Operating Systems & C

Lecture 3: Parallelism

Willard Rafnsson
IT University of Copenhagen

ITU CPH

Today: Parallelism

parallel computing: many calculations carried out *simultaneously*

calculations in program often parallelizable.
==> opportunities for speedup.

- pipelining (instance of superscalar)
- vectorization (AVX2)
- multicore (in general)
- heterogeneous computing (CUDA)

ITU CPH

Taxonomy of parallel architectures

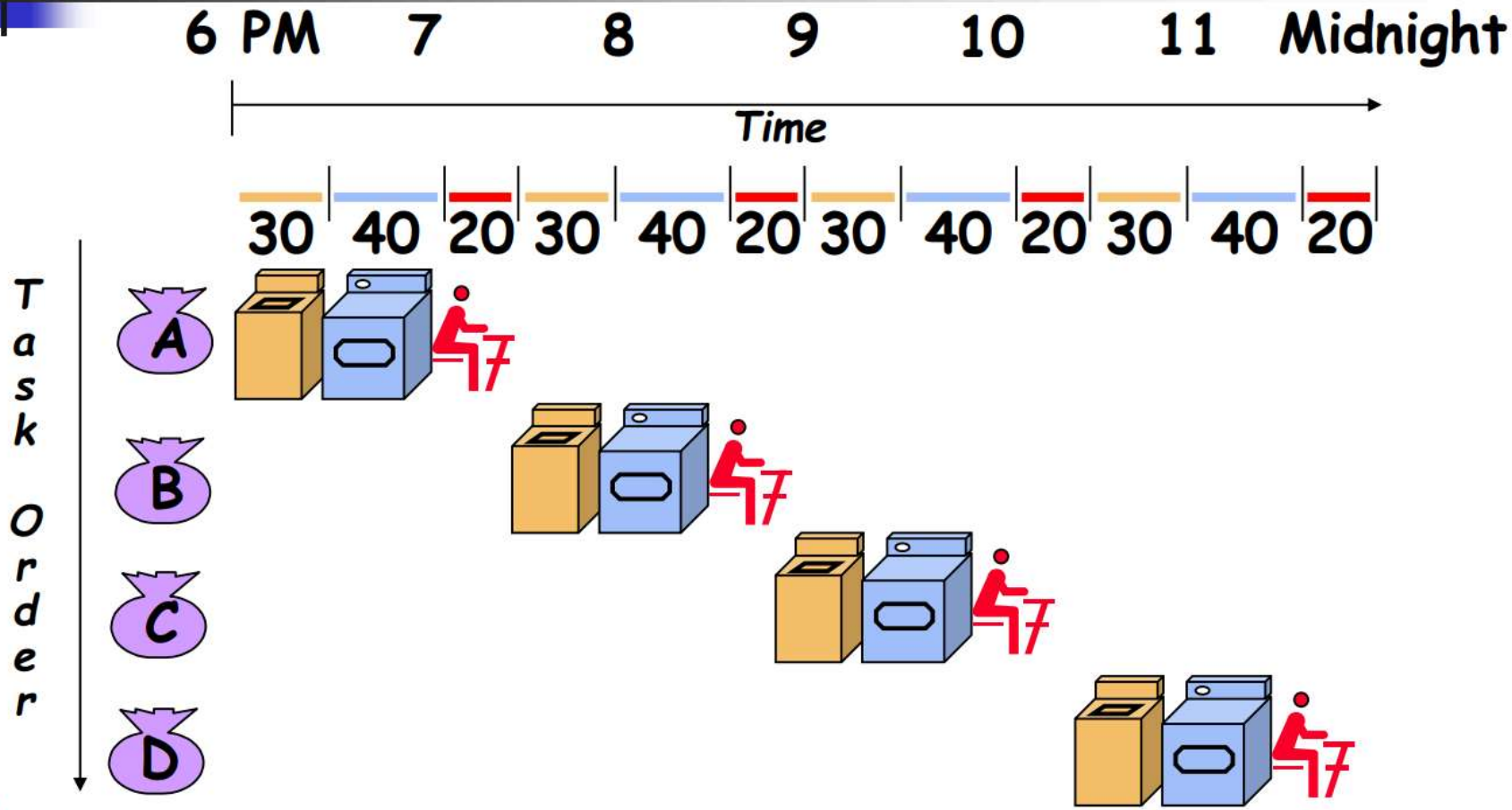
	Horizontal	Vertical
ILP	Superscalar / VLIW	Pipelined
TLP	Multi-core SMT	Interleaved / switch-on-event multithreading
DLP	SIMD / SIMT	Vector / temporal SIMT

ITU CPH

Pipelining

ITU CPH

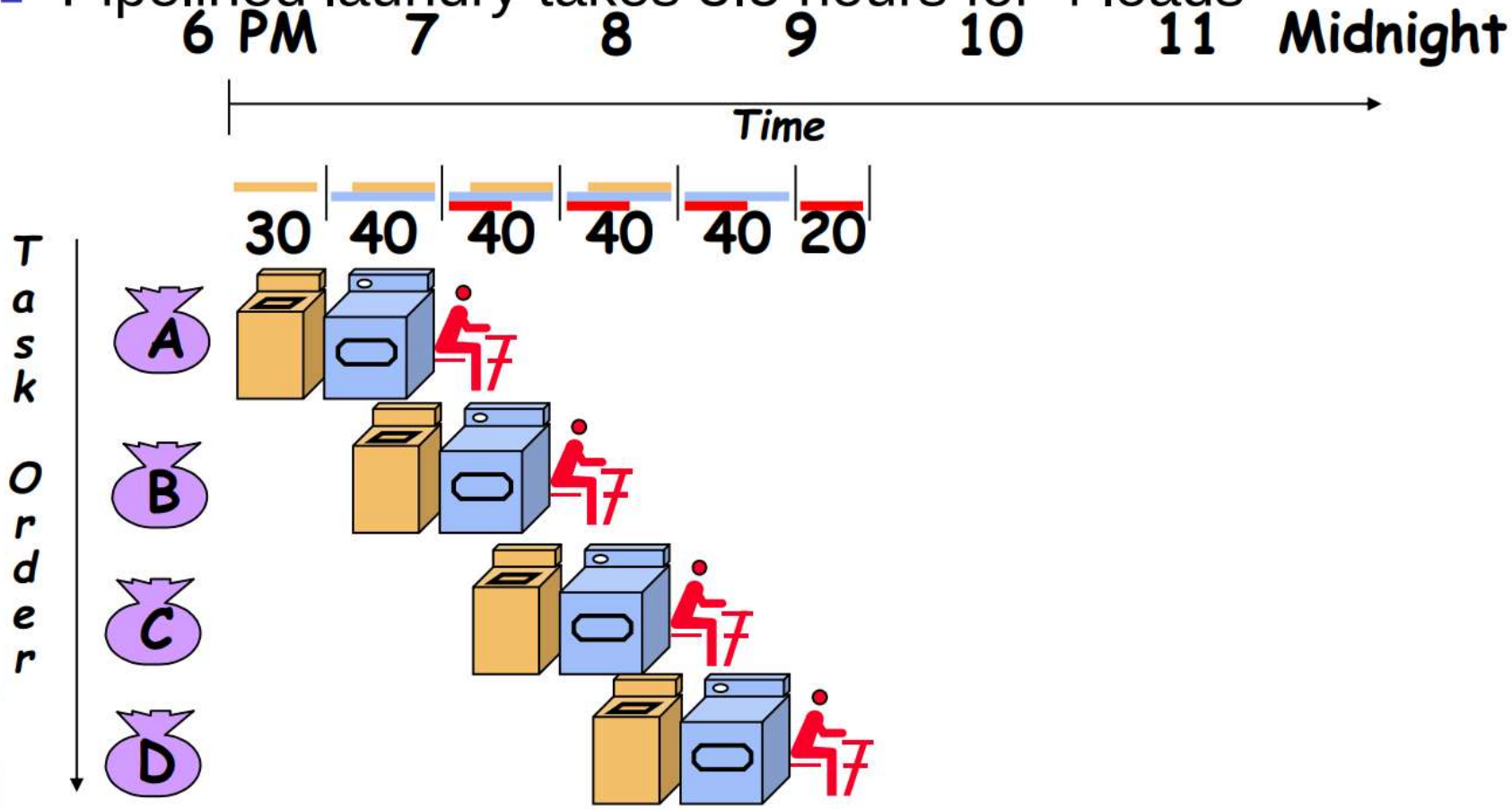
Sequential Laundry



- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry

- Pipelined laundry takes 3.5 hours for 4 loads



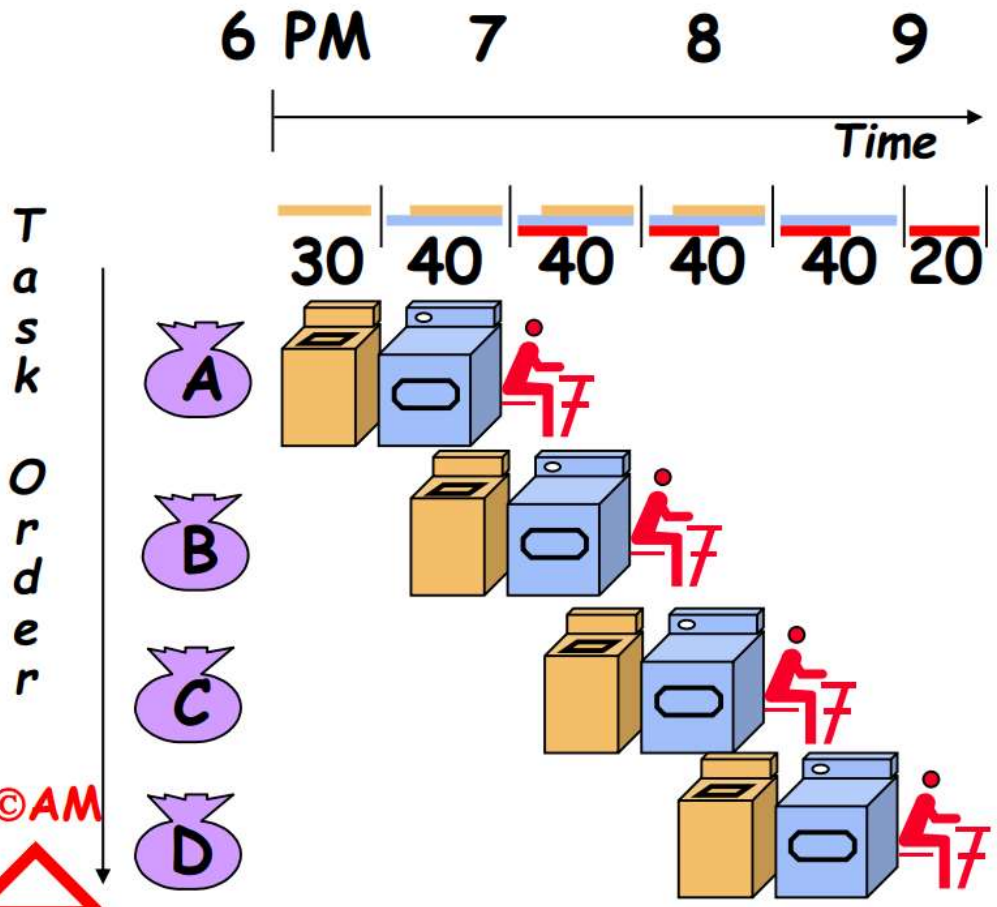
©AM

LaCASA

ITU CPH

Billions of instructions ==> throughput is what matters!

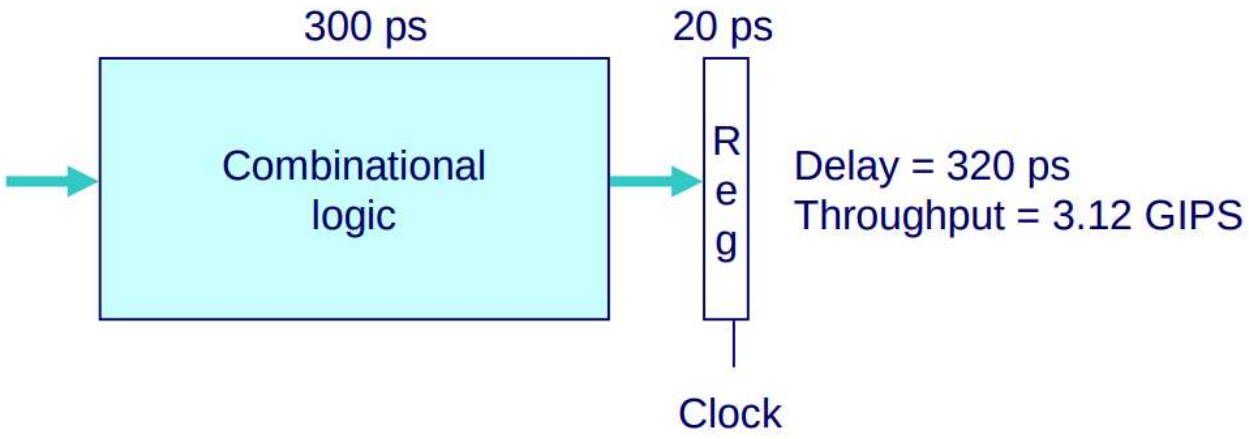
Pipelining Lessons



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

ITU CPH

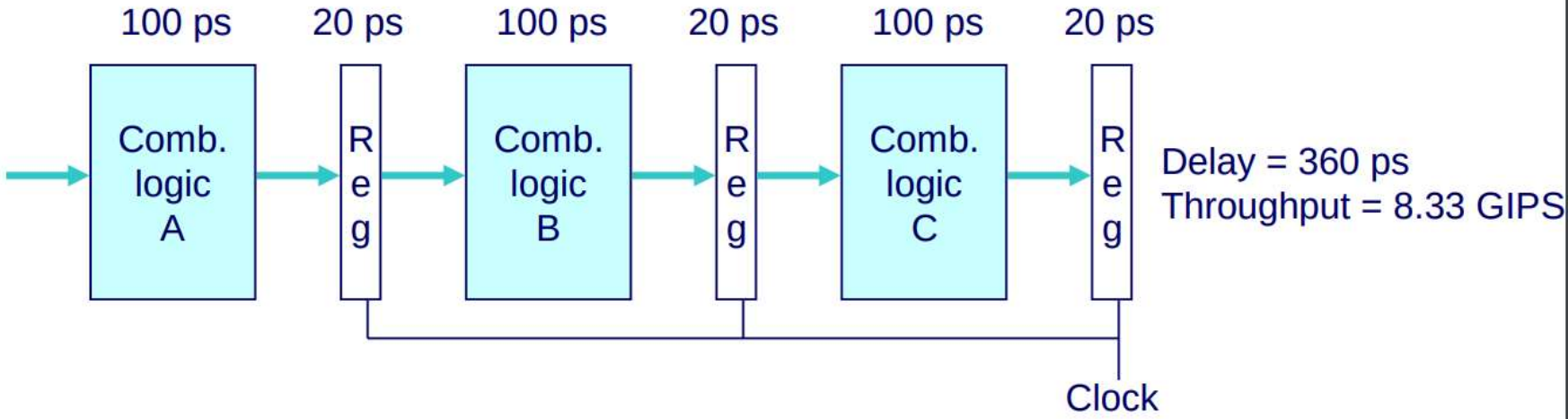
Computational Example



System

- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register
- Must have clock cycle of at least 320 ps

3-Way Pipelined Version

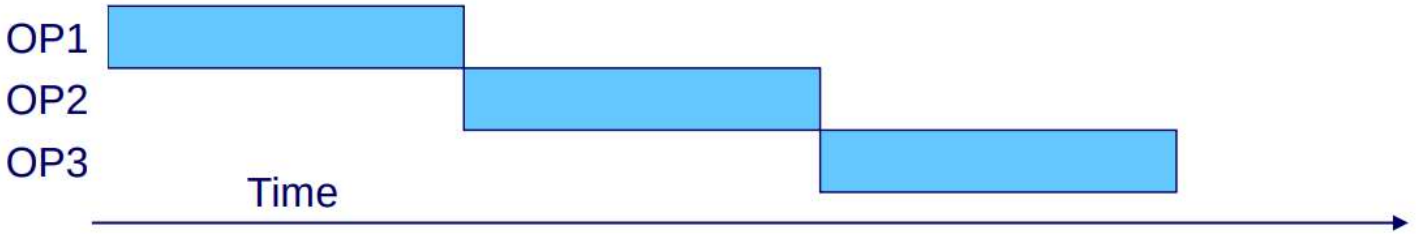


System

- Divide combinational logic into 3 blocks of 100 ps each
- Can begin new operation as soon as previous one passes through stage A.
  - Begin new operation every 120 ps
- Overall latency increases
  - 360 ps from start to finish

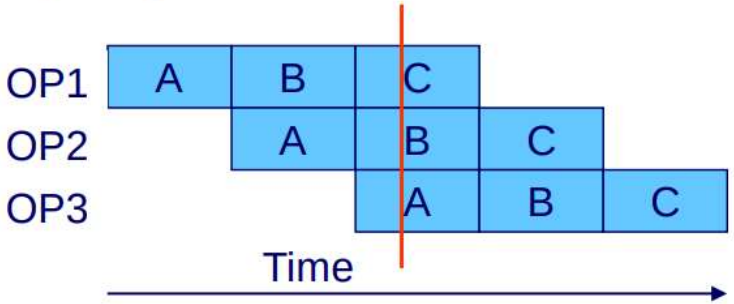
Pipeline Diagrams

Unpipelined



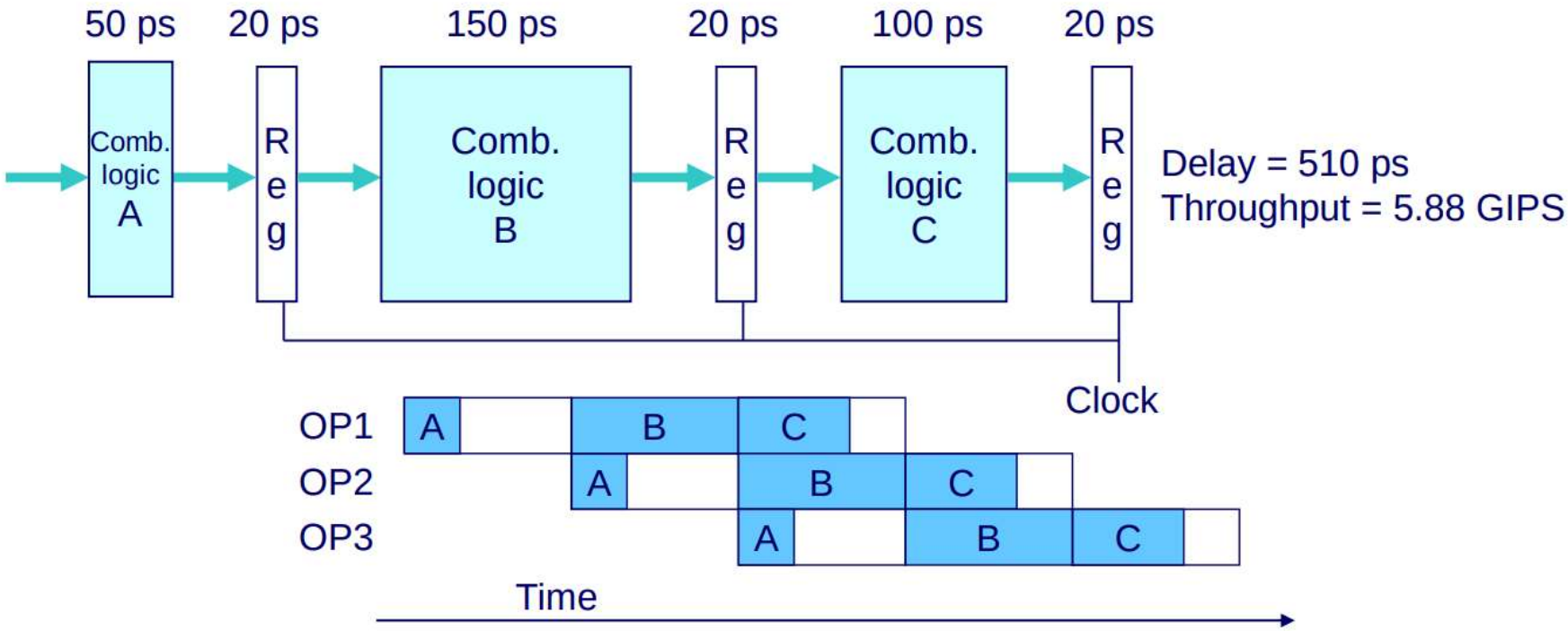
- Cannot start new operation until previous one completes

3-Way Pipelined



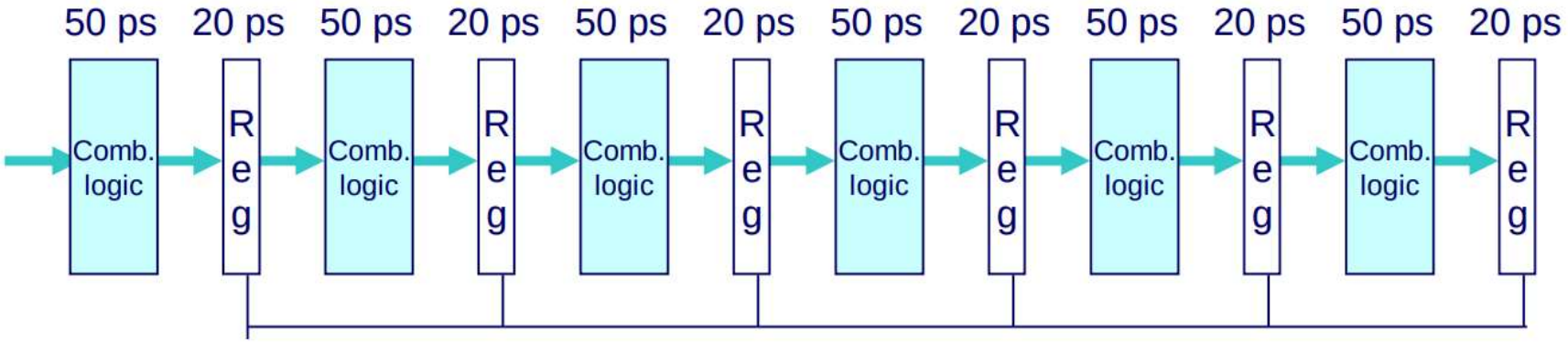
- Up to 3 operations in process simultaneously

Limitations: Nonuniform Delays



- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- Challenging to partition system into balanced stages

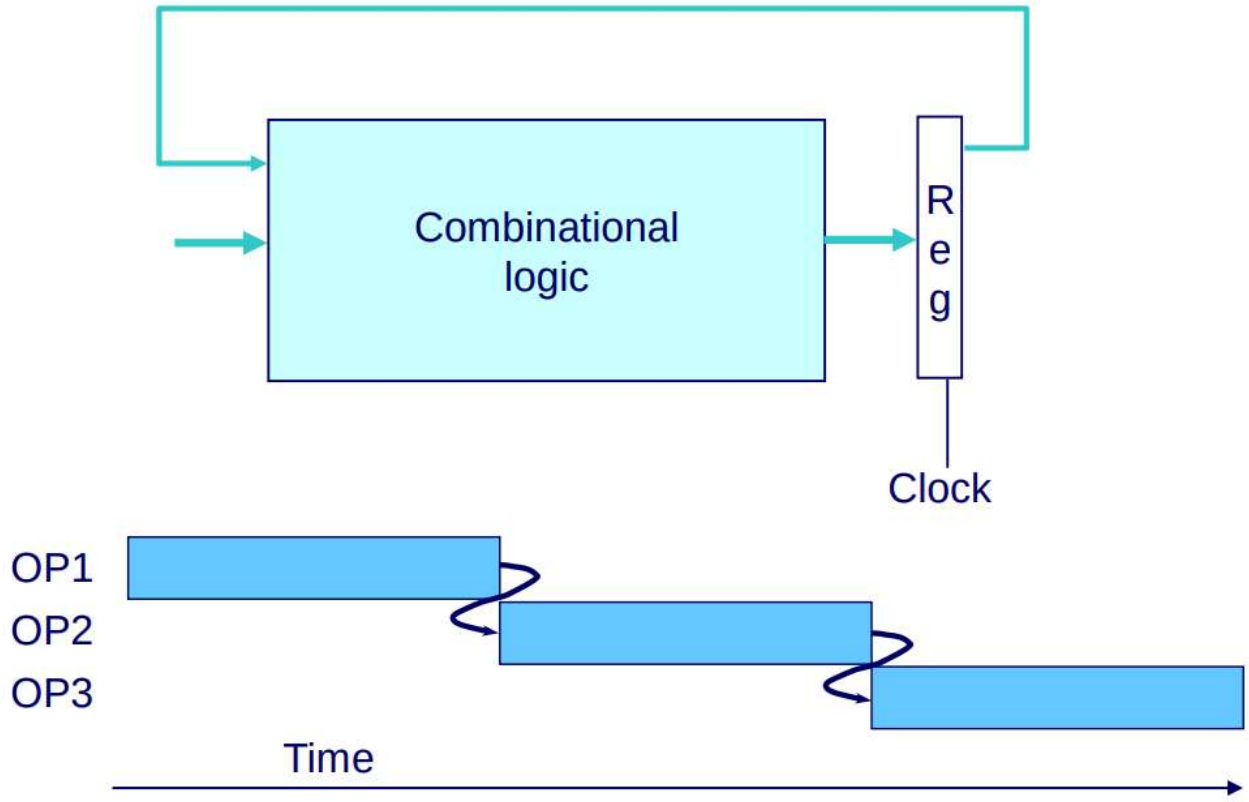
Limitations: Register Overhead



- As try to deepen pipeline, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
  - 1-stage pipeline: 6.25%
  - 3-stage pipeline: 16.67%
  - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through very deep pipelining

