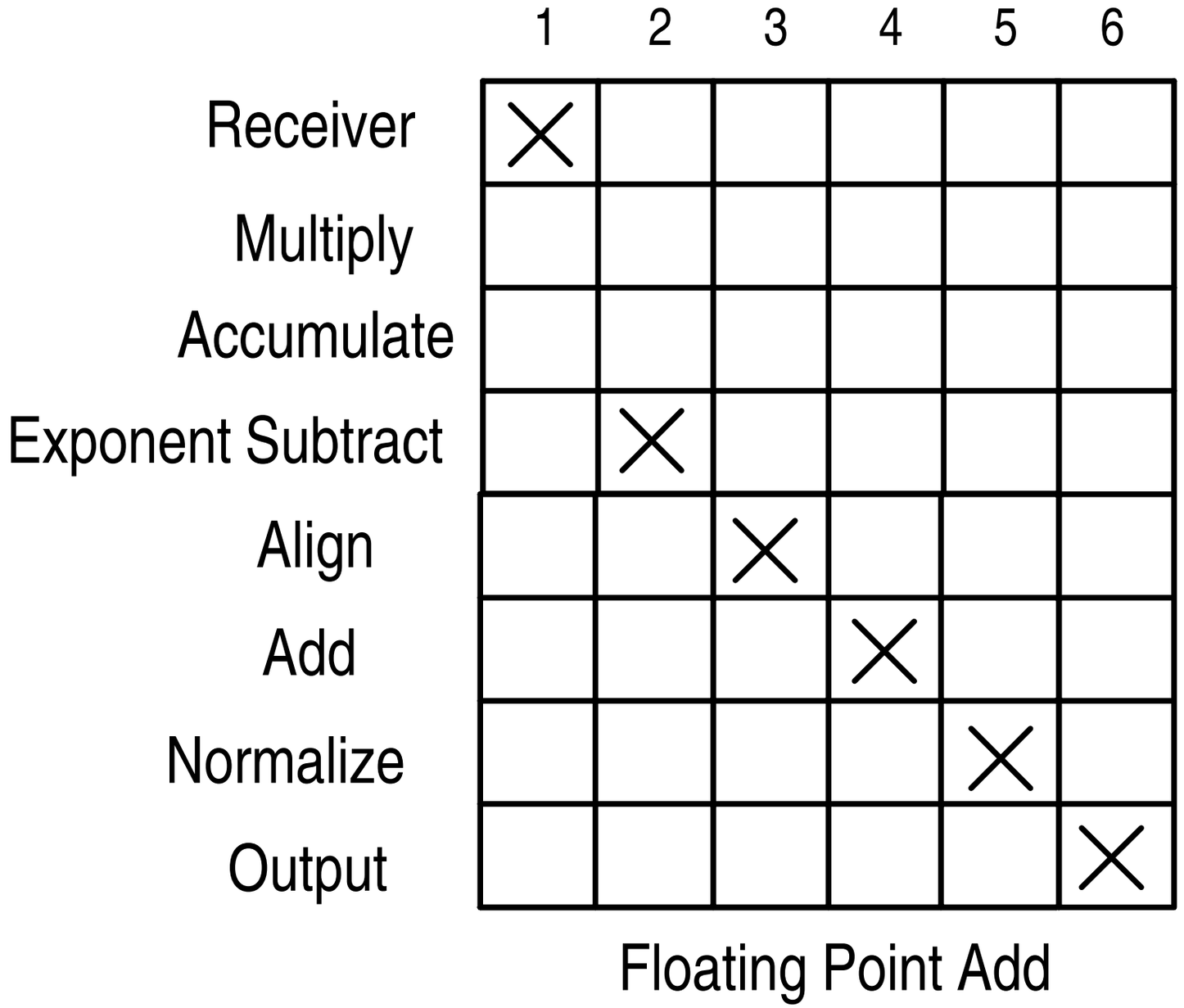


Integer Multiply

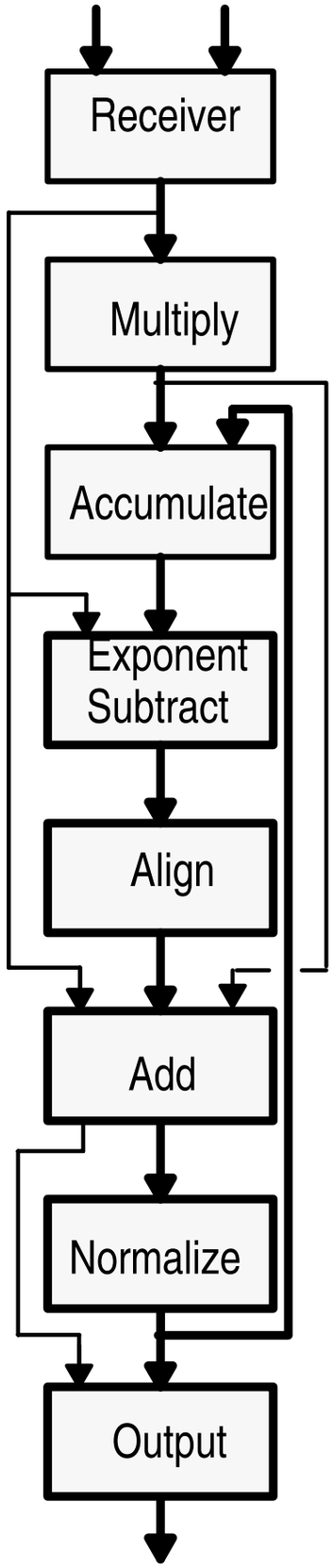
Adds a ze



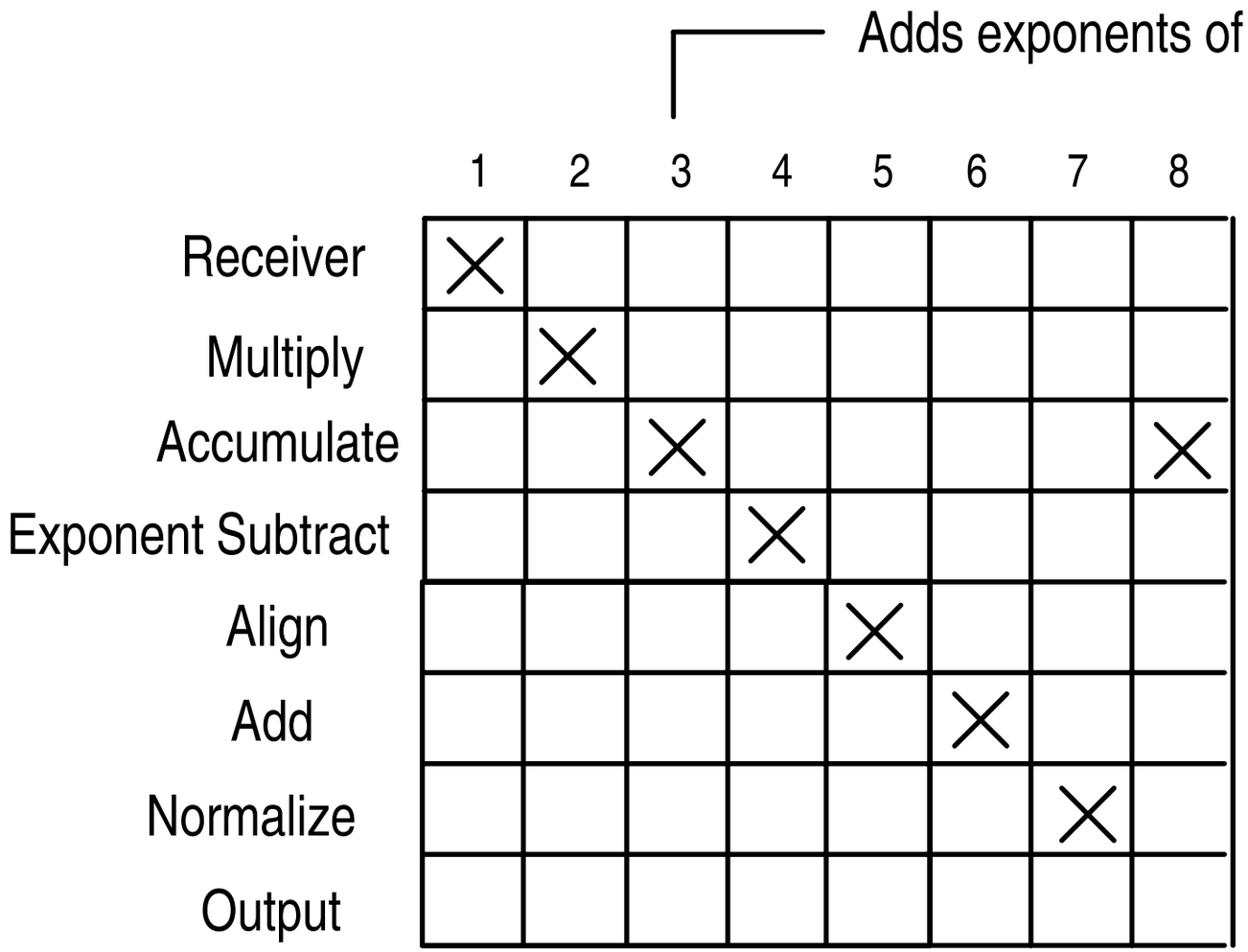
Floating
Point
Add



Floating Point Add



Floating Point Vector Dot Produc



Addition of accumulated values for final result, after inputs are exhausted, is not shown

Vector Inner Product

Multifunction pipelines: Software controlled

- Fixed multiply & add

```
operand_1 * operand_2 + operand_3
```

- Three functions:
  - Multiply and add
  - Multiply: Force operand_3 to be zero
  - Addition: Force operand_1 to be one
- Reservation table same for all three