Data Dependencies

the inhibitor of parallelization

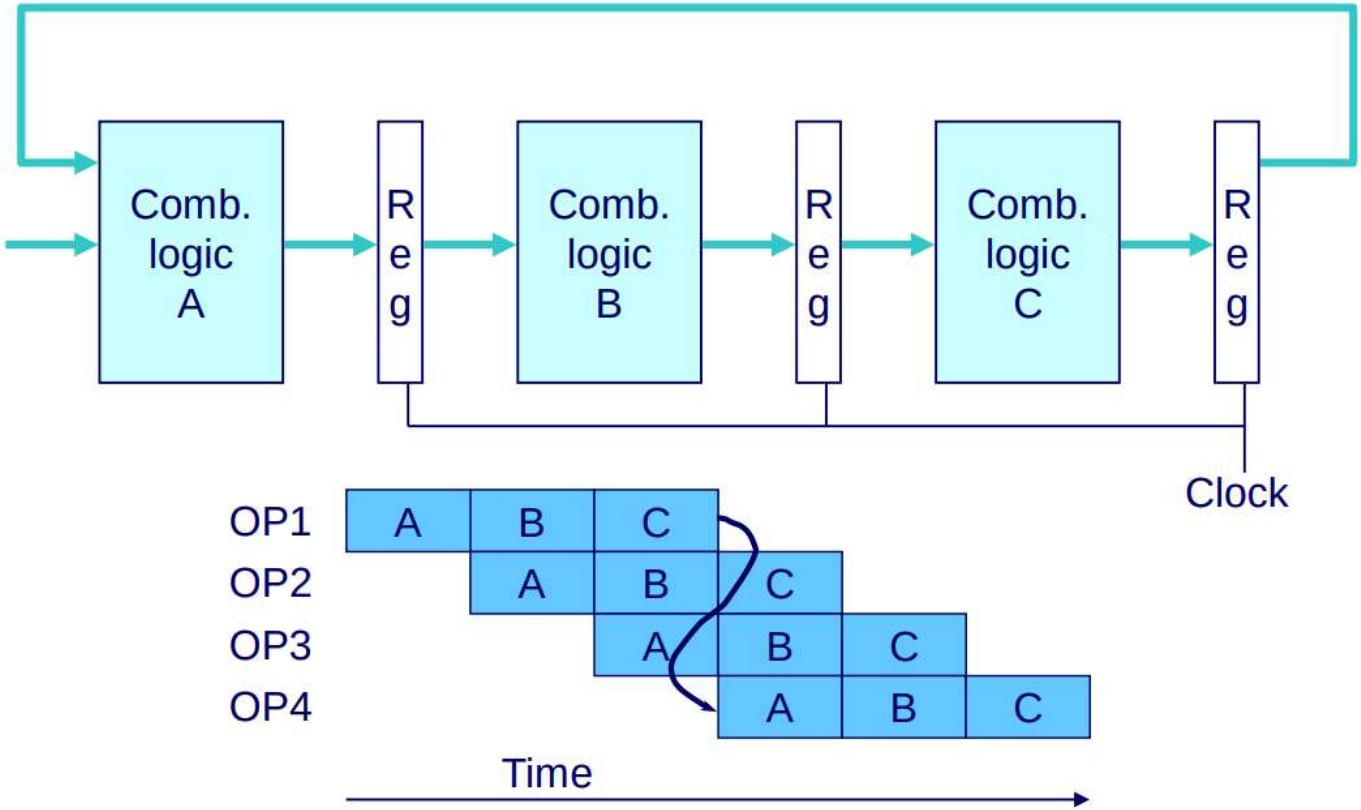


can be mitigated in SW and HW. today: HW

System

- **Each operation depends on result from preceding one**

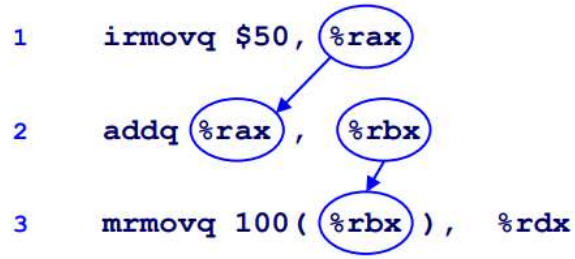
Data Hazards



- **Result does not feed back around in time for next operation**
- **Pipelining has changed behavior of system**

Important criteria:
pipelining ==> "same behavior, just faster".

Data Dependencies in Processors

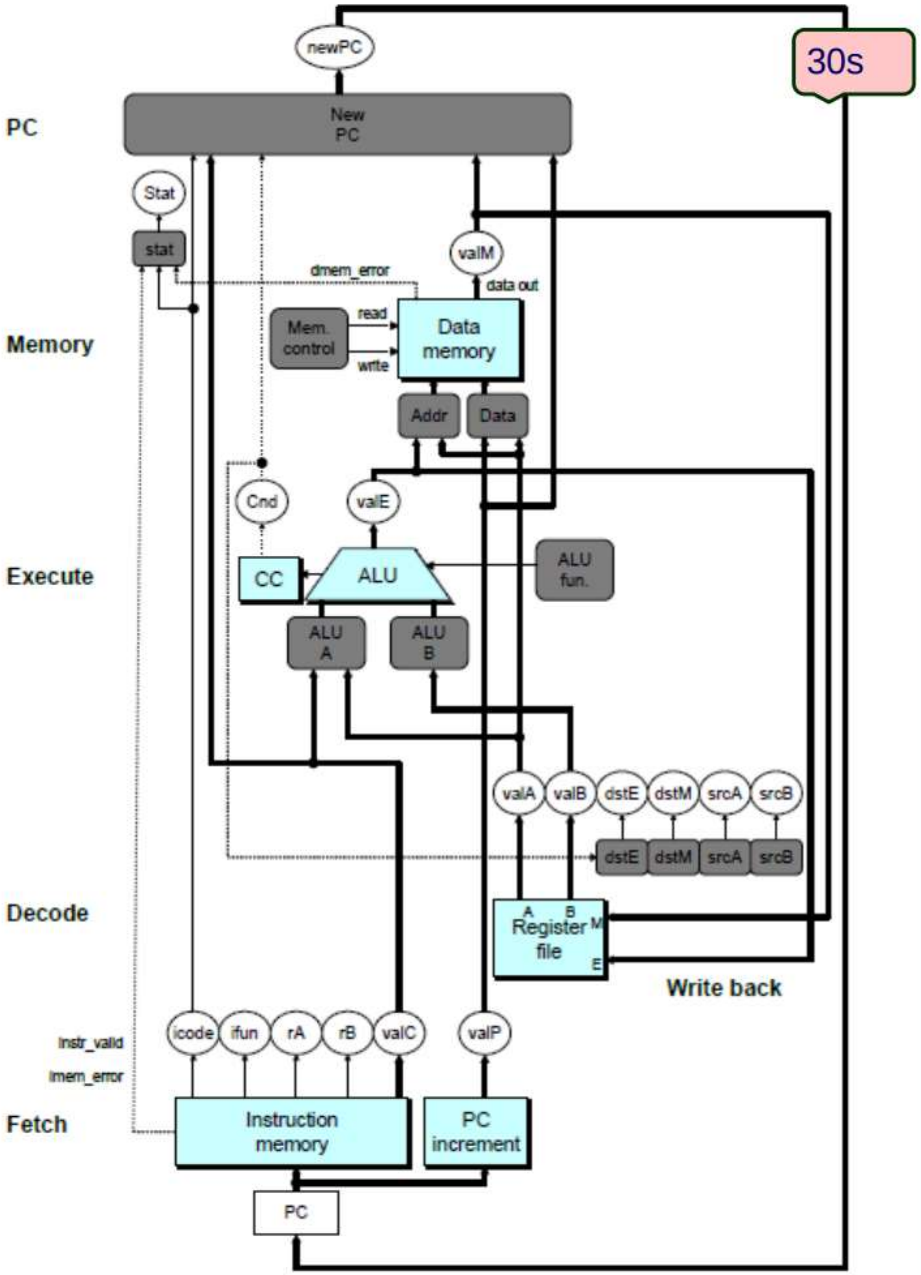


- Result from one instruction used as operand for another
  - Read-after-write (RAW) dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
  - Get correct results
  - Minimize performance impact

recall:

SEQ Hardware

- Stages occur in sequence
- One operation in process at a time



note how this computes what's mentioned in the **semantics**.

Now: let's pipeline this.
issue: **branching** (jump, ret); which instruction is next?
predict next instruction (**speculative execution**).
let's be optimistic for now (address misprediction later)

– 17 –

ITU CPH

15s

SEQ+ Hardware

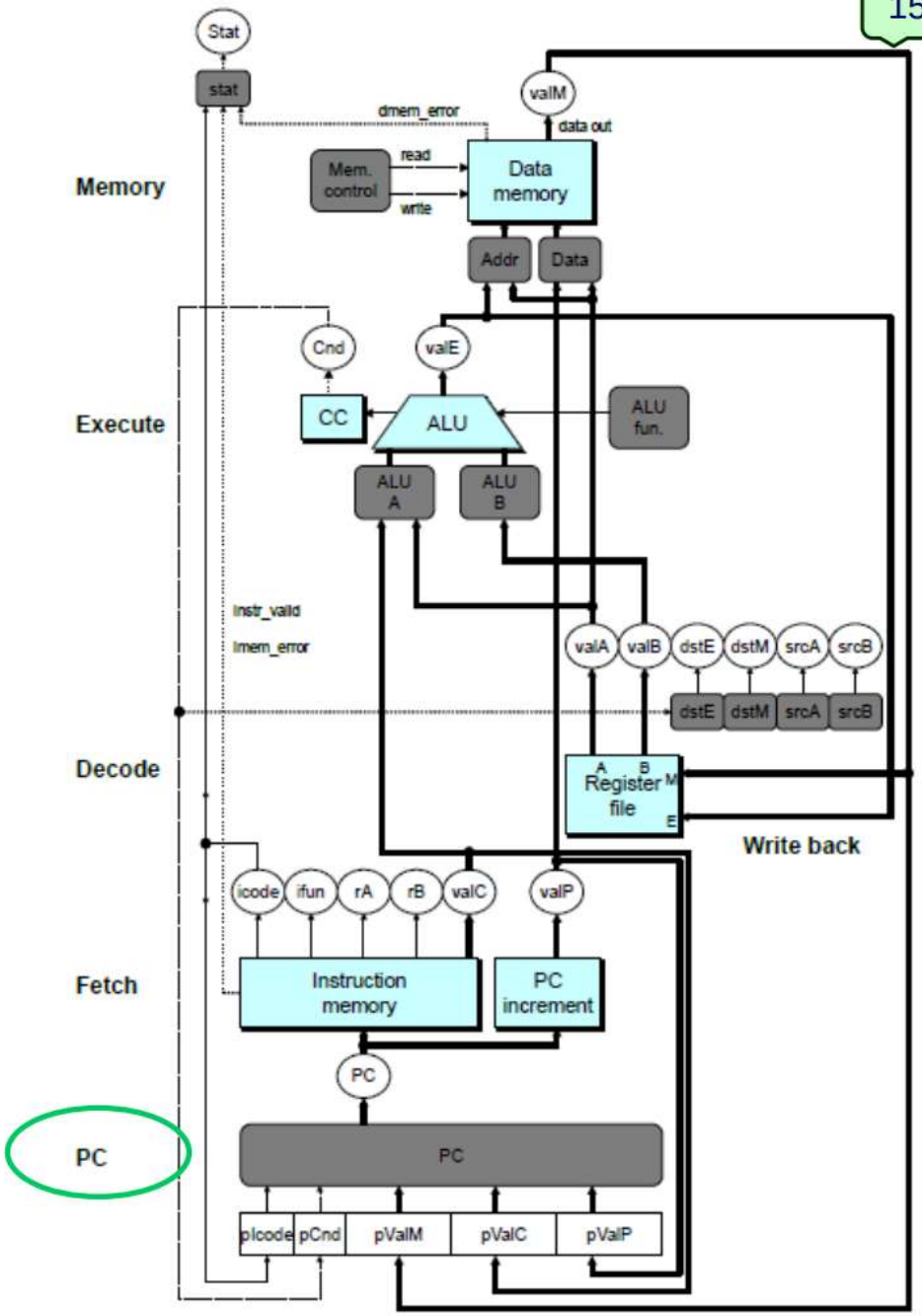
- Still sequential implementation
- Reorder PC stage to put at beginning

PC Stage

- Task is to select PC for current instruction
- Based on results computed by previous instruction

Processor State

- PC is no longer stored in register
- But, can determine PC based on other stored information



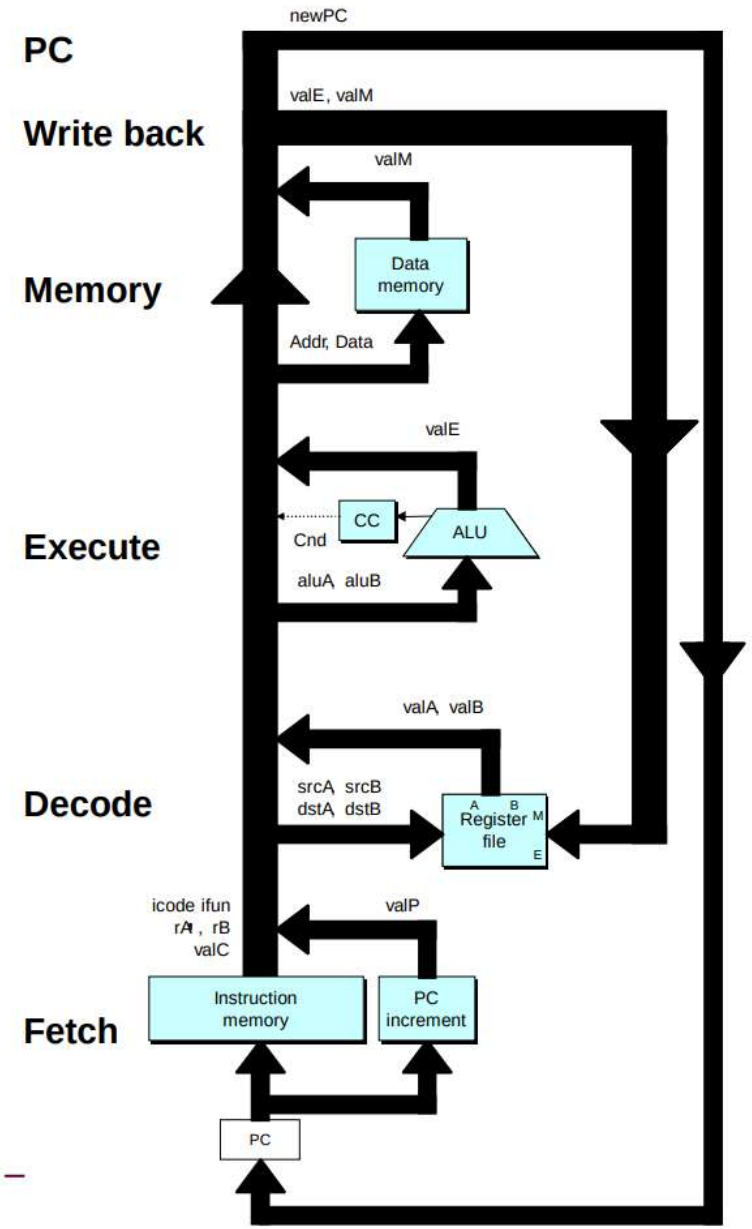
– 18 –

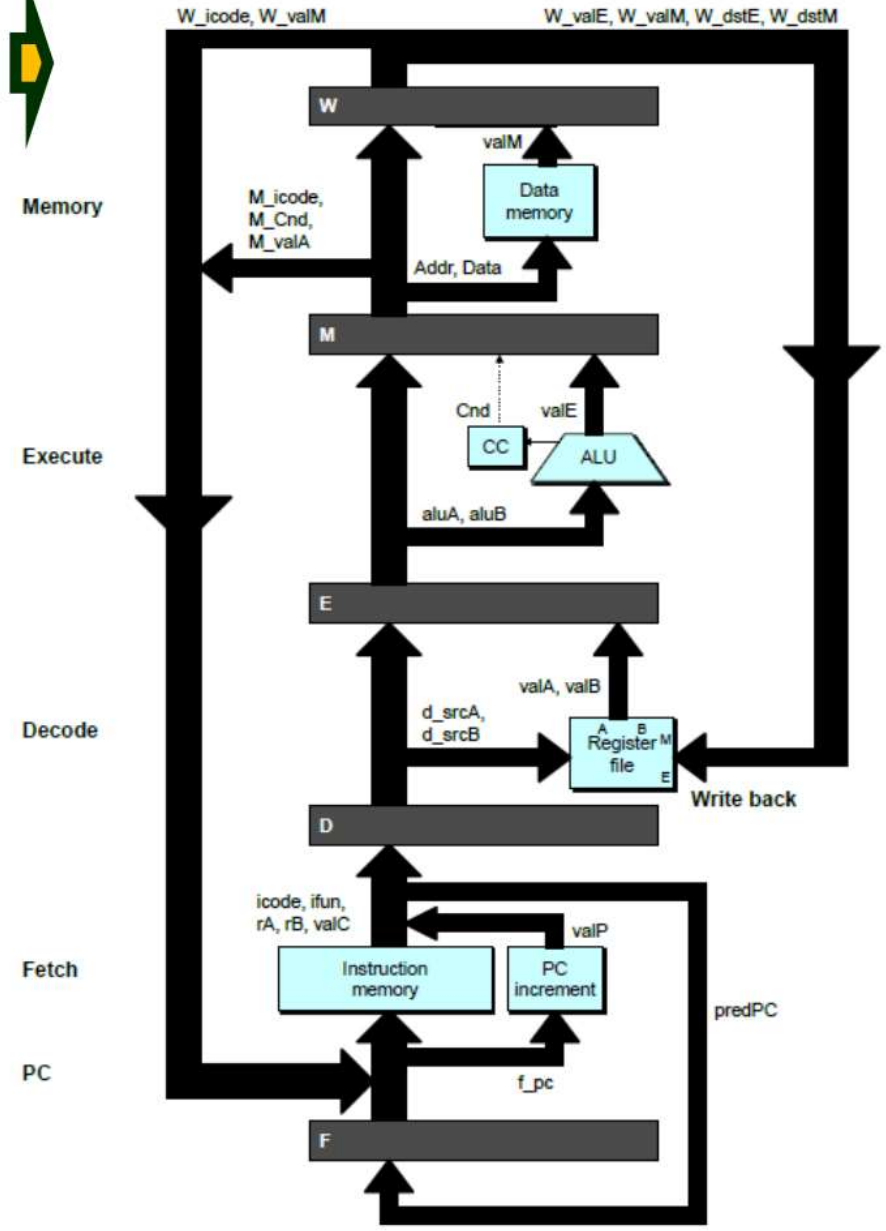
ITU CPH

Adding Pipeline Registers

at each stage

15s





– 19 –

ITU CPH

15s

Pipeline Stages

Fetch

- Select current PC
- Read instruction
- Compute incremented PC

Decode

- Read program registers

Execute

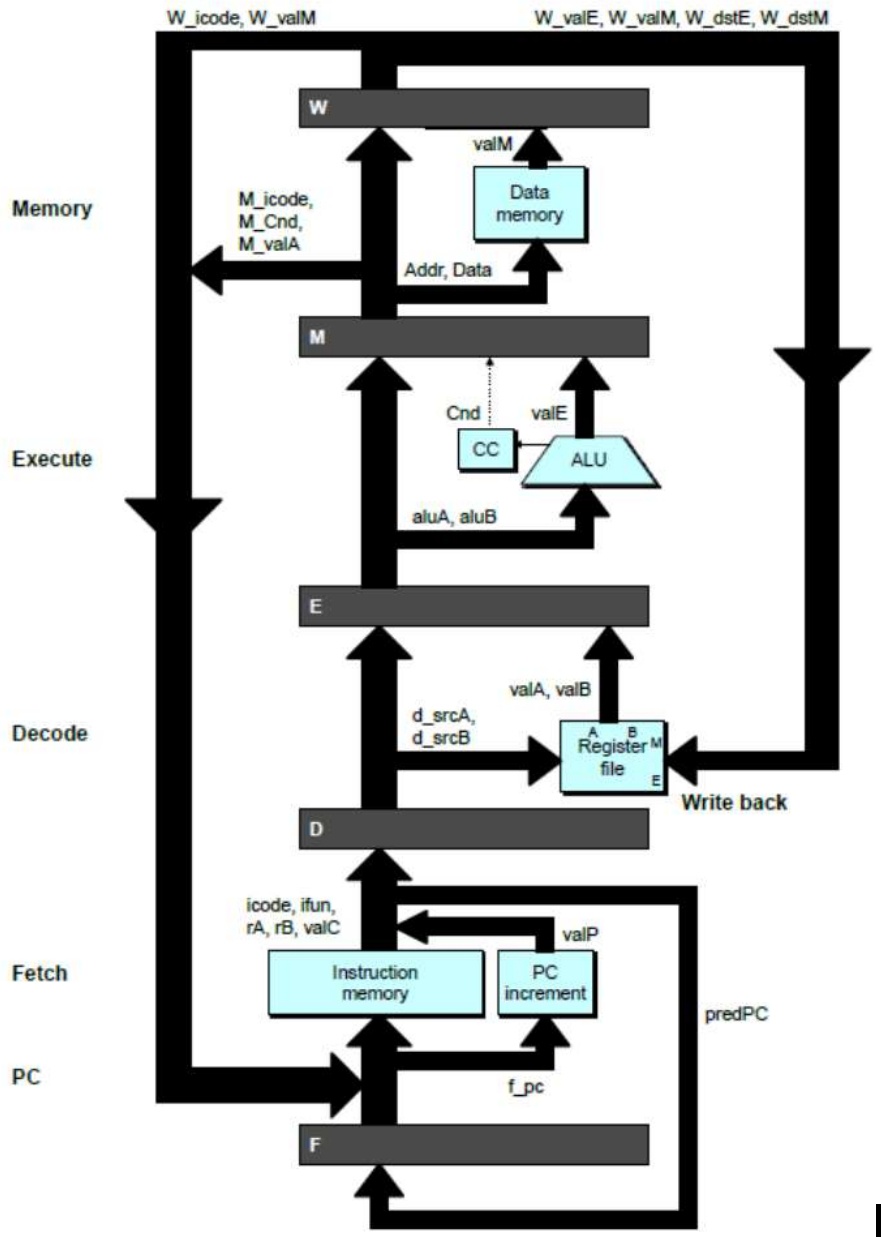
- Operate ALU

Memory

- Read or write data memory

Write Back

- Update register file



ITU CPH

10s

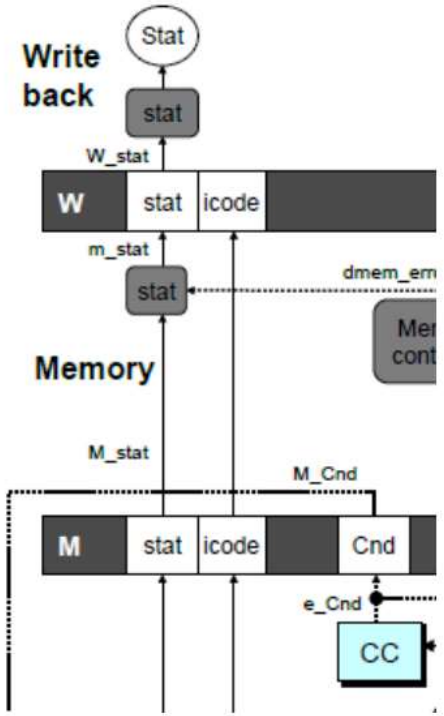
Signal Naming Conventions

S_Field

- Value of Field
  held in stage S pipeline register

s_Field

- Value of Field
  computed in stage S



sharp edge: store
rounded edge: logic

in its glory (w/ combinational logic drawn):

15s

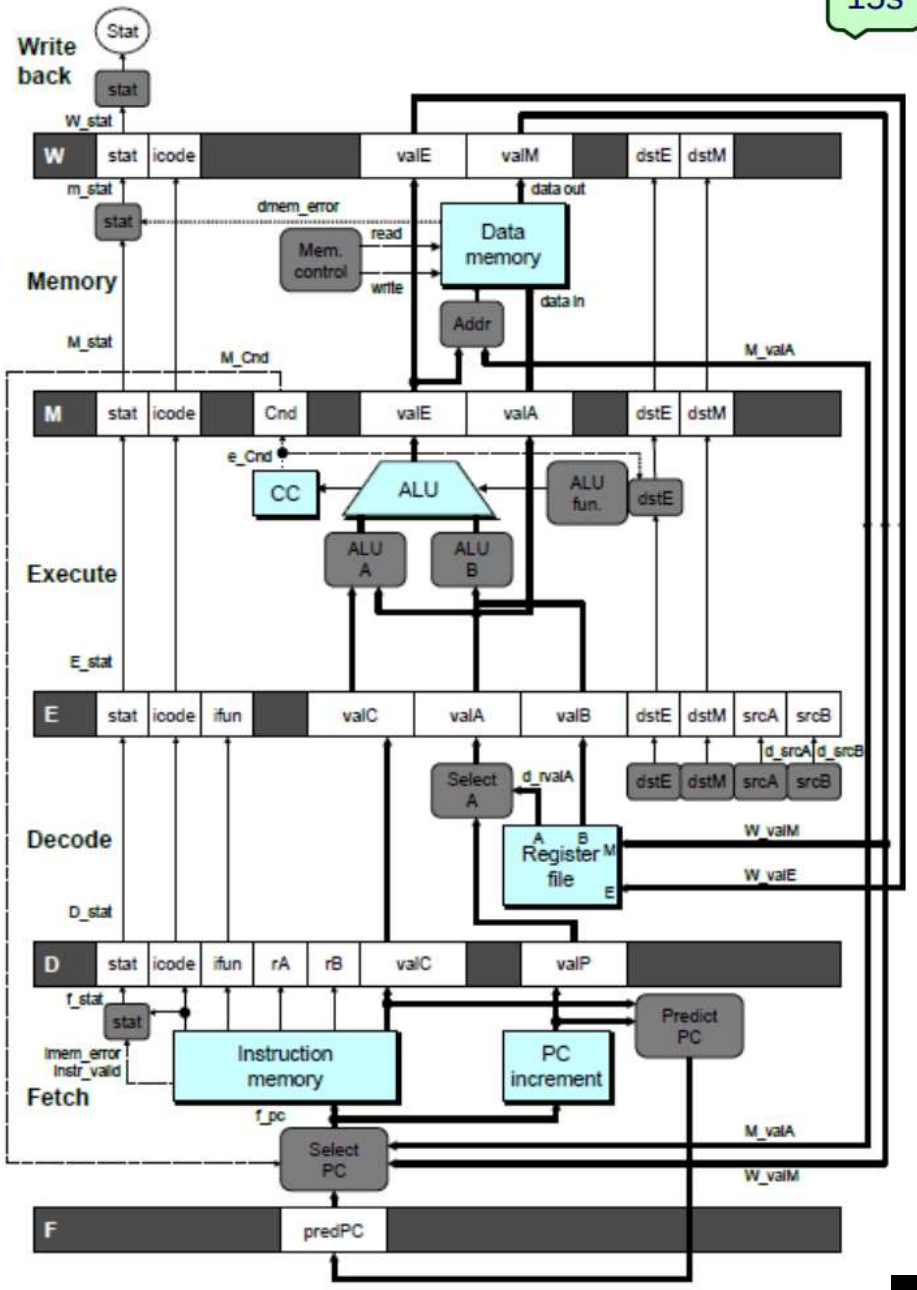
PIPE- Hardware

- Pipeline registers hold intermediate values from instruction execution

Forward (Upward) Paths

- Values passed from one stage to next
- Cannot jump past stages
  - e.g., valC passes through decode

behaves as explained in the previous slide



ITU CPH

highlight:

Feedback Paths

Predicted PC

- Guess value of next PC

Branch information

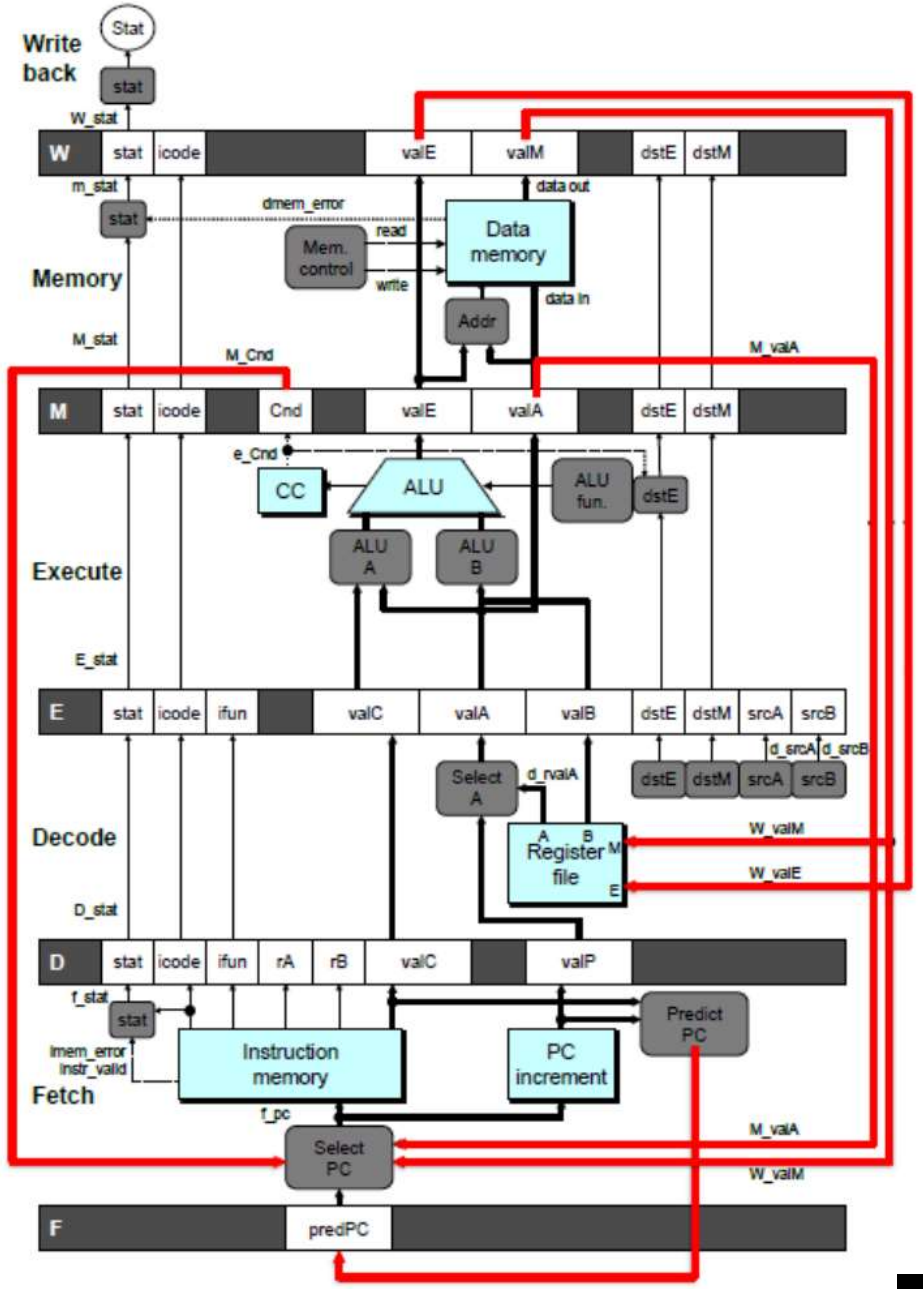
- Jump taken/not-taken
- Fall-through or target address

Return point

- Read from memory

Register updates

- To register file write ports

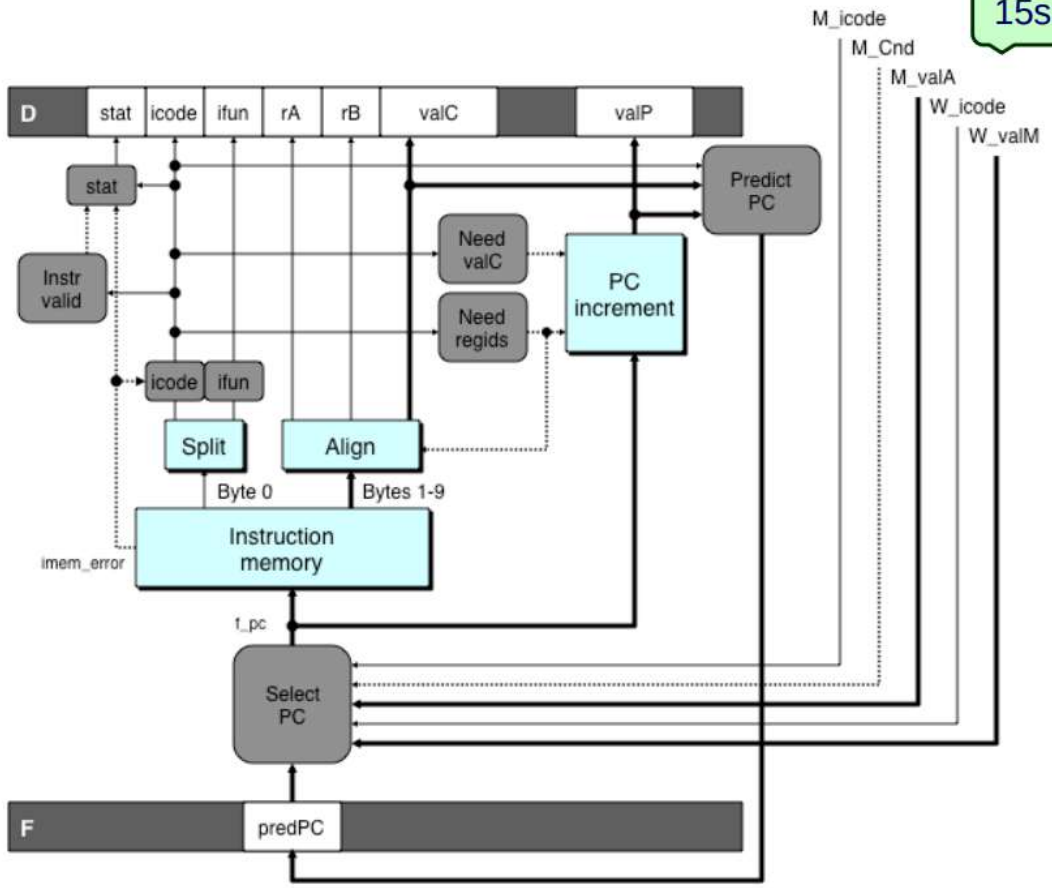


ITU CPH

Predicting the PC

15s

detailed view of the fetch-logic



- Start fetch of new instruction after current one has completed fetch stage
  - Not enough time to reliably determine next instruction
- Guess which instruction will follow
  - Recover if prediction was incorrect

remainder of PIPE slides are about this **branch prediction** & recovery.

– 23 –

ITU CPH

Our Prediction Strategy

Instructions that Don't Transfer Control

- Predict next PC to be valP (increment)
- Always reliable

remember semantics

Call and Unconditional Jumps

- Predict next PC to be valC (destination)
- Always reliable

remember semantics

Conditional Jumps

- Predict next PC to be valC (destination)
- Only correct if branch is taken
  - Typically right 60% of time

cond. assumed **true**

Return Instruction

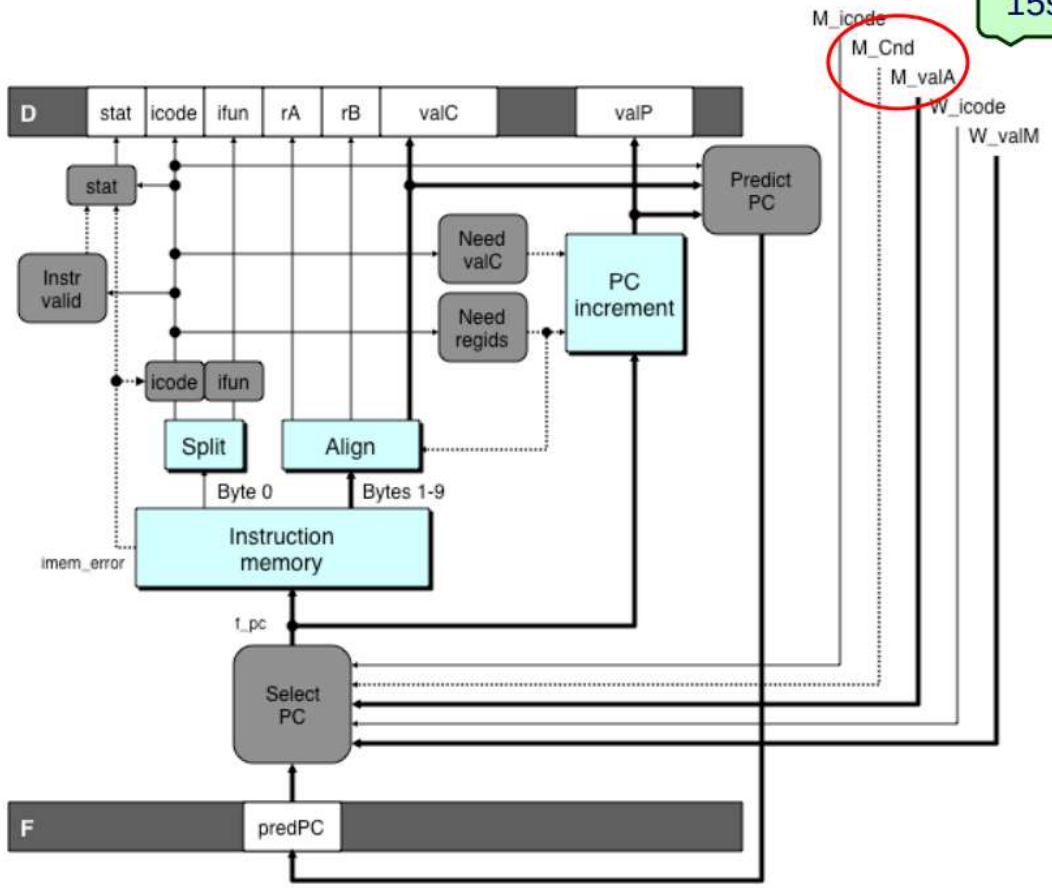
- Don't try to predict

have nothing to go on in our registers; target addr. is on the stack

recovery? (later)

Recovering from PC Misprediction

15s



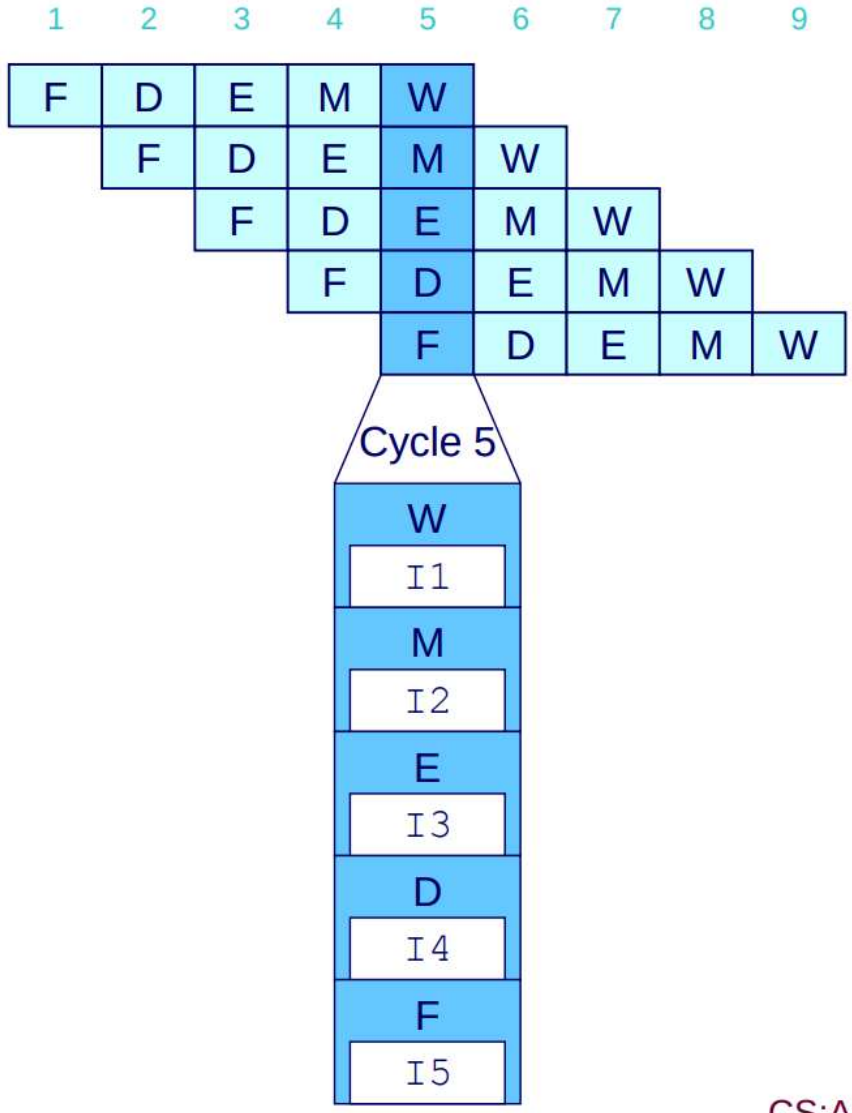
- **Mispredicted Jump**
  - Will see branch condition flag once instruction reaches M stage
  - Can get fall-through PC from valA (value M_valA)
- **Return Instruction**
  - Will get return PC when `ret` reaches write-back (W) stage (W_valM)

should give some idea of cost of misprediction.

15s

Pipeline Demonstration

```
irmovq    $1,%rax   #I1
irmovq    $2,%rcx   #I2
irmovq    $3,%rdx   #I3
irmovq    $4,%rbx   #I4
halt                #I5
```



File: `demo-basic.ys`

(all is well)

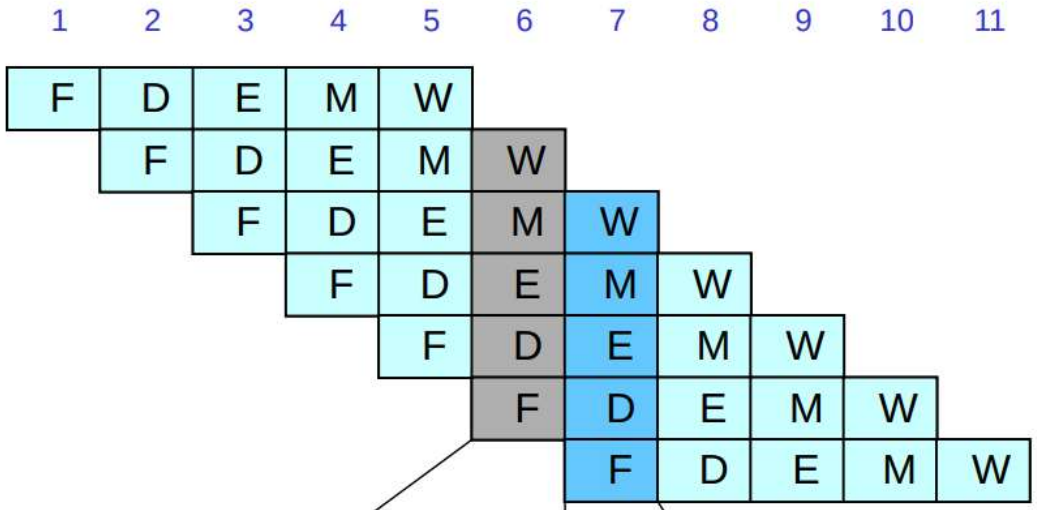
CS:APP3e

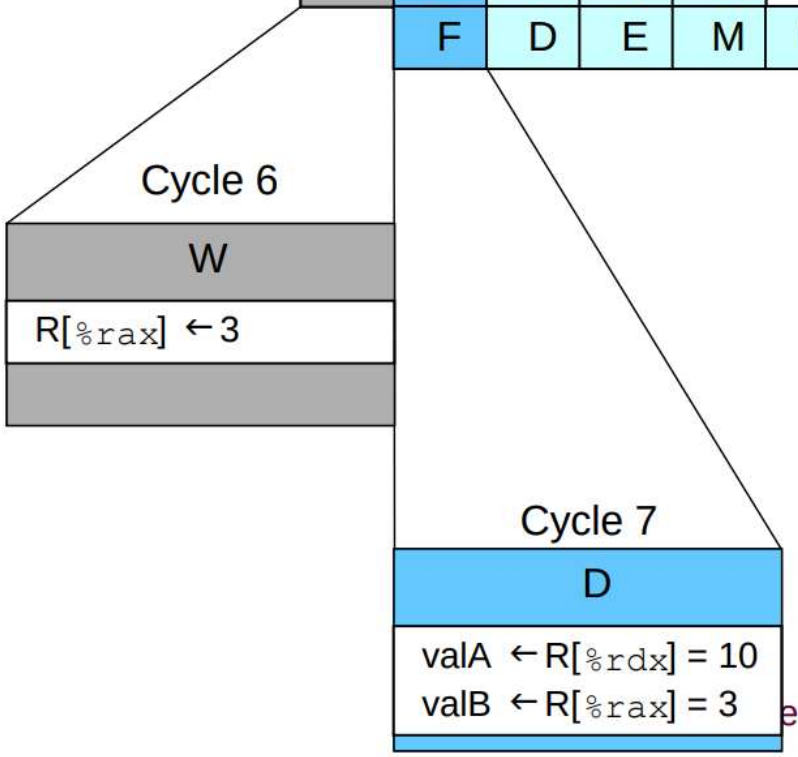
ITU CPH

Data Dependencies: 3 Nop's

30s

```
# demo-h3.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: nop
0x017: addq %rdx,%rax
0x019: halt
```





instance where pipelining doesn't help speedup much, due to **data dependency** (nops needed, or else...)

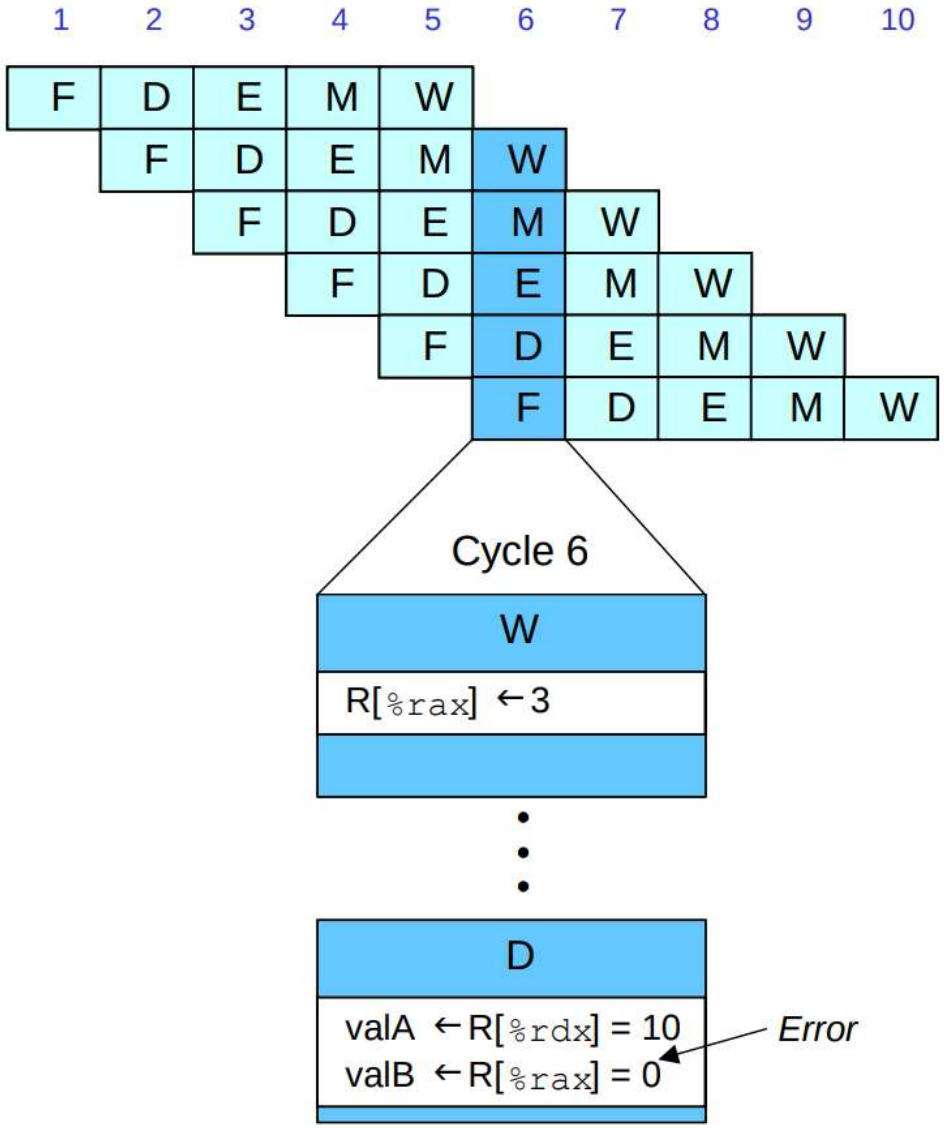
– 27 –

ITU CPH

Data Dependencies: 2 Nop's

15s

```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



`rax` writeback hasn't finished; instruction depends on previous instruction still in pipeline.
DATA HAZARD

– 28 –

CS:APP3e

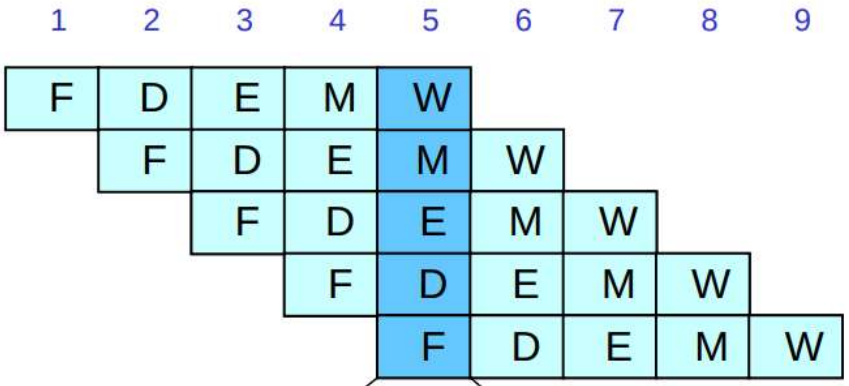
ITU CPH

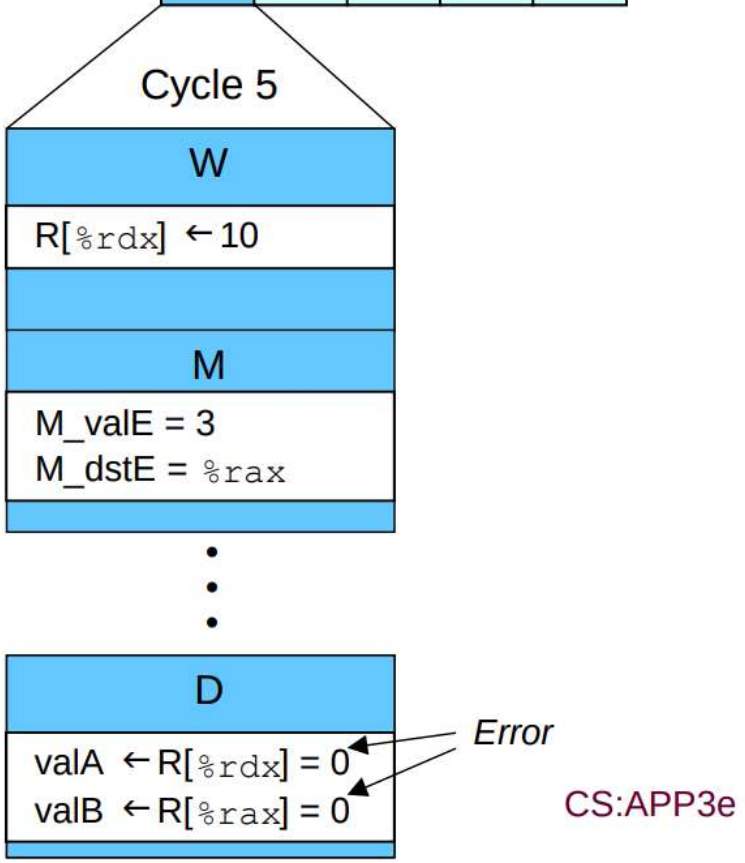
Data Dependencies: 1 Nop

5s

```
# demo-h1.ys

0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: addq %rdx,%rax
0x017: halt
```





rdx writeback hasn't finished, rax assignment hasn't even executed

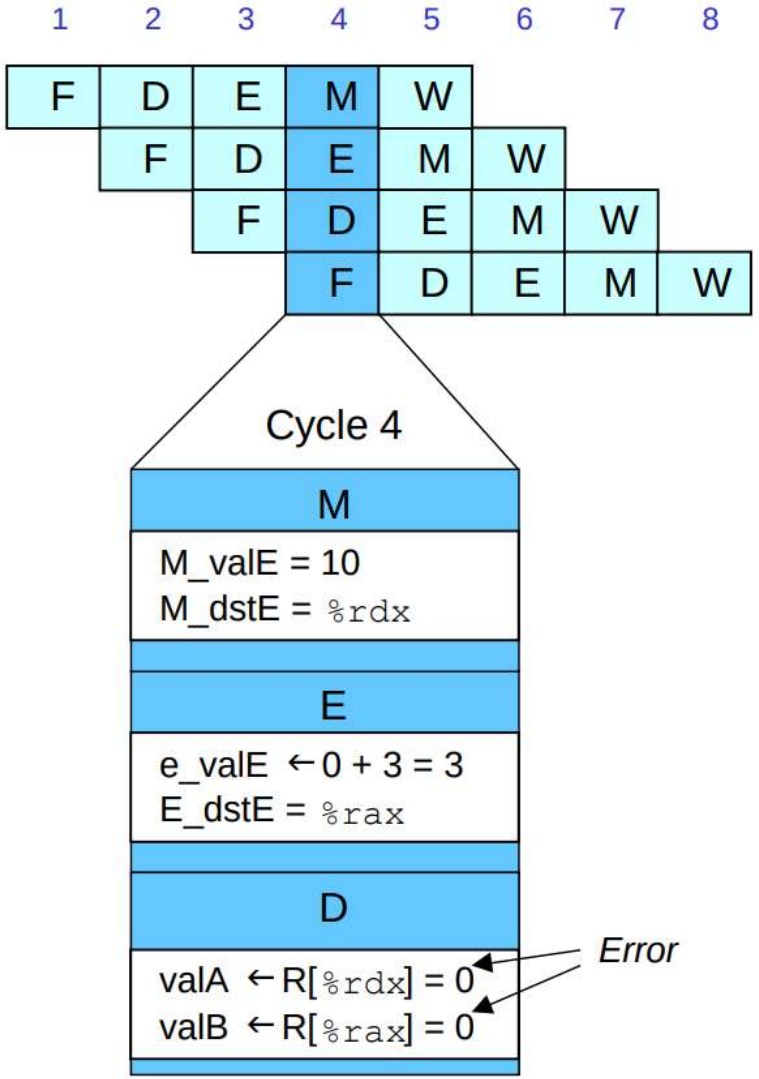
CS:APP3e

ITU CPH

Data Dependencies: No Nop

5s

```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



Branch Misprediction Example

demo-j.ys

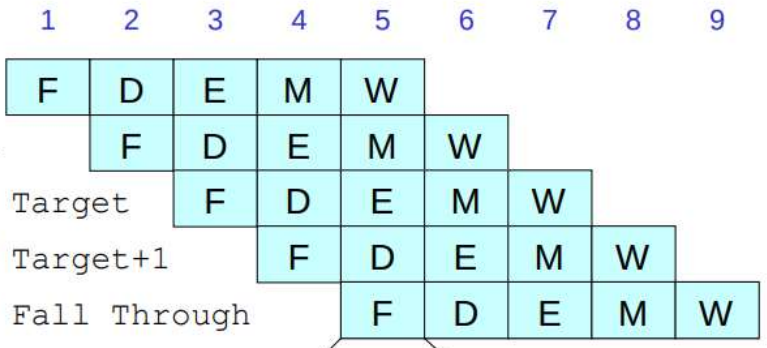
```
0x000:     xorq %rax,%rax
0x002:     jne  t                # Not taken
0x00b:     irmovq $1, %rax       # Fall through
0x015:     nop
0x016:     nop
0x017:     nop
0x018:     halt
0x019: t:  irmovq $3, %rdx       # Target (Should not execute)
0x023:     irmovq $4, %rcx       # Should not execute
0x02d:     irmovq $5, %rdx       # Should not execute
```

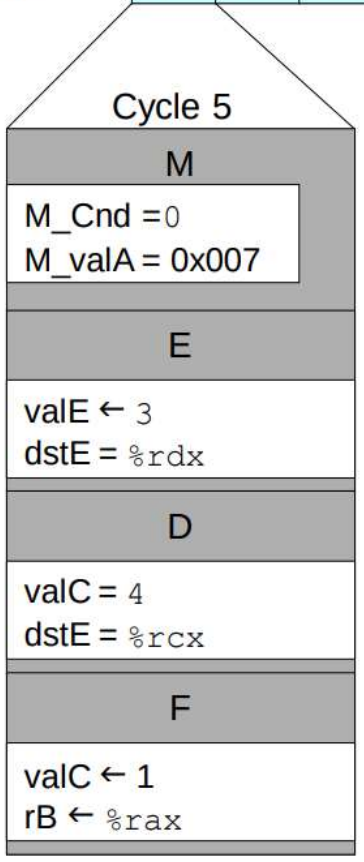
- **Should only execute first 8 instructions**

another instance where pipelining doesn't help speedup: **branch misprediction** (jump)

Branch Misprediction Trace

```
# demo-j
0x000:     xorq %rax,%rax
0x002:     jne t # Not taken
0x019: t:  irmovq $3, %rdx # Target
0x023:     irmovq $4, %rcx # Target+1
0x00b:     irmovq $1, %rax # Fall Through
```





- **Incorrectly execute two instructions at branch target**

CONTROL HAZARD

CS:APP3e

ITU CPH

Return Example

demo-ret.ys

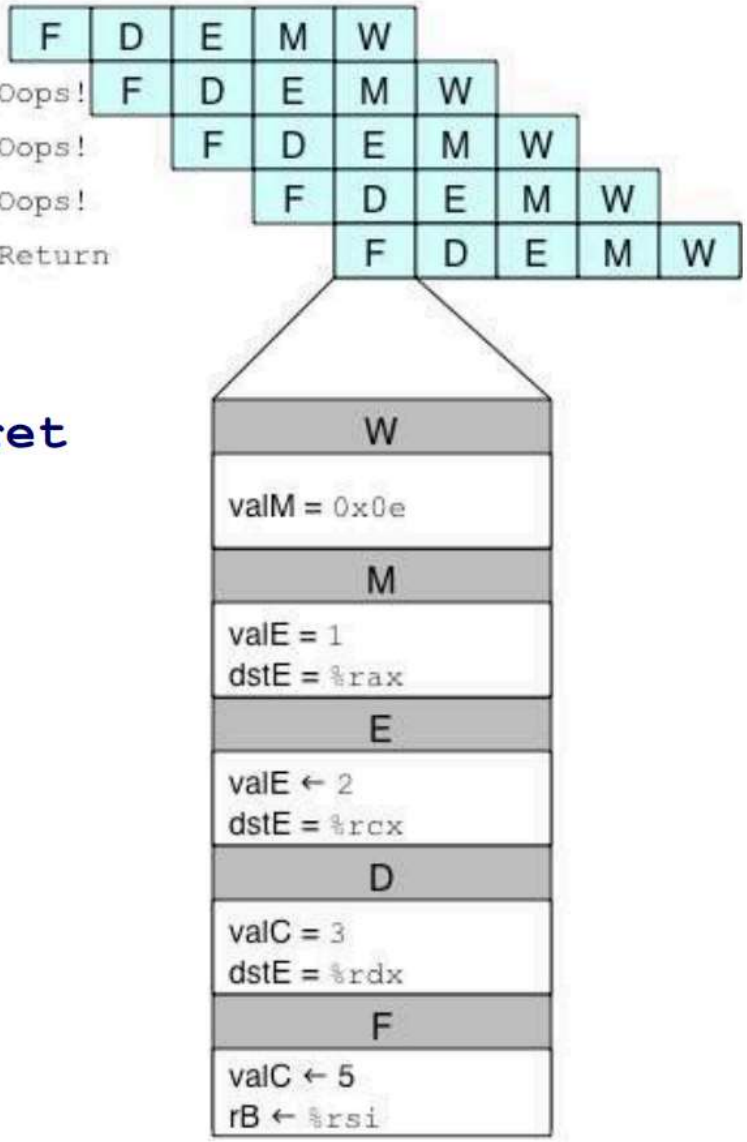
```
0x000:    irmovq Stack,%rsp    # Intialize stack pointer
0x00a:    nop                  # Avoid hazard on %rsp
0x00b:    nop
0x00c:    nop
0x00d:    call p               # Procedure call
0x016:    irmovq $5,%rsi       # Return point
0x020:    halt
0x020: .pos 0x20
0x020: p: nop                  # procedure
0x021:    nop
0x022:    nop
0x023:    ret
0x024:    irmovq $1,%rax       # Should not be executed
0x02e:    irmovq $2,%rcx       # Should not be executed
0x038:    irmovq $3,%rdx       # Should not be executed
0x042:    irmovq $4,%rbx       # Should not be executed
0x100: .pos 0x100
0x100: Stack:                  # Initial stack pointer
```

another instance where pipelining doesn't help speedup: **branch misprediction** (return)

- Require lots of nops to avoid data hazards

Incorrect Return Example

```
# demo-ret
0x033:     ret
0x034:     irmovq $1,%rax # Oops!
0x03e:     irmovq $2,%rcx # Oops!
0x048:     irmovq $3,%rdx # Oops!
0x052:     irmovq $5,%rsi # Return
```



- **Incorrectly execute 3 instructions following `ret`**

CS:APP3e

ITU CPH

Pipeline Summary, so far

Concept

- Break instruction execution into 5 stages
- Run instructions through in pipelined mode

Limitations

- Can't handle dependencies between instructions when instructions follow too closely
- Data dependencies
  - One instruction writes register, later one reads it
- Control dependency
  - Instruction sets PC in way that pipeline did not predict correctly
  - Mispredicted branch and return

How to handle these? coming right up...

spoiler: dynamically inject the right number of `nop`s

Fixing the Pipeline

Make the pipelined processor work!

Data Hazards

- Instruction having register R as source follows shortly after instruction having register R as destination
- Common condition, don't want to slow down pipeline

Control Hazards

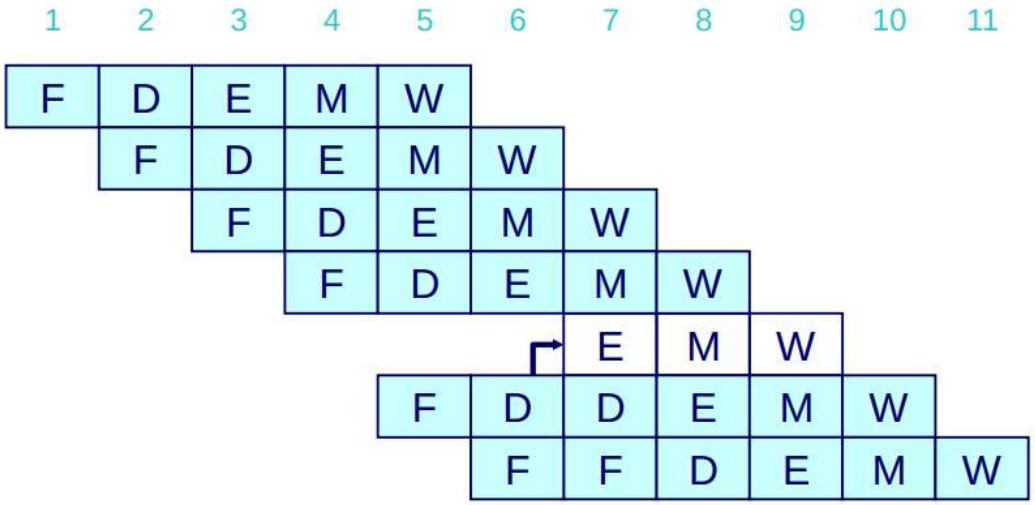
- Mispredict conditional branch
  - Our design predicts all branches as being taken
  - Naïve pipeline executes two extra instructions
- Getting return address for `ret` instruction
  - Naïve pipeline executes three extra instructions

Making Sure It Really Works

- What if multiple special cases happen simultaneously?

Stalling for Data Dependencies

```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
       bubble
0x016: addq %rdx,%rax
0x018: halt
```

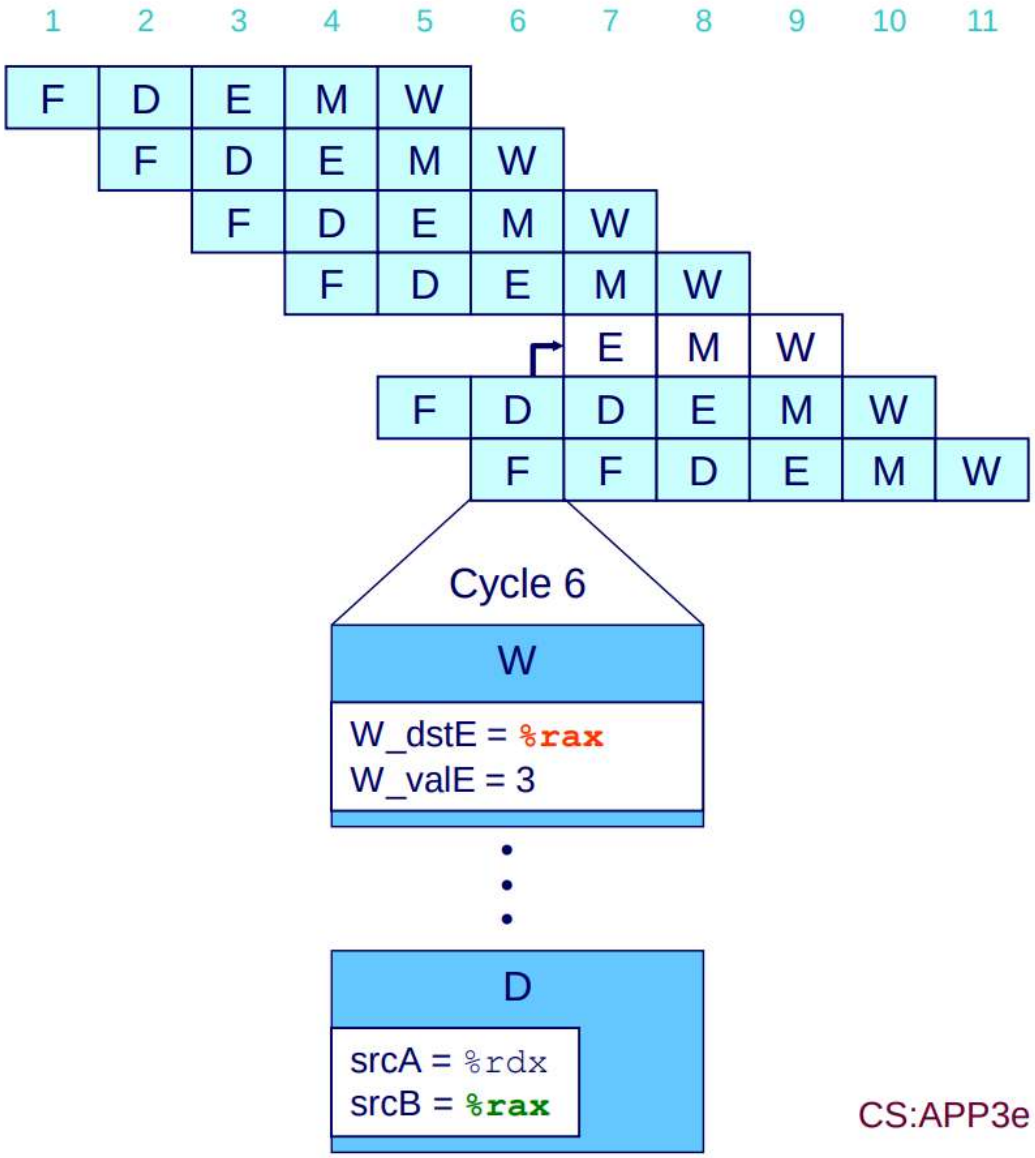


- If instruction follows too closely after one that writes register, slow it down
- Hold instruction in decode
- Dynamically inject nop into execute stage

(a *bubble* is such a dynamically injected nop instruction)

Detecting Stall Condition

```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
       bubble
0x016: addq %rdx,%rax
0x018: halt
```

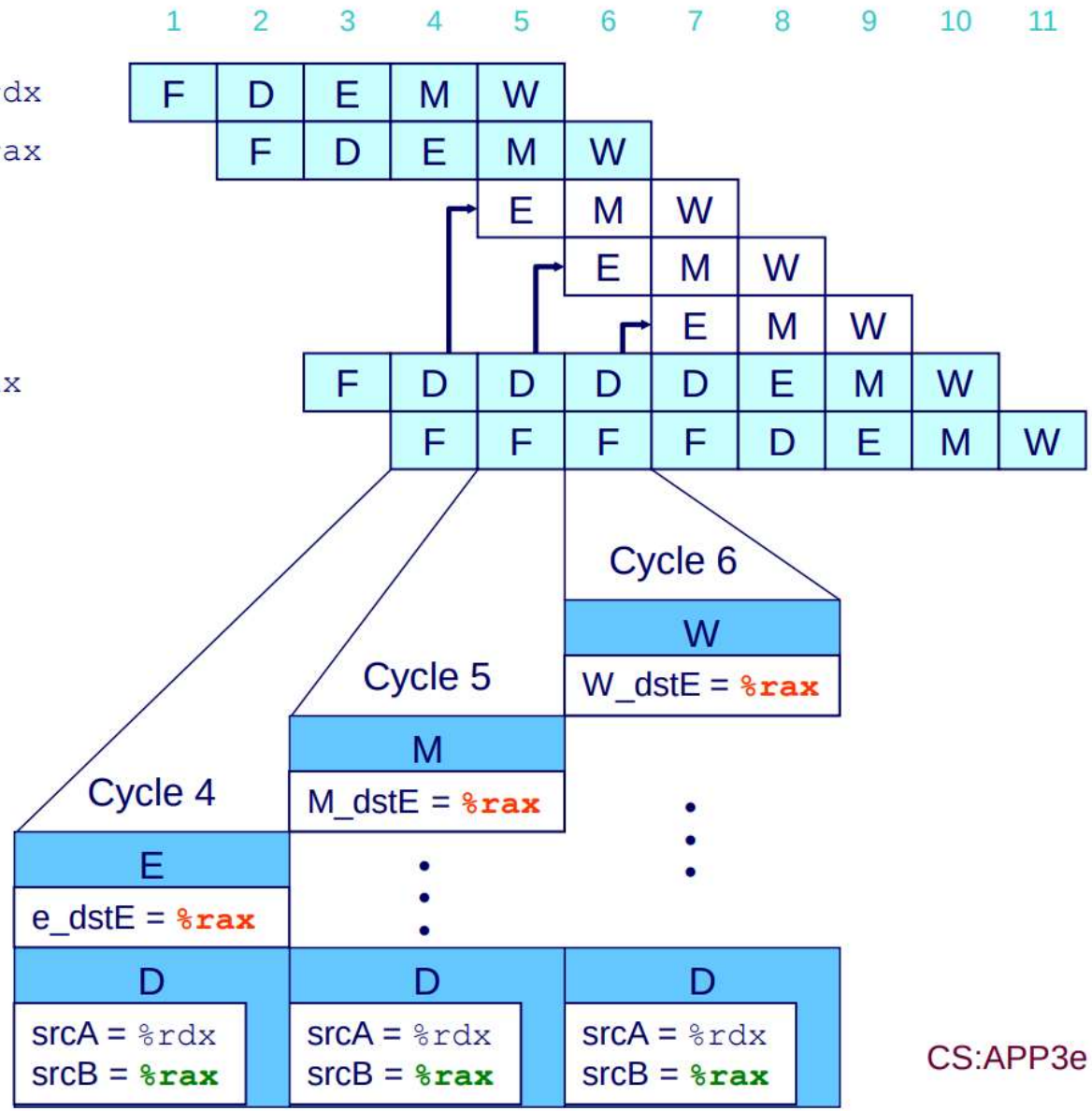


CS:APP3e

ITU CPH

Stalling X3

```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
       bubble
       bubble
       bubble
0x014: addq %rdx,%rax
0x016: halt
```



CS:APP3e

ITU CPH

What Happens When Stalling?

```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

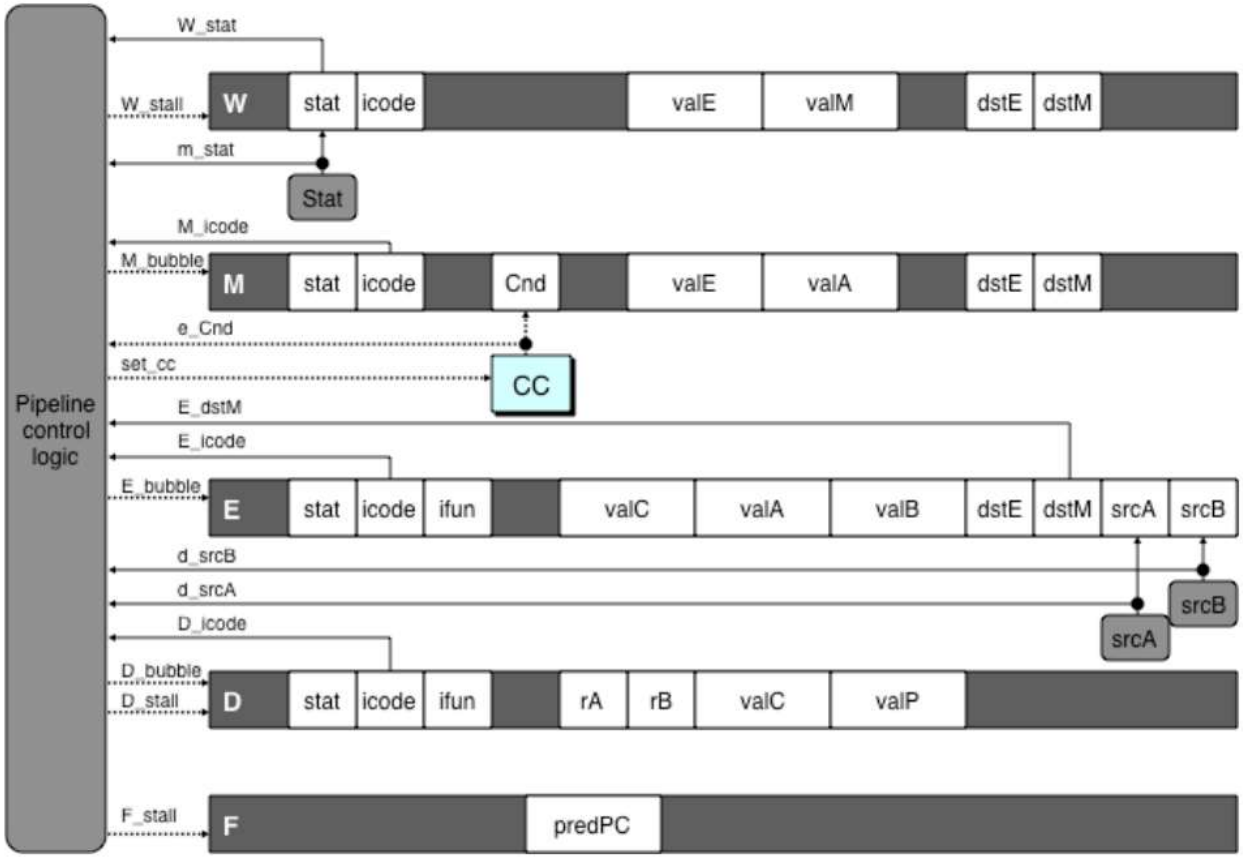
Cycle 8

Write Back	***bubble***
Memory	***bubble***
Execute	`0x014: addq %rdx,%rax`
Decode	`0x016: halt`
Fetch	

- Stalling instruction held back in decode stage
- Following instruction stays in fetch stage
- Bubbles injected into execute stage
  - Like dynamically generated nop's
  - Move through later stages

15s

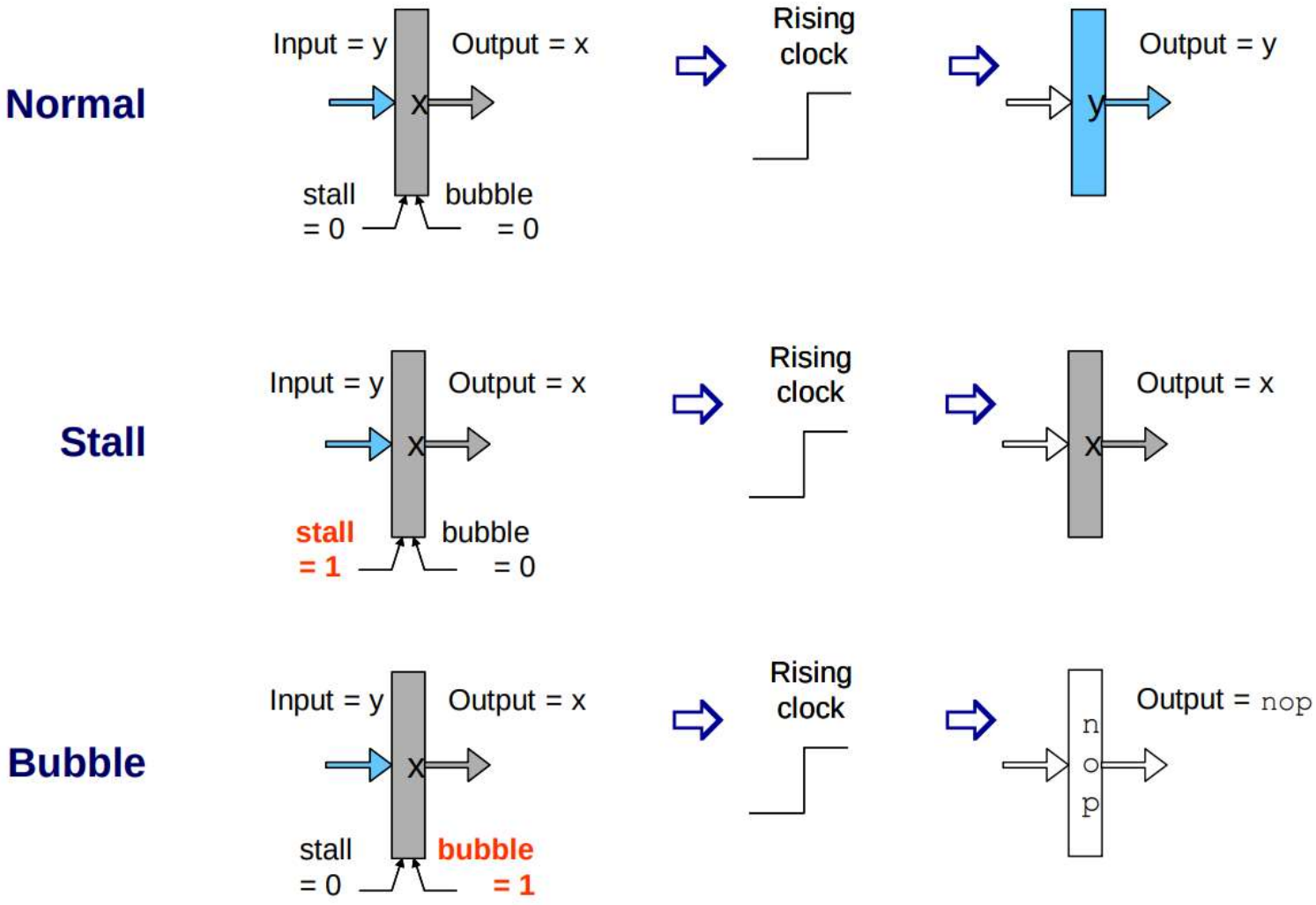
Implementing Stalling



Pipeline Control

- Combinational logic detects stall condition
- Sets mode signals for how pipeline registers should update

Pipeline Register Modes



CS:APP3e

ITU CPH

Data Forwarding

Naïve Pipeline

- Register isn't written until completion of write-back stage
- Source operands read from register file in decode stage
  - Needs to be in register file at start of stage

late

too early

Observation

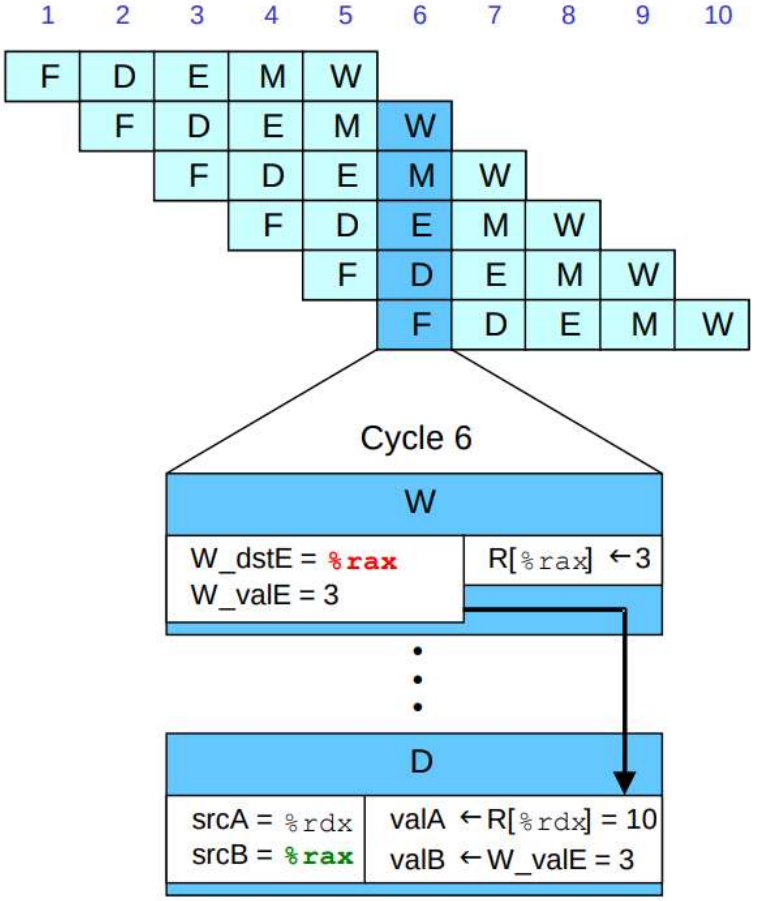
- Value generated is already in execute or memory stage

Trick

- Pass value directly from generating instruction to decode stage
- Needs to be available at end of decode stage

Data Forwarding Example

```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



- `irmovq` in write-back stage
- Destination value in W pipeline register
- Forward as valB for decode stage

didn't need to **bubble**!

implemented by:

Bypass Paths

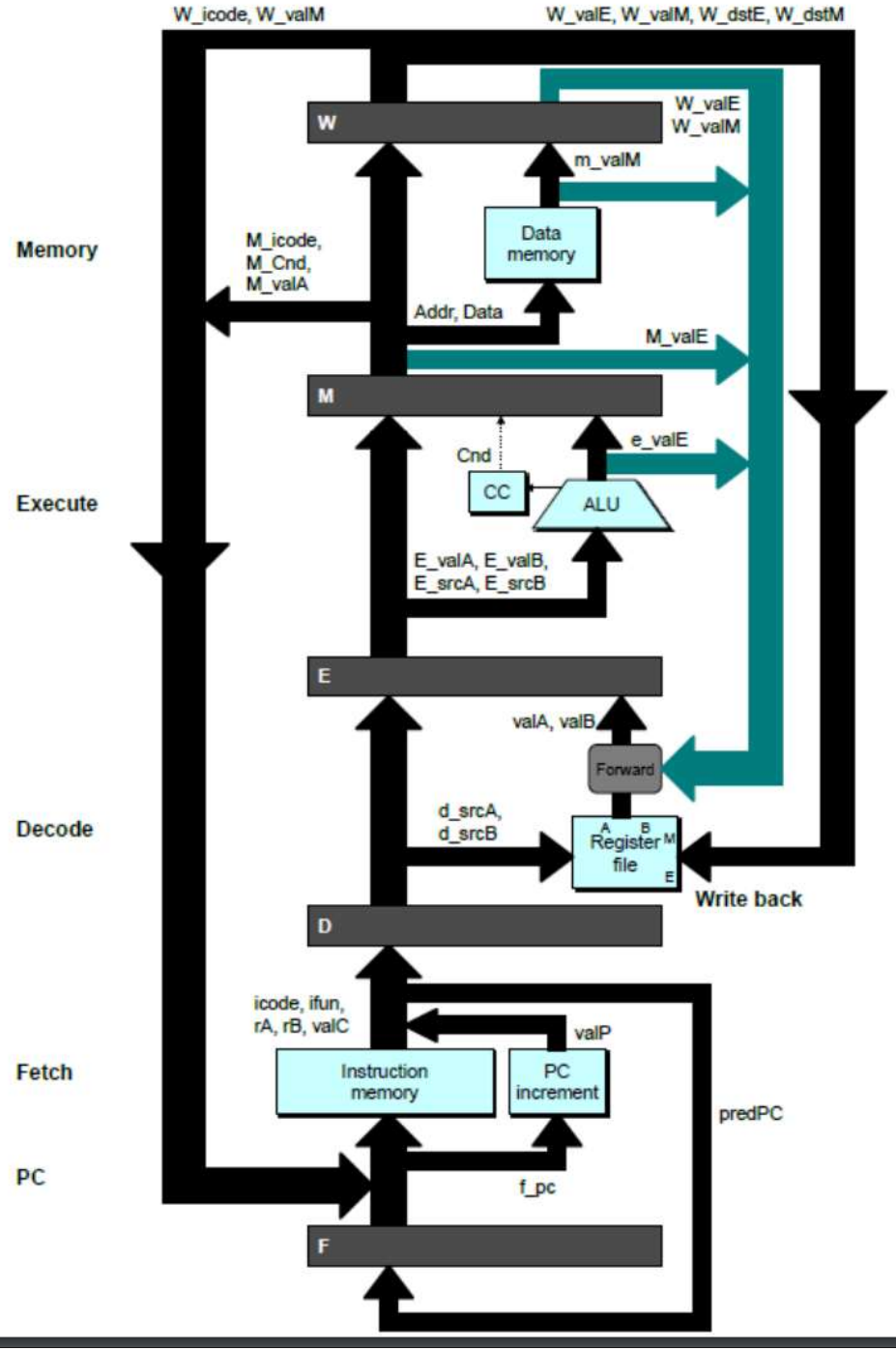
Decode Stage

- Forwarding logic selects valA and valB
- Normally from register file
- Forwarding: get valA or valB from later pipeline stage

Forwarding Sources

- Execute: valE
- Memory: valE, valM
- Write back: valE, valM

& logic

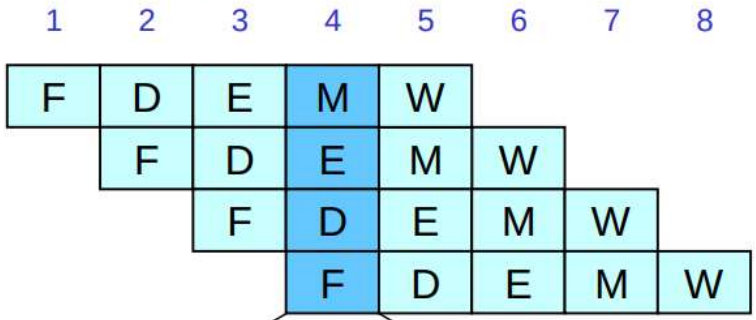


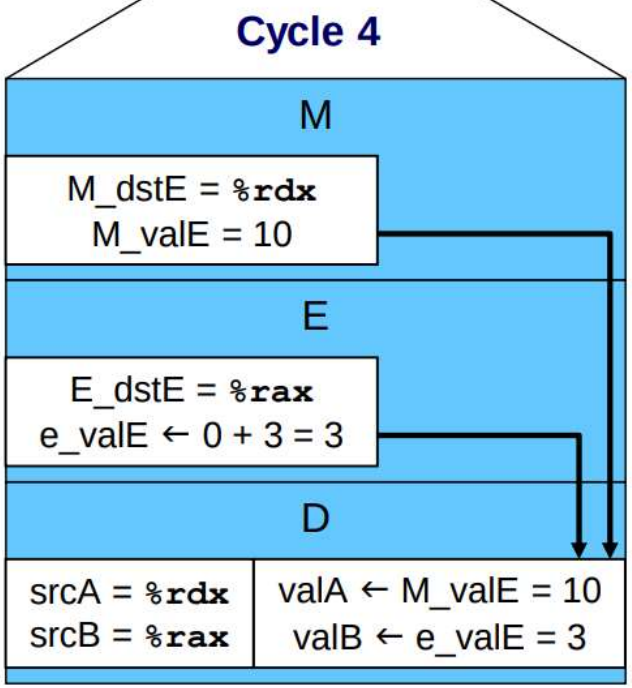
– 46 –

ITU CPH

Data Forwarding Example #2

```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```





Register %rdx

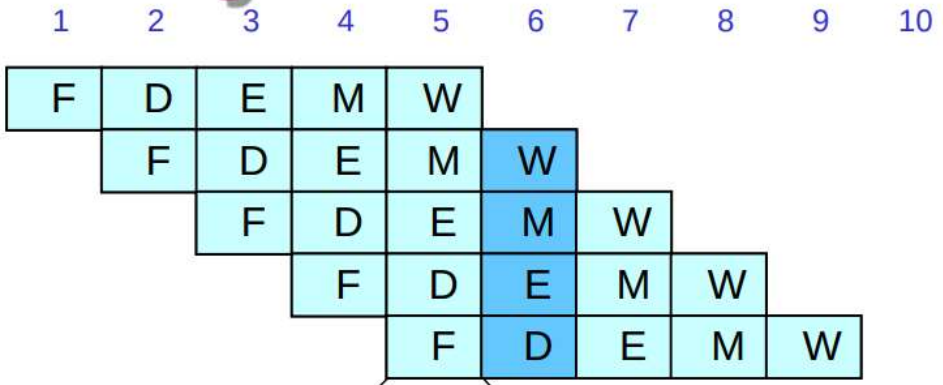
- Generated by ALU during previous cycle
- Forward from memory as valA

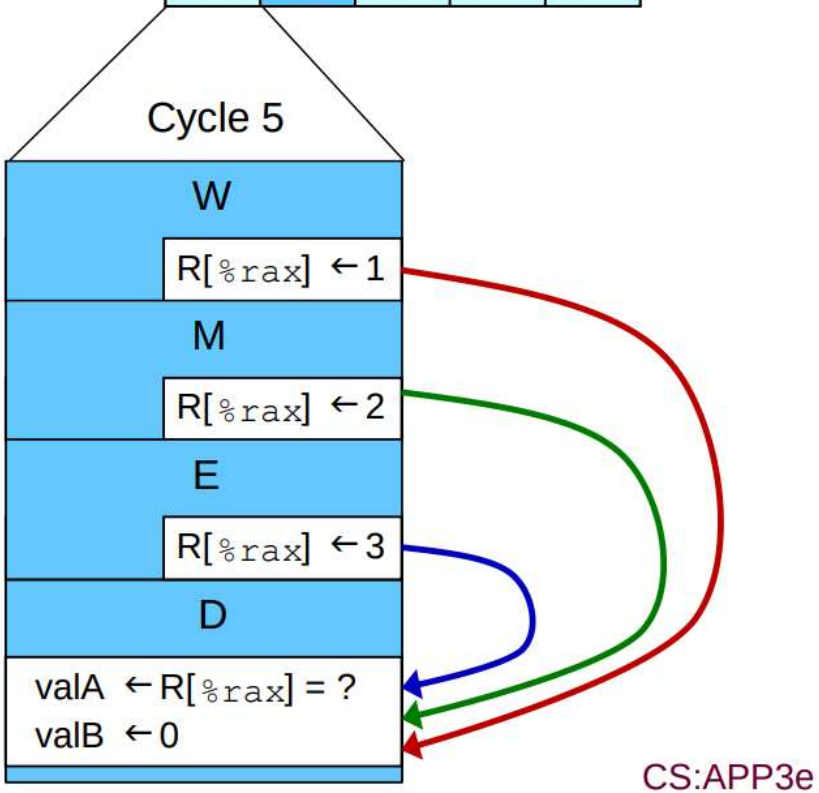
Register %rax

- Value just generated by ALU
- Forward from execute as valB

Forwarding Priority

```
# demo-priority.ys
0x000: irmovq $1, %rax
0x00a: irmovq $2, %rax
0x014: irmovq $3, %rax
0x01e: rrmovq %rax, %rdx
0x020: halt
```





Multiple Forwarding Choices

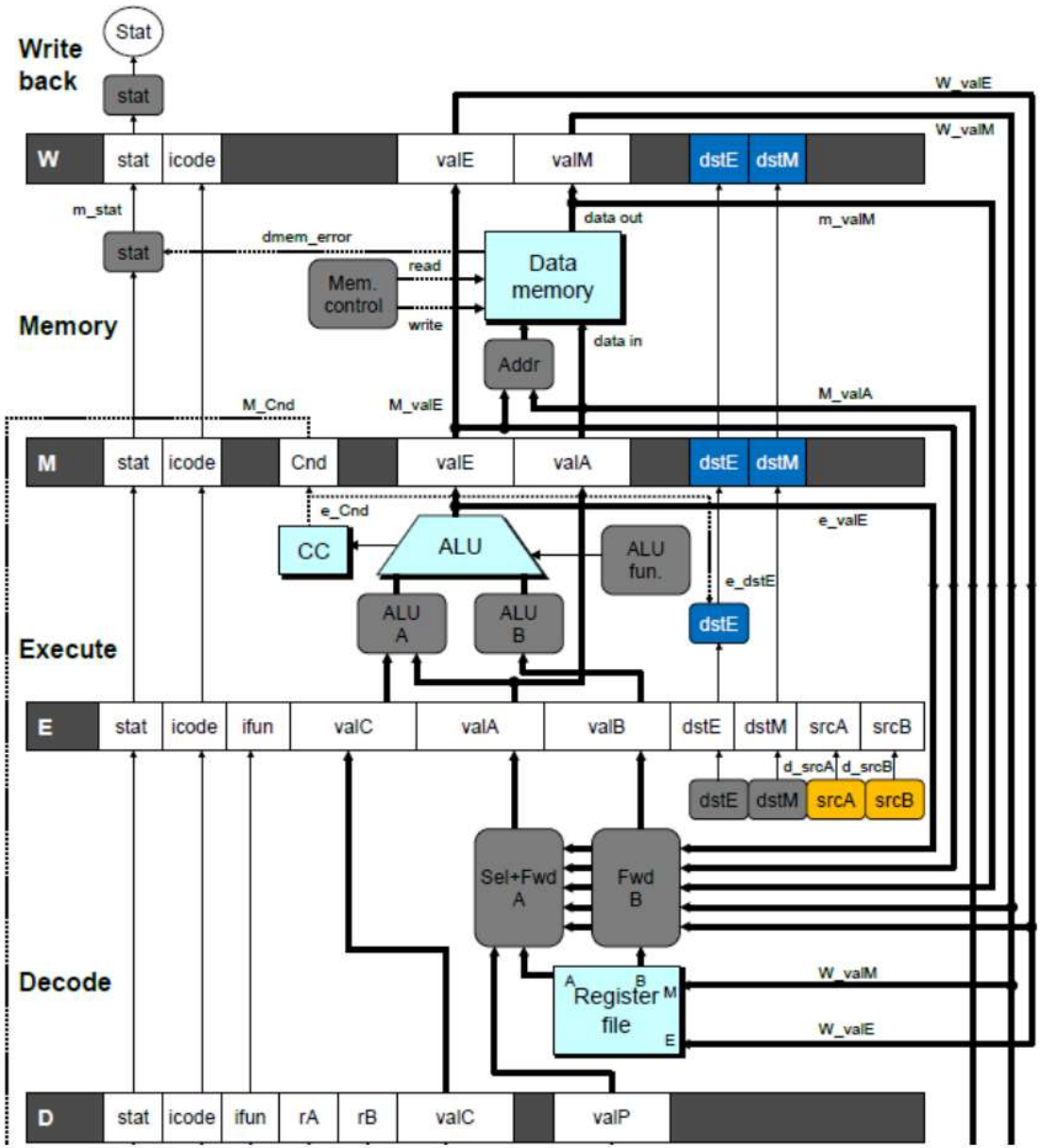
- Which one should have priority
- Match serial semantics
- Use matching value from earliest pipeline stage

==> most up-to-date value

CS:APP3e

ITU CPH

Implementing Forwarding

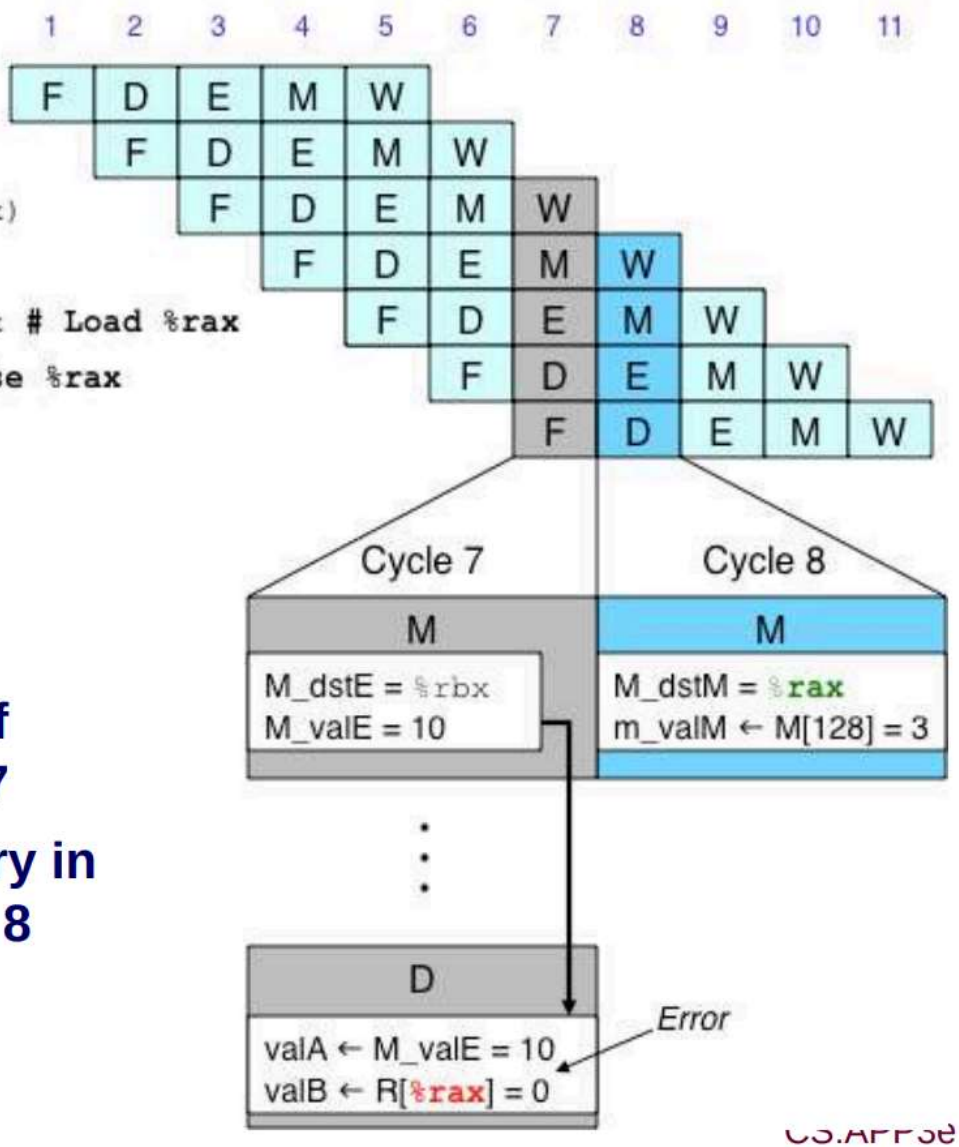


- Add additional feedback paths from E, M, and W pipeline registers into decode stage
- Create logic blocks to select from multiple sources for valA and valB in decode stage

15s

Limitation of Forwarding

```
# demo-luh.ys
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
0x032: addq %rbx,%rax # Use %rax
0x034: halt
```



Load-use dependency

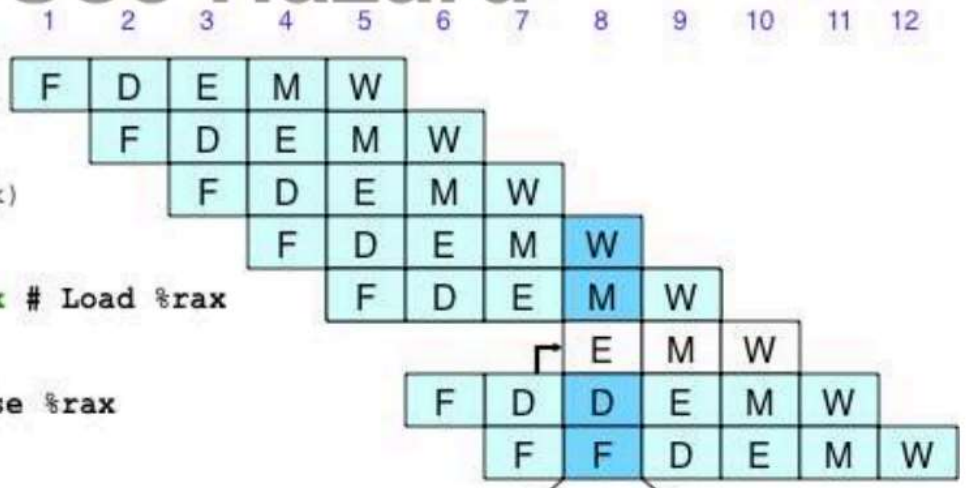
- Value needed by end of decode stage in cycle 7
- Value read from memory in memory stage of cycle 8

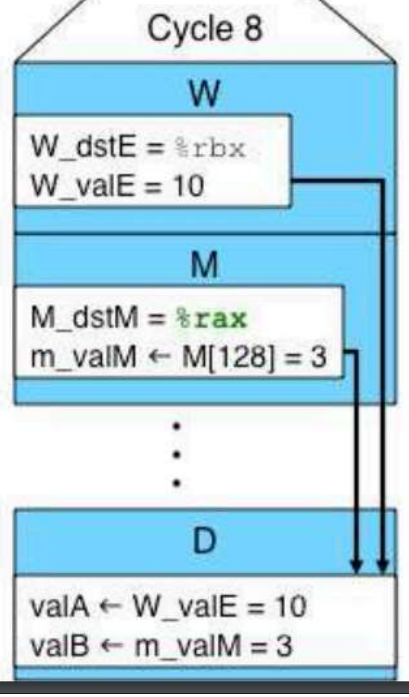
`mrmovq` read from memory hasn't happened yet; cannot forward it.

solution: stall

Avoiding Load/Use Hazard

```
# demo-luh.ys
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
     bubble
0x032: addq %rbx,%rax # Use %rax
0x034: halt
```



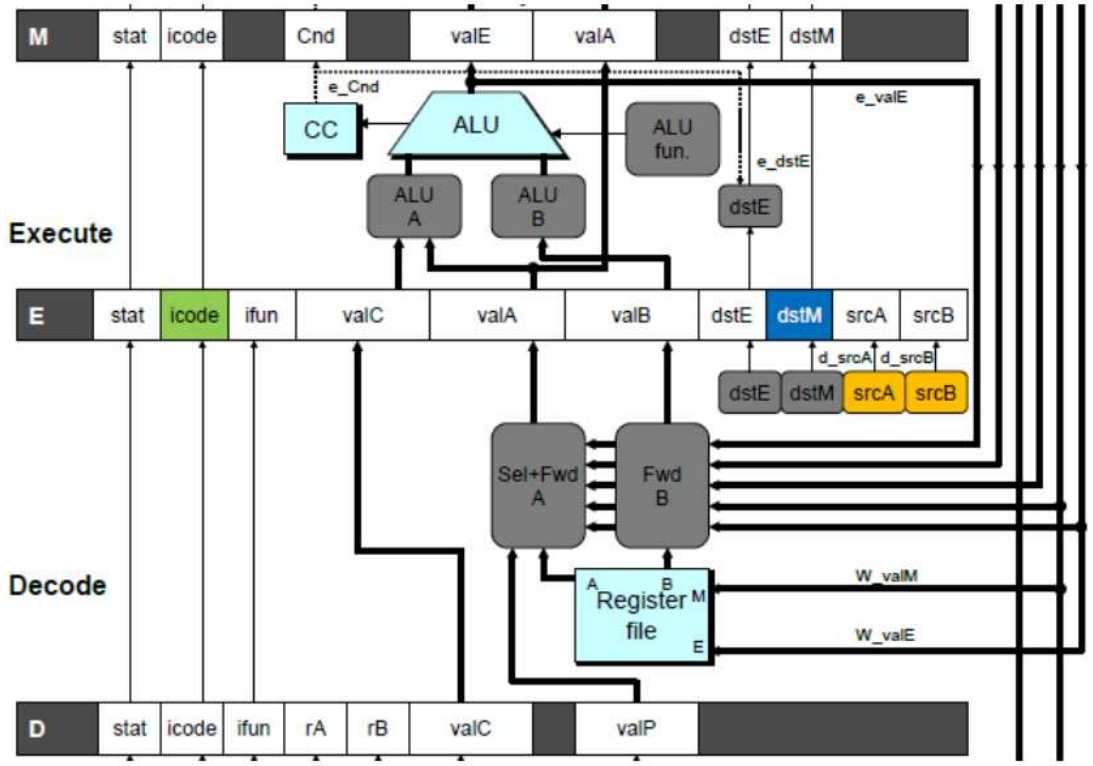


- Stall using instruction for one cycle
- Can then pick up loaded value by forwarding from memory stage

ITU CPH

5s

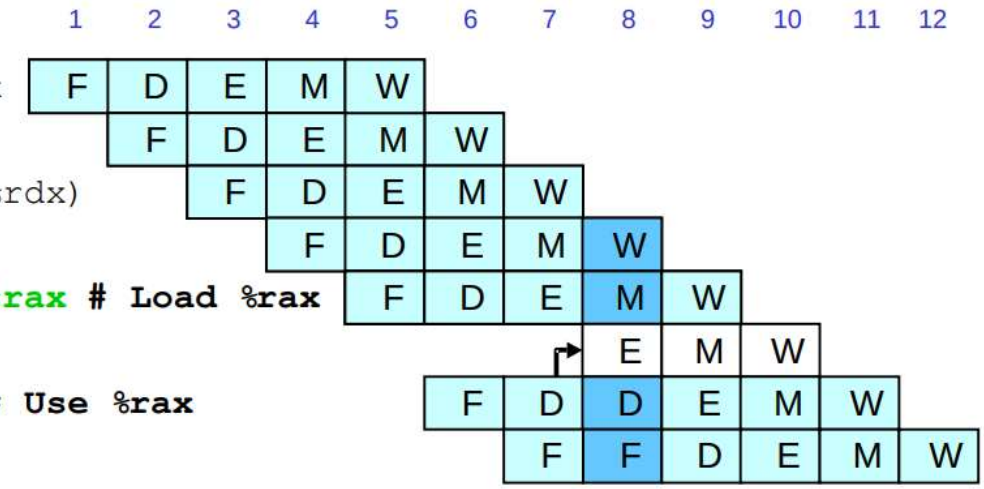
Detecting Load/Use Hazard



Condition	Trigger
Load/Use Hazard	`E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }`

Control for Load/Use Hazard

```
# demo-luh.ys
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%ebx
0x028: mrmovq 0(%rdx),%rax # Load %rax
       bubble
0x032: addq %ebx,%rax # Use %rax
0x034: halt
```



- Stall instructions in fetch and decode stages
- Inject bubble into execute stage

Condition	F	D	E	M	W
Load/Use Hazard	stall	stall	bubble	normal	normal

Branch Misprediction Example

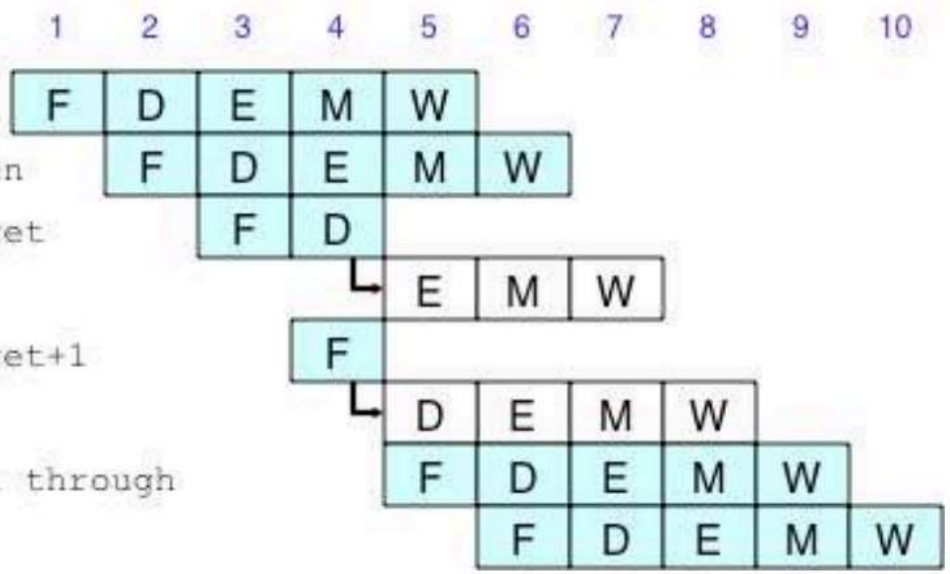
demo-j.ys

```
0x000:     xorq %rax,%rax
0x002:     jne  t                # Not taken
0x00b:     irmovq $1, %rax       # Fall through
0x015:     nop
0x016:     nop
0x017:     nop
0x018:     halt
0x019: t:  irmovq $3, %rdx       # Target
0x023:     irmovq $4, %rcx       # Should not execute
0x02d:     irmovq $5, %rdx       # Should not execute
```

- Should only execute first 8 instructions

Handling Misprediction

```
# demo-j.ys
0x000: xorq %rax,%rax
0x002: jne target # Not taken
0x016: irmovq $2,%rdx # Target
       bubble
0x020: irmovq $3,%rbx # Target+1
       bubble
0x00b: irmovq $1,%rax # Fall through
0x015: halt
```



Predict branch as taken

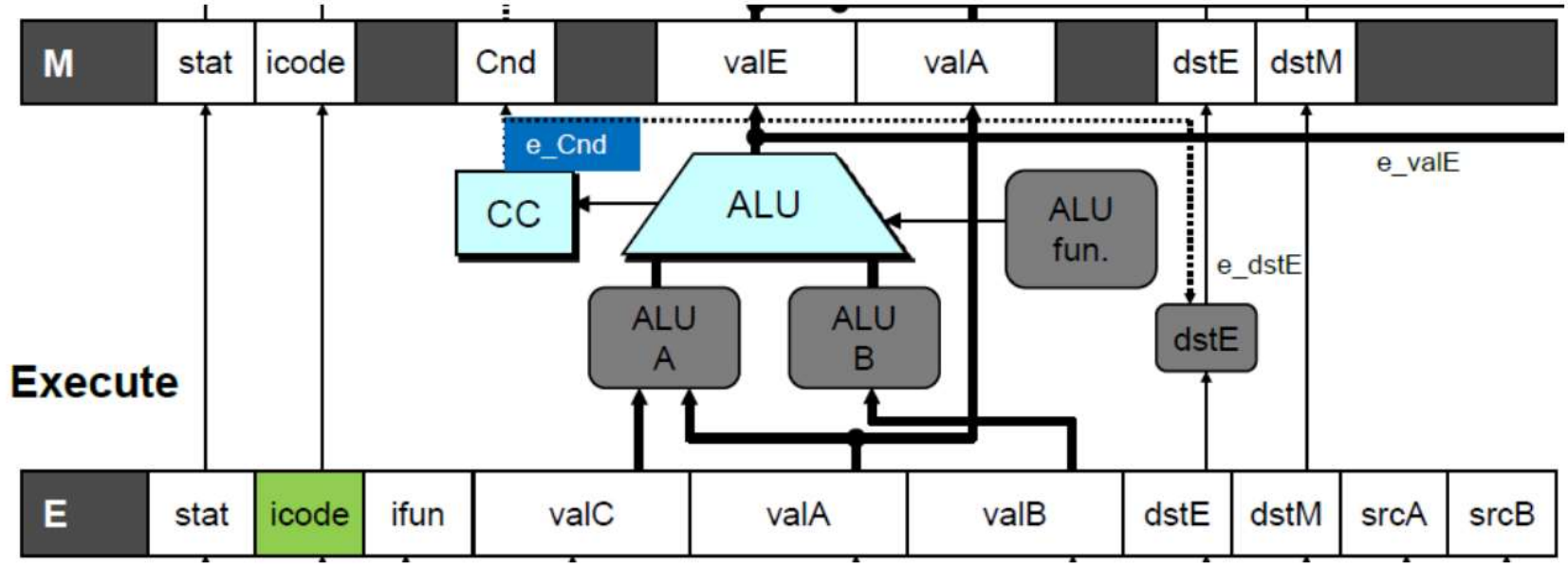
- Fetch 2 instructions at target

Cancel when mispredicted

- Detect branch not-taken in execute stage
- On following cycle, replace instructions in execute and decode by bubbles
- No side effects yet (bad Ws haven't happened)

5s

Detecting Mispredicted Branch

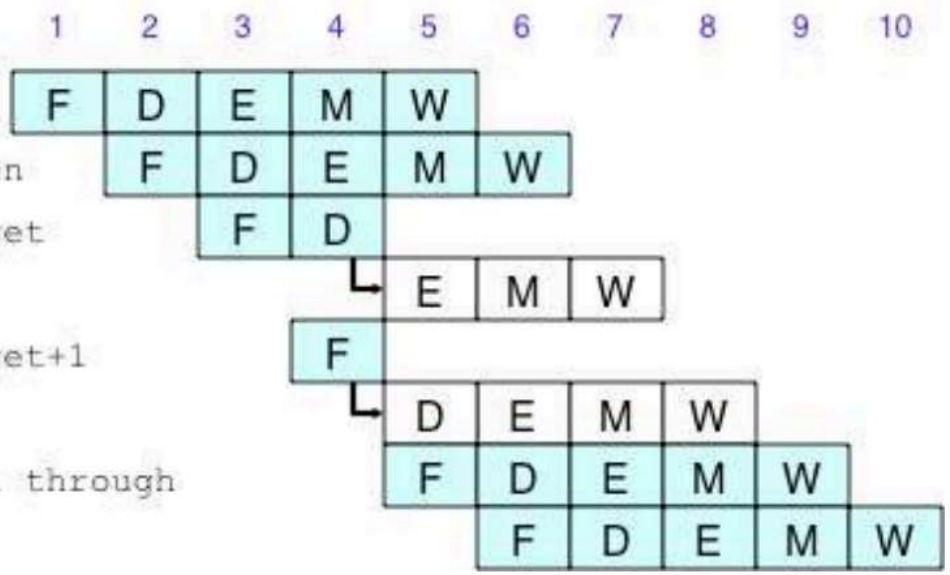


Condition	Trigger
Mispredicted Branch	`E_icode = IJXX & !e_Cnd`

it's a jump, and it's false (we predict true)

Control for Misprediction

```
# demo-j.ys
0x000: xorq %rax,%rax
0x002: jne target # Not taken
0x016: irmovq $2,%rdx # Target
       bubble
0x020: irmovq $3,%rbx # Target+1
       bubble
0x00b: irmovq $1,%rax # Fall through
0x015: halt
```



Condition	F	D	E	M	W
Mispredicted Branch	normal	bubble	bubble	normal	normal

Return Example

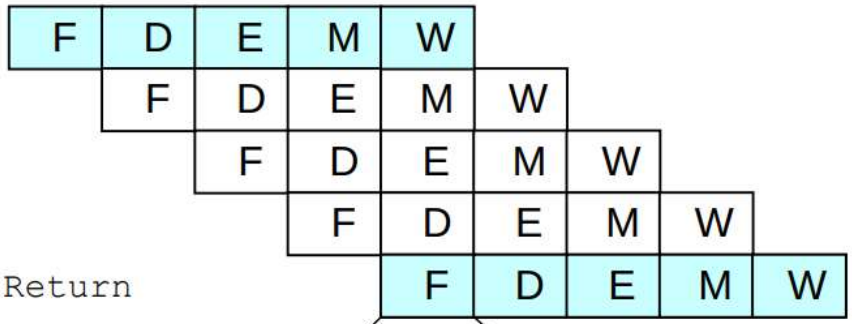
demo-retb.ys

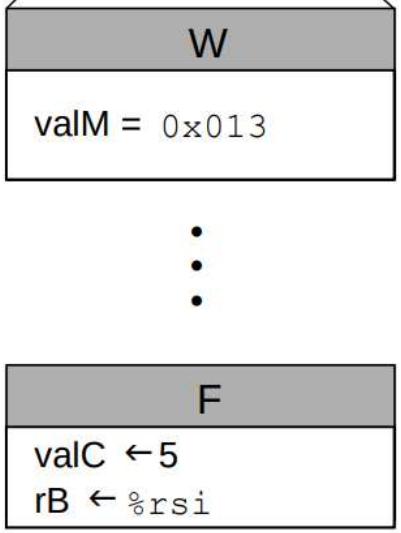
```
0x000:     irmovq Stack,%rsp  # Intialize stack pointer
0x00a:     call p             # Procedure call
0x013:     irmovq $5,%rsi     # Return point
0x01d:     halt
0x020:  .pos 0x20
0x020: p:  irmovq $-1,%rdi    # procedure
0x02a:     ret
0x02b:     irmovq $1,%rax     # Should not be executed
0x035:     irmovq $2,%rcx     # Should not be executed
0x03f:     irmovq $3,%rdx     # Should not be executed
0x049:     irmovq $4,%rbx     # Should not be executed
0x100:  .pos 0x100
0x100: Stack:                 # Stack: Stack pointer
```

- Previously executed three additional instructions

Correct Return Example

```
# demo-retb
0x026:      ret
            bubble
            bubble
            bubble
0x013:      irmovq $5,%rsi # Return
```





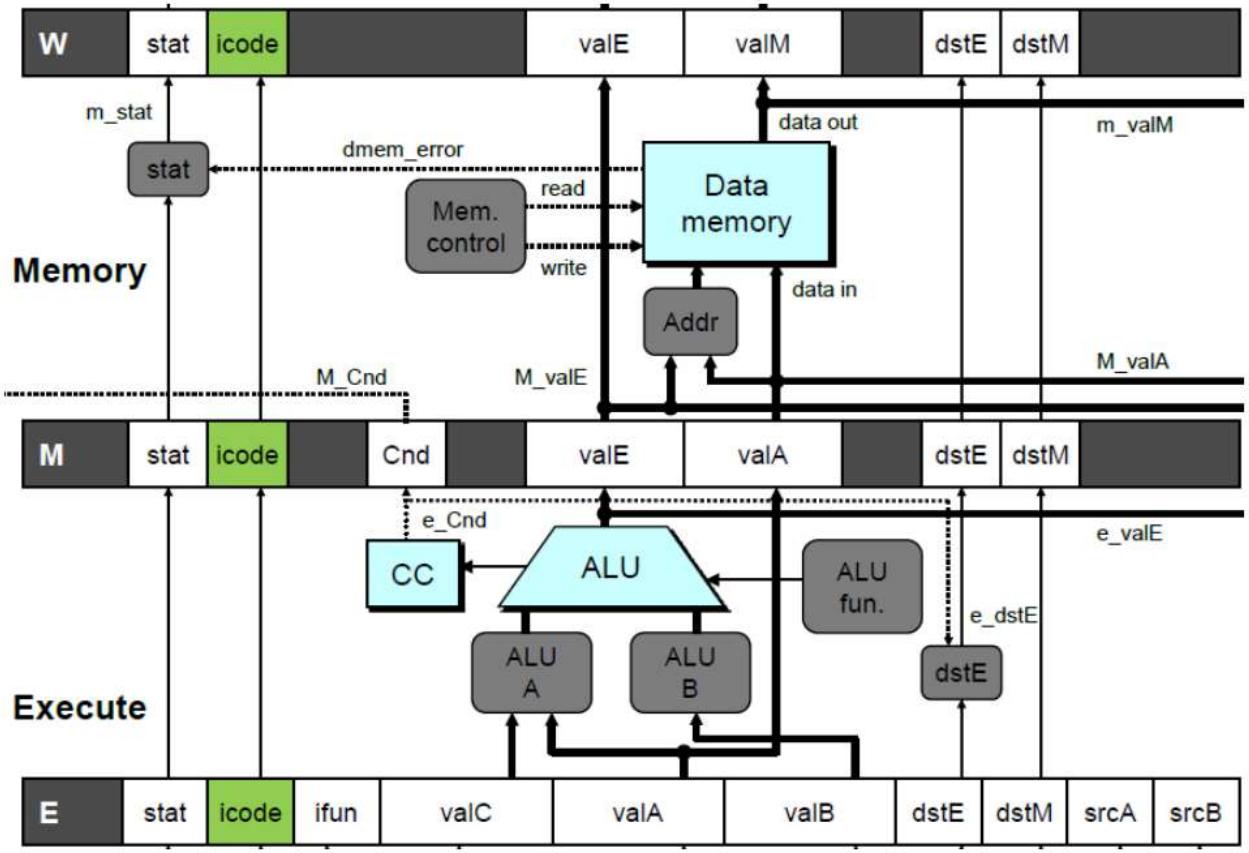
- As `ret` passes through pipeline, stall at fetch stage
  - While in decode, execute, and memory stage
- Inject bubble into decode stage
- Release stall when reach write-back stage

basically stop pipelining until we've read the return address.

ITU CPH

5s

Detecting Return



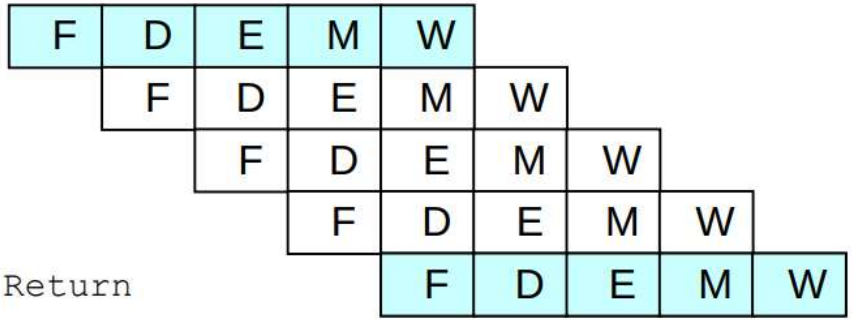
Condition	Trigger
Processing `ret`	`IRET in { D_icode, E_icode, M_icode }`

CS:APP3e

ITU CPH

Control for Return

```
# demo-retb
0x026:     ret
           bubble
           bubble
           bubble
0x014:     irmovq $5,%rsi # Return
```



Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal

Summary: Special Control Cases

Detection

Condition	Trigger
Processing `ret`	IRET in { D_icode, E_icode, M_icode }
Load/Use Hazard	E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }
Mispredicted Branch	E_icode = IJXX & !e_Cnd

Action (on next cycle)

Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal

Performance Metrics

Clock rate

- Measured in Gigahertz
- Function of stage partitioning and circuit design
  - Keep amount of work per stage small

Rate at which instructions executed

- CPI: cycles per instruction
- On average, how many clock cycles does each instruction require?
- Function of pipeline design and benchmark programs
  - E.g., how frequently are branches mispredicted?

CPI for PIPE

CPI ≈ 1.0

- Fetch instruction each clock cycle
- Effectively process new instruction almost every cycle
  - Although each individual instruction has latency of 5 cycles

CPI > 1.0

- Sometimes must stall or cancel branches

Computing CPI

- C clock cycles
- I instructions executed to completion
- B bubbles injected (C = I + B)

$$CPI = C/I = (I+B)/I = 1.0 + B/I$$

- Factor B/I represents average penalty due to bubbles

CPI for PIPE (Cont.)

B/I = LP + MP + RP

	Typical Values
■ LP: Penalty due to load/use hazard stalling	
● Fraction of instructions that are loads	0.25
● Fraction of load instructions requiring stall	0.20
● Number of bubbles injected each time	1
⇒ LP = 0.25 * 0.20 * 1 = 0.05	
■ MP: Penalty due to mispredicted branches	
● Fraction of instructions that are cond. jumps	0.20
● Fraction of cond. jumps mispredicted	0.40
● Number of bubbles injected each time	2
⇒ MP = 0.20 * 0.40 * 2 = 0.16	
■ RP: Penalty due to `ret` instructions	
● Fraction of instructions that are returns	0.02
● Number of bubbles injected each time	3
⇒ RP = 0.02 * 3 = 0.06	

■ Net effect of penalties 0.05 + 0.16 + 0.06 = 0.27

⇒ CPI = 1.27 (Not bad!)

Pipelining Summary

Data Hazards

- Most handled by forwarding: no performance penalty
- Load/use hazard requires one cycle stall

Control Hazards

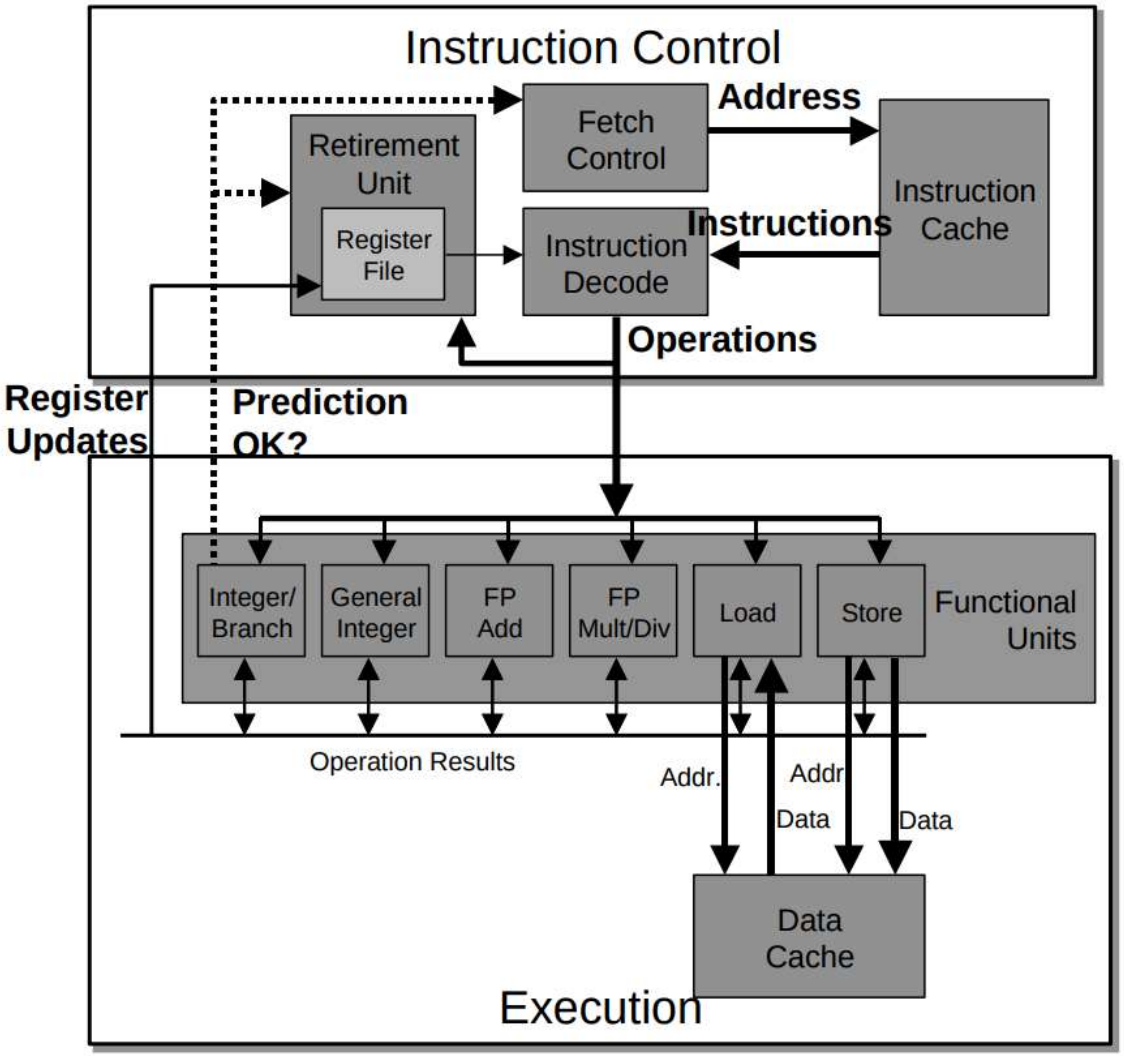
- Cancel instructions when detect mispredicted branch
  - Two clock cycles wasted
- Stall fetch stage while `ret` passes through pipeline
  - Three clock cycles wasted

Control Combinations (Corner Cases)

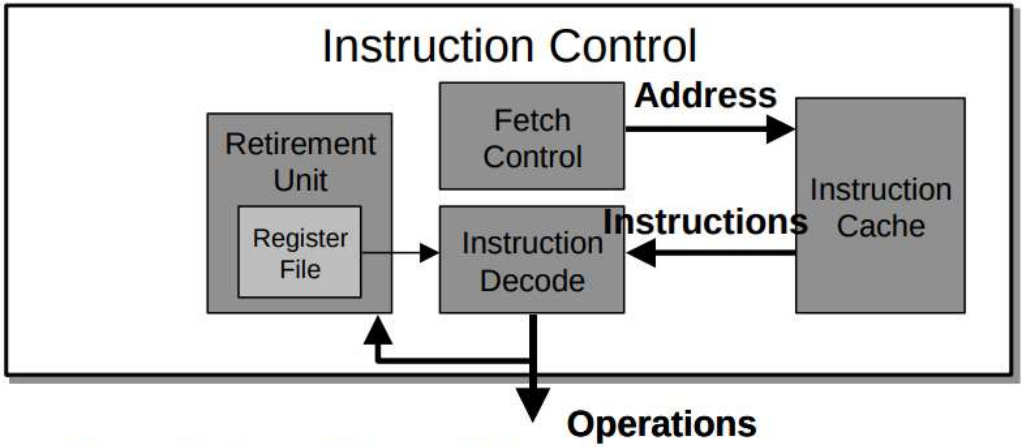
- Must analyze carefully; first version subtle bug


ITU CPH

Modern CPU Design



Instruction Control



Grabs Instruction Bytes From Memory

- Based on Current PC + Predicted Targets for Predicted Branches
- Hardware dynamically guesses whether branches taken/not taken and (possibly) branch target

Translates Instructions Into *Operations*

- Primitive steps to perform instruction
- Typical instruction requires 1–3 operations

this is called **microcode**
shallower logic ==> more parallelizable
ex: (add instruction)

```
T4:  IR_out(addr part), MAR_in
T5:  read
T6:  ACC_out, aluadd
T7:  TEMP_out, ACC_in, reset to T0
```

(bonus: later, add instructions to ISA!)

Converts Register References Into *Tags*

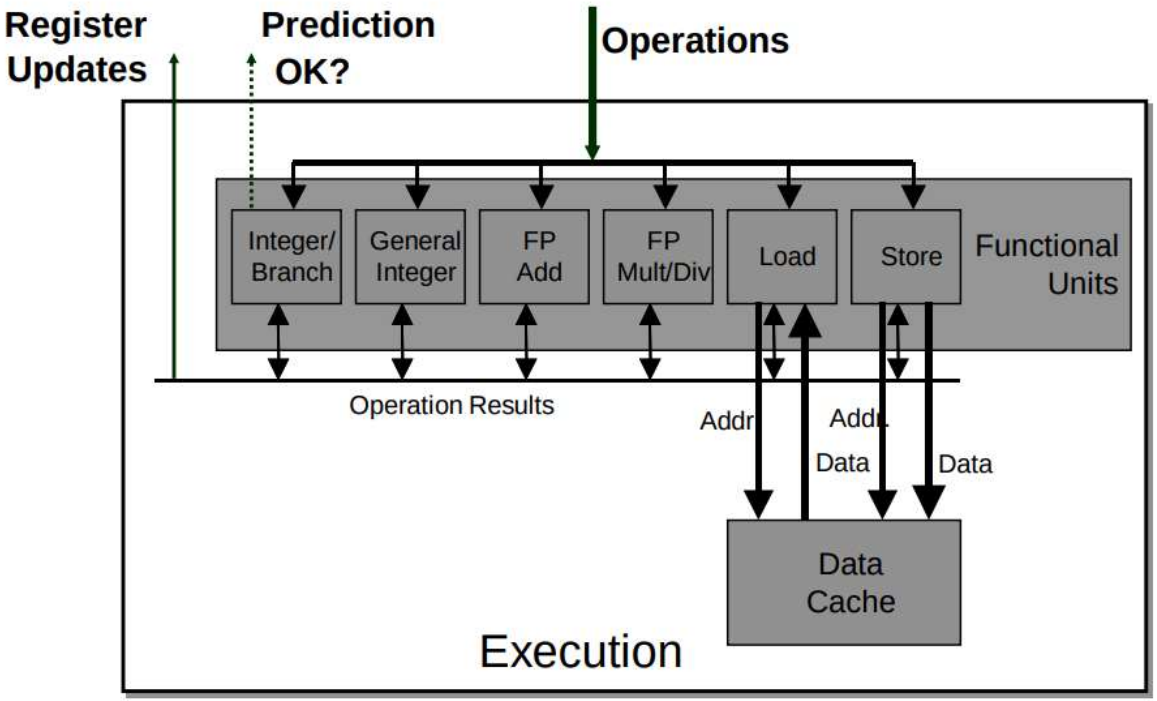
- Abstract identifier linking destination of one operation with sources of later operations

this technique is called **register renaming**.
eliminates false data dependencies.

```
r1 := m[1024]          r1 := m[1024]
r1 := r1 + 2           r1 := r1 + 2
m[1032] := r1          m[1032] := r1
r1 := m[2048]          r2 := m[2048]
r1 := r1 + 4           r2 := r1 + 4
m[2056] := r1          m[2056] := r2
```


ITU CPH

Execution Unit



- Multiple functional units
  - Each can operate in independently
- Operations performed as soon as operands available
  - Not necessarily in program order
  - Within limits of functional units
- Control logic
  - Ensures behavior equivalent to sequential program execution

this is called **out-of-order execution.** CPU reorders instructions, to e.g. Avoid need to stall/bubble if there's a data-independent instruction nearby.

CPU Capabilities of Intel Haswell

Multiple Instructions Can Execute in Parallel

- 2 load
- 1 store
- 4 integer
- 2 FP multiply
- 1 FP add / divide

Some Instructions Take > 1 Cycle, but Can be Pipelined

Instruction	Latency	Cycles/Issue
Load / Store	4	1
Integer Multiply	3	1
Integer Divide	3—30	3—30
Double/Single FP Multiply	5	1
Double/Single FP Add	3	1
Double/Single FP Divide	10—15	6—11

Haswell Operation

Translates instructions dynamically into “Uops”

- ~118 bits wide
- Holds operation, two sources, and destination

Executes Uops with “Out of Order” engine

- Uop executed when
  - Operands available
  - Functional unit available
- Execution controlled by “Reservation Stations”
  - Keeps track of data dependencies between uops
  - Allocates resources

High-Performance Branch Prediction

Critical to Performance

- Typically 11–15 cycle penalty for misprediction

Branch Target Buffer

- 512 entries
- 4 bits of history
- Adaptive algorithm
  - Can recognize repeated patterns, e.g., alternating taken–not taken

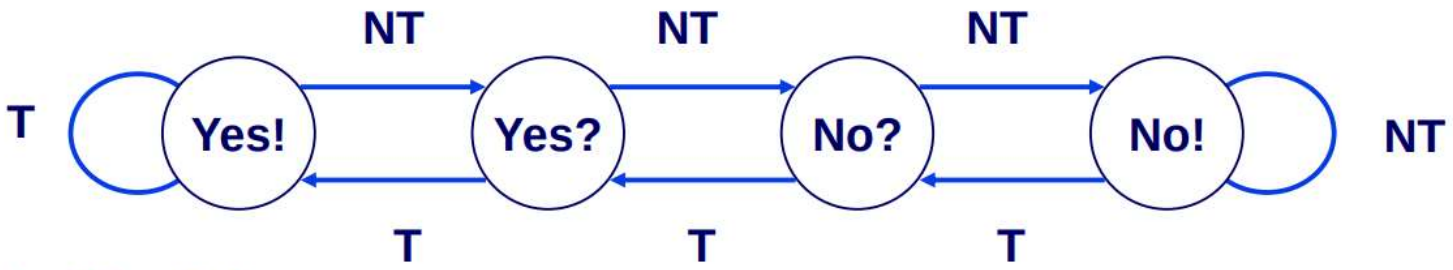
Handling BTB misses

- Detect in ~cycle 6
- Predict taken for negative offset, not taken for positive
  - Loops vs. conditionals

Example Branch Prediction

Branch History

- Encode information about prior history of branch instructions
- Predict whether or not branch will be taken



State Machine

- Each time branch taken, transition to right
- When not taken, transition to left
- Predict branch taken when in state Yes! or Yes?

Processor Summary

Design Technique

- Create uniform framework for all instructions
  - Want to share hardware among instructions
- Connect standard logic blocks with bits of control logic

Operation

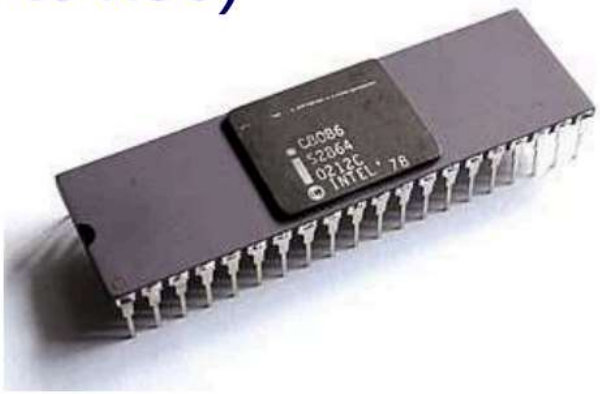
- State held in memories and clocked registers
- Computation done by combinational logic
- Clocking of registers/memories sufficient to control overall behavior

Enhancing Performance

- Pipelining increases throughput and improves resource utilization
- Must make sure to maintain ISA behavior

All this... to understand an old CPU?

all these ideas are implemented in the Intel 8086 (1976) (gave rise to x86)



since: aggressively optimizing this design, incorporating breakthroughs in chip manufacturing

"Hey, I ran on airplanes, the Nintendo Gameboy, and all sorts of things!"

… until ca. 2005; we hit a limit.
since: multicore (next lecture).

Vectorization

ITU CPH

SIMD Vector Instructions in a Nutshell

- **What are these instructions?**
  - Extension of the ISA. Data types and instructions for parallel computation on short (2-16) vectors of integers and floats

4-way

- **Why are they here?**
  - **Useful:** Many applications (e.g.,multi media) feature the required fine grain parallelism – code potentially faster
  - **Doable:** Chip designers have enough transistors available, easy to implement

SIMD = Single Instruction Multiple Data

Franchetti: 18-613: Foundations of Computer Systems, Lecture 4. Based on Material by M. Püschel, ETH Zürich.

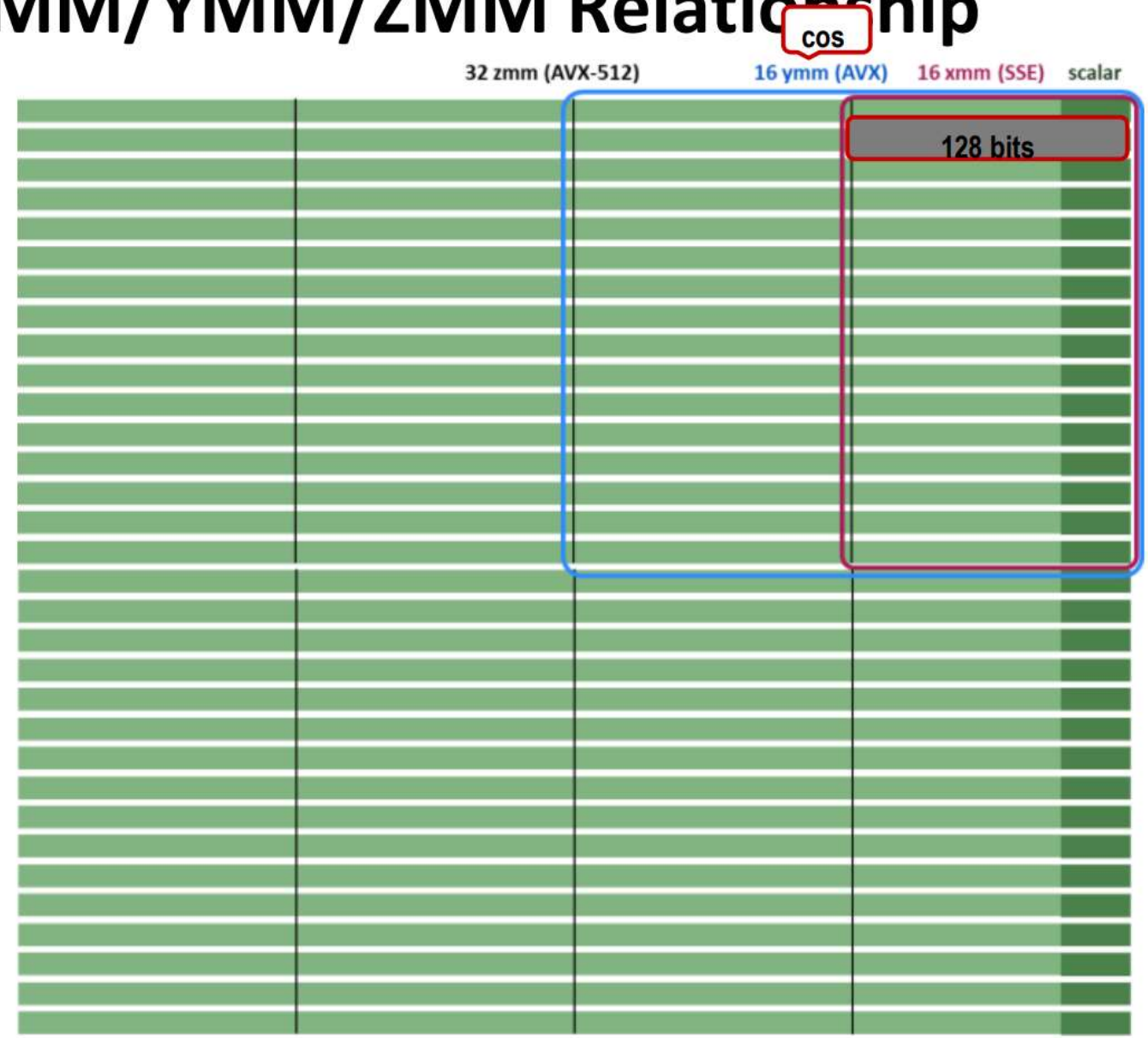
ITU CPH

What are the new registers?

Carnegie Mellon

Electrical & Computer ENGINEERING

XMM/YMM/ZMM Relationship



Instruction set extensions add new register names. Their storage overlaps.

Franchetti: 18-613: Foundations of Computer Systems, Lecture 4. Based on Material by M. Püschel, ETH Zürich.

ITU CPH

What can you put in a register?

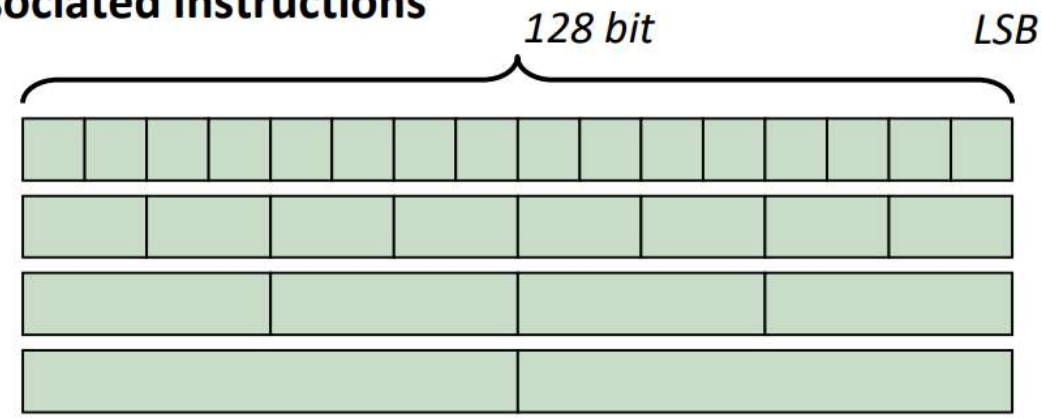
Carnegie Mellon

Electrical & Computer ENGINEERING

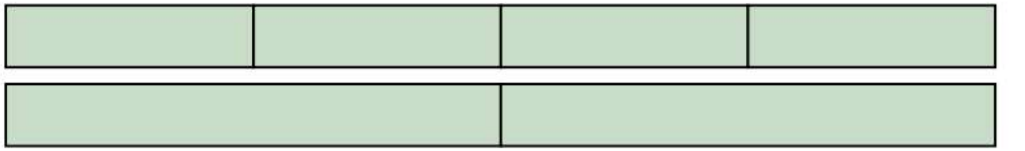
Detail: XMM/SSE2/3/4 Register

This is SSE example (128 bit reg).
For instance, **AVX512** lets you op on eight 64-bit or **sixteen** 32-bit integers

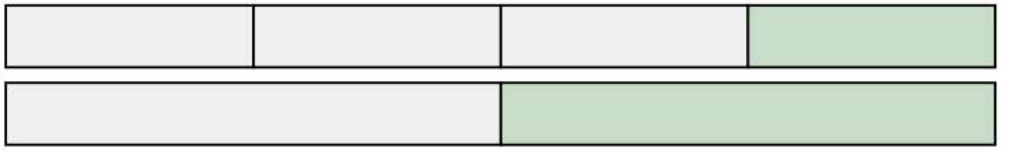
- **Different data types and associated instructions**
- **Integer vectors:**
  - 16-way byte
  - 8-way 2 bytes
  - 4-way 4 bytes
  - 2-way 8 bytes



- **Floating point vectors:**
  - 4-way single (since SSE)
  - 2-way double (since SSE2)



- **Floating point scalars:**
  - single (since SSE)
  - double (since SSE2)



Franchetti: 18-613: Foundations of Computer Systems, Lecture 4. Based on Material by M. Püschel, ETH Zürich.

ITU CPH

Carnegie Mellon

Electrical & Computer ENGINEERING

Instruction Names

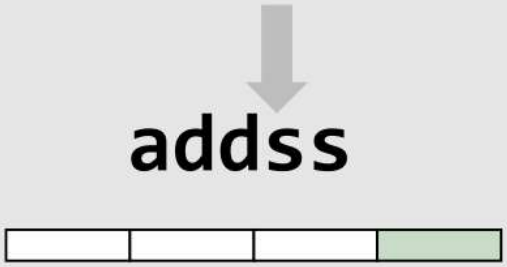
packed (vector)

addps

single precision

single slot (scalar)

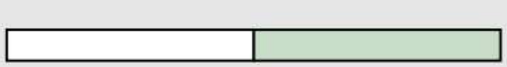
addss



vaddpd

3 op *double precision*

addsd



SSE vs AVX: register operand decides

Compiler will use this for floating point

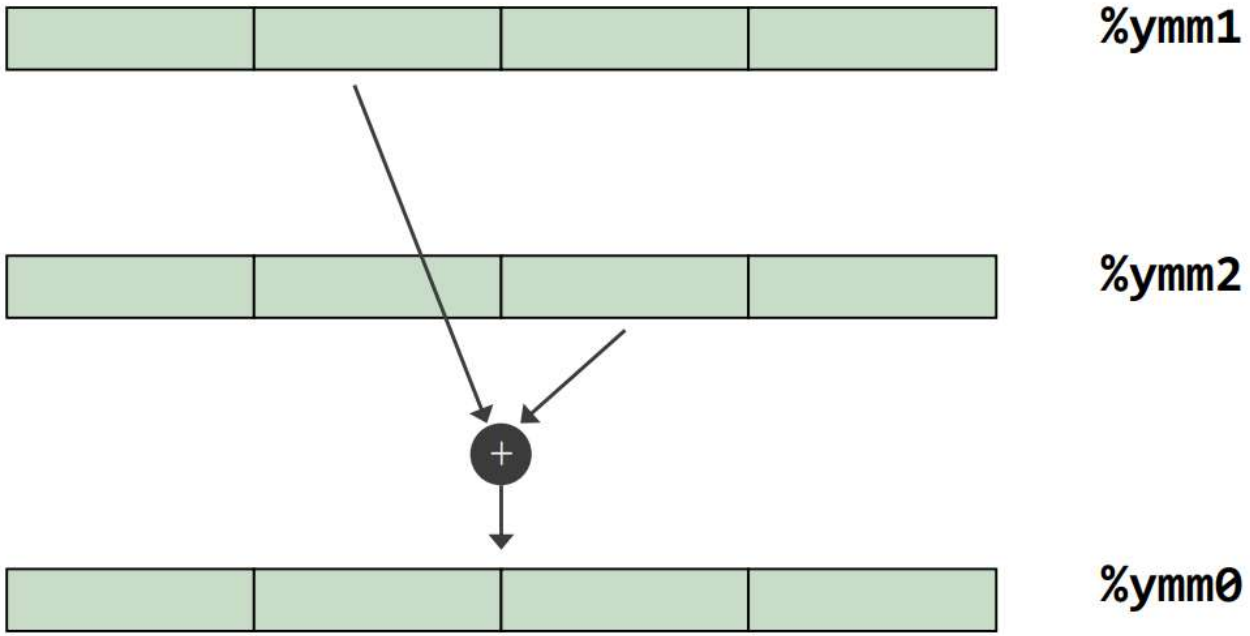
- *Set arch to AVX even for SSE code on SandyBridge and newer*

Franchetti: 18-613: Foundations of Computer Systems, Lecture 4. Based on Material by M. Püschel, ETH Zürich.

ITU CPH

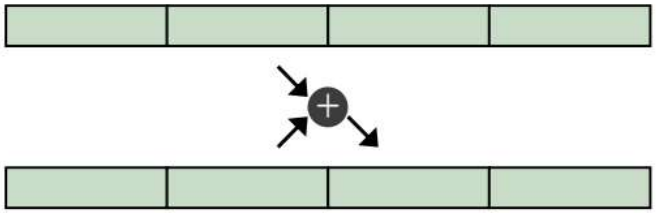
AVX Instructions: Examples

- Double precision *4-way vector add:* `vaddpd %ymm0, %ymm1, %ymm2`

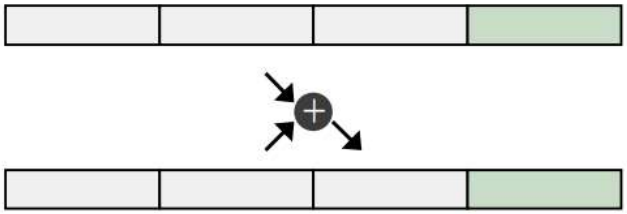


Since AVX: Intel has 3-operand instructions and added AVX-style SSE instructions

SSE/AVX: How to Take Advantage?



instead of



- **Necessary: fine grain parallelism**
- **Options (ordered by effort):**
  - Use vectorized libraries (easy, not always available)
  - Compiler vectorization (good option)
  - Use intrinsics (this lecture)
  - Write assembly
- **We will focus on 4-way (SSE single and AVX double)**

Multicore

ITU CPH

Two Laws of CS

30s

Moore's law

"... the observation that the number of transistors in a dense integrated circuit doubles approximately every two years."

UNIVERSITY OF COPENHAGEN

ITU CPH

Two Laws of CS

30s

Moore's law

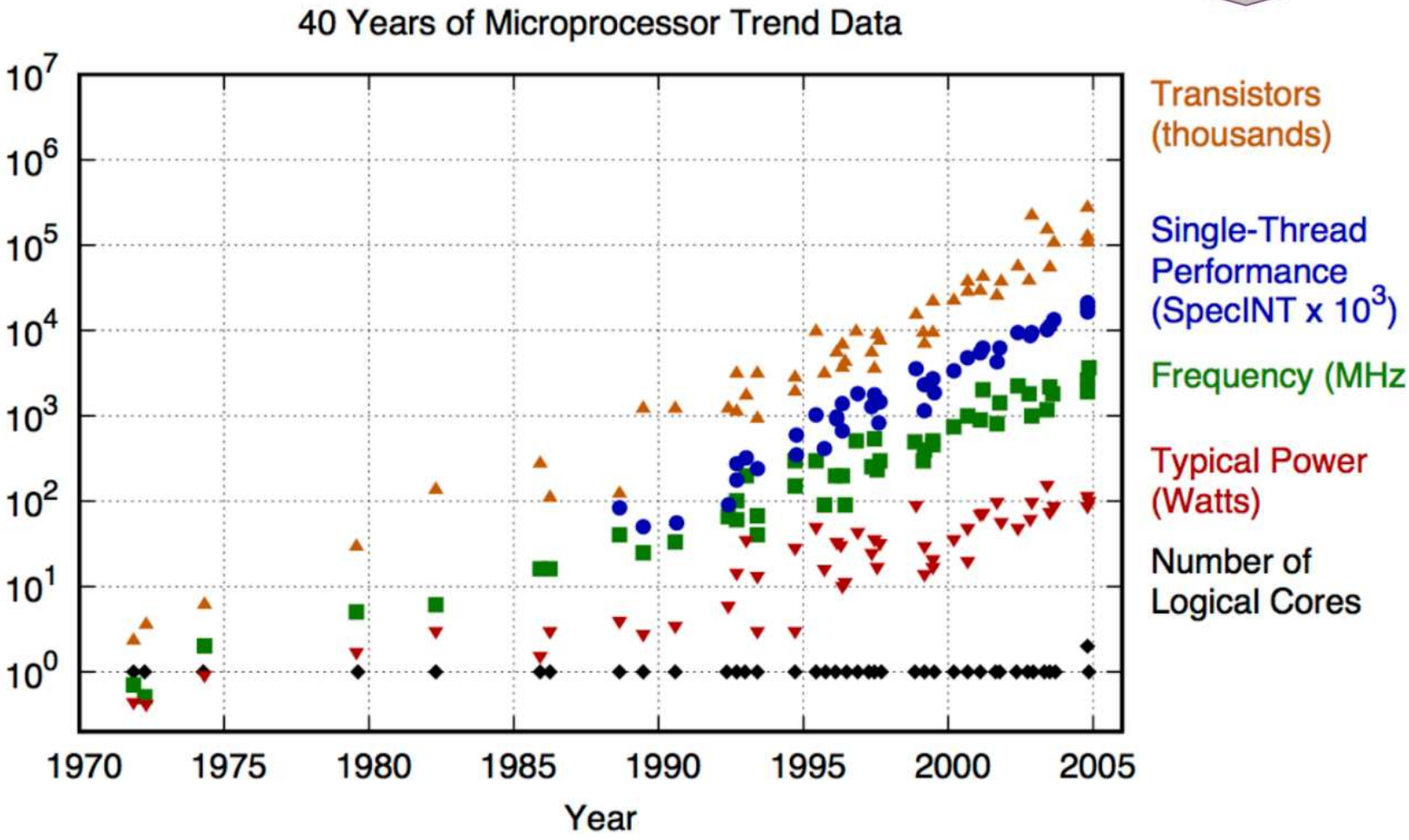
"... the observation that the number of transistors in a dense integrated circuit doubles approximately every two years."

Dennard scaling

" ... as transistors get smaller their power density stays constant, so that the power use stays in proportion with area."

Processor Trends - Before 2005

free lunch;
exponential scaling, w/ constant power draw.



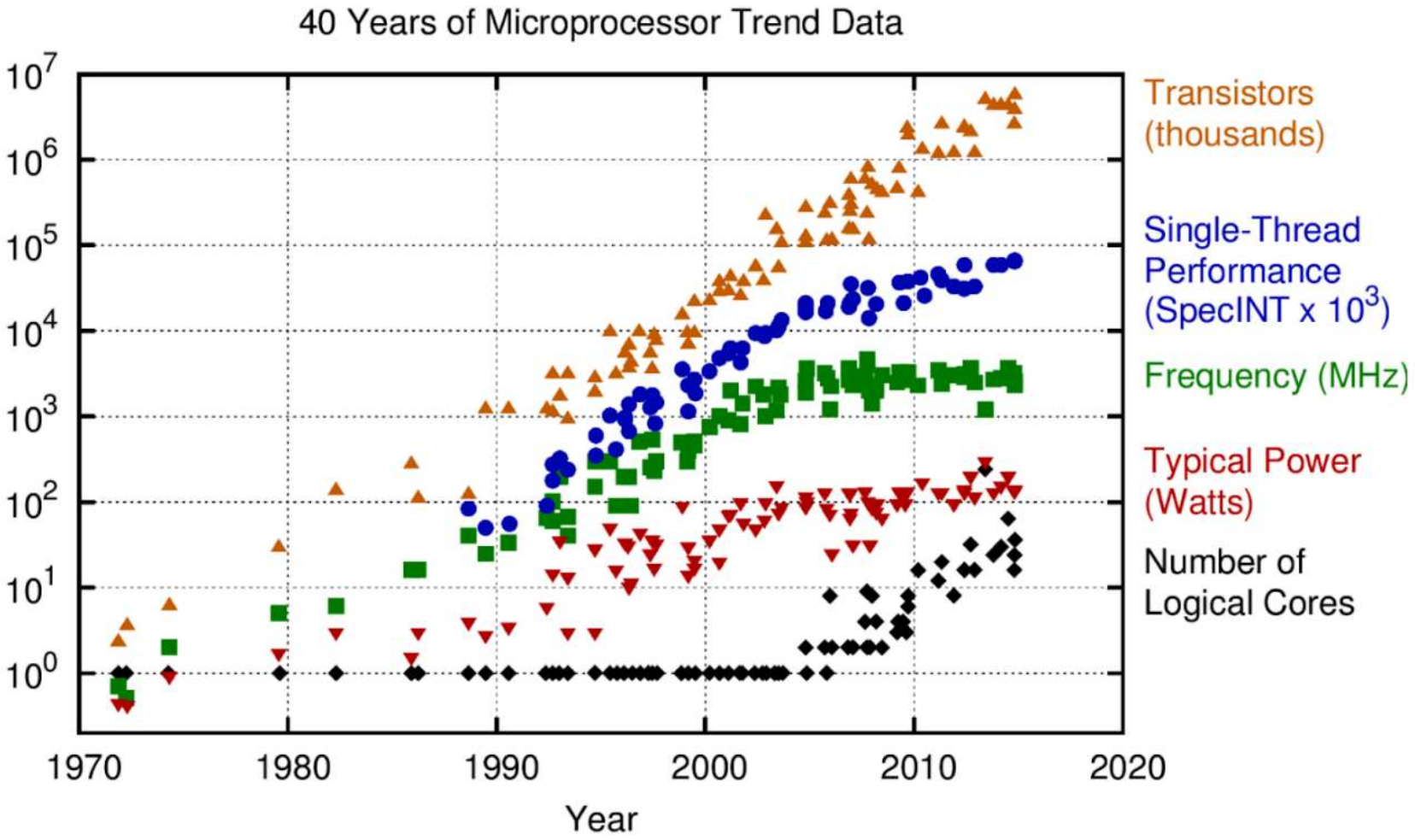
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

UNIVERSITY OF COPENHAGEN

18. ITU CPH

Processor Trends - After 2005

Dennard-scaling breaks.



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labo
New plot and data collected for 2010-2015 by K. Rupp

why: per-core not exponentially increasing;
only way to keep Moore's law:
increase number of cores.

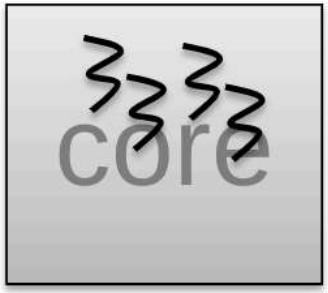
UNIVERSITY OF COPENHAGEN

ITU CPH

Processor Trends

2005

implicit parallelism



pipelining (ILP)

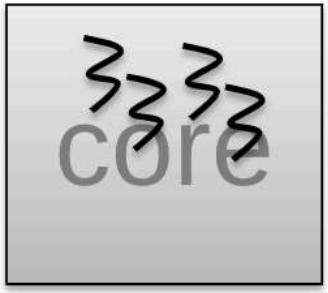
instruction-level parallelism

multithreading

Processor Trends

2005

implicit parallelism



pipelining (ILP) — instruction-level parallelism

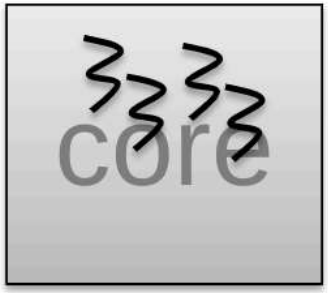
multithreading

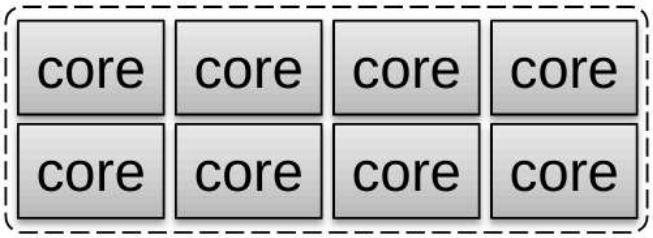
implicit parallelism → free lunch (almost)

Processor Trends

2005

implicit parallelism | **explicit parallelism**





pipelining (ILP)
multithreading

instruction-level parallelism

1 socket

multicores (CMP)

chip multiprocessor

implicit parallelism → free lunch (almost)

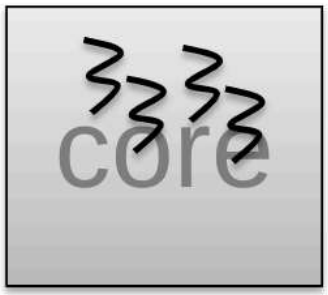
UNIVERSITY OF COPENHAGEN

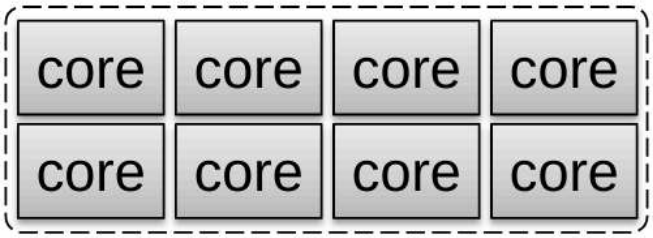
ITU CPH

Processor Trends

2005

implicit parallelism | **explicit parallelism**





pipelining (ILP)
multithreading

instruction-level parallelism

1 socket

multicores (CMP)

chip multiprocessor

implicit parallelism → free lunch (almost)
explicit parallelism → must work to exploit it (sometimes hard)

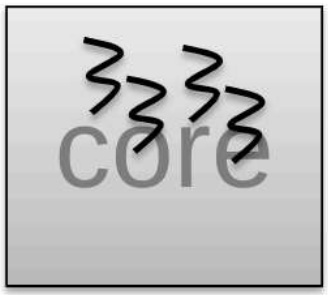
UNIVERSITY OF COPENHAGEN

ITU CPH

Processor Trends

2005

implicit parallelism

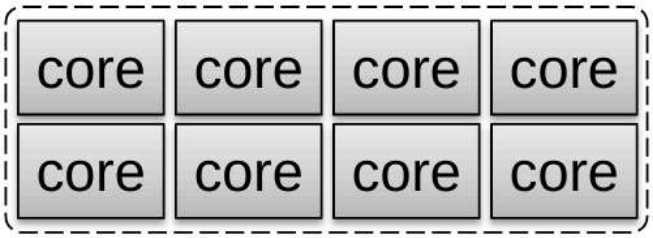


pipelining (ILP)

instruction-level parallelism

multithreading

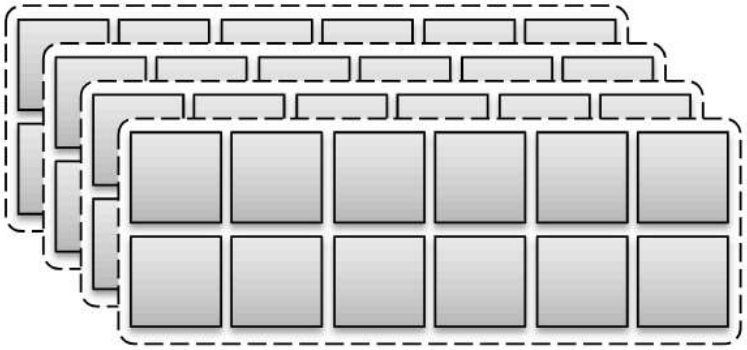
explicit parallelism



1 socket

multicores (CMP)

chip multiprocessor



multisocket multicores

difference in memory access
(uniform within socket, not across sockets)
(caching, next lecture)

implicit parallelism → free lunch (almost)

explicit parallelism → must work to exploit it (sometimes hard)

UNIVERSITY OF COPENHAGEN

ITU CPH

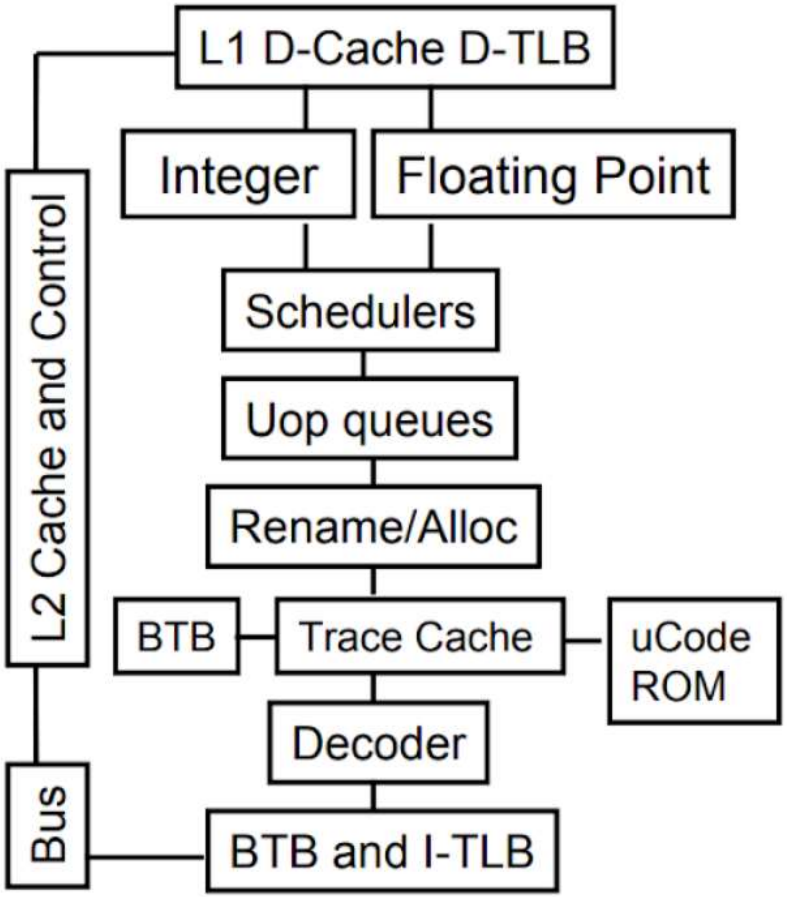
There's also SMT (Intel's **Hyperthreading**)

A technique complementary to multi-core:

Simultaneous multithreading

- Problem addressed: The processor pipeline can get stalled:
  - Waiting for the result of a long floating point (or integer) operation
  - Waiting for data to arrive from memory

Other execution units wait unused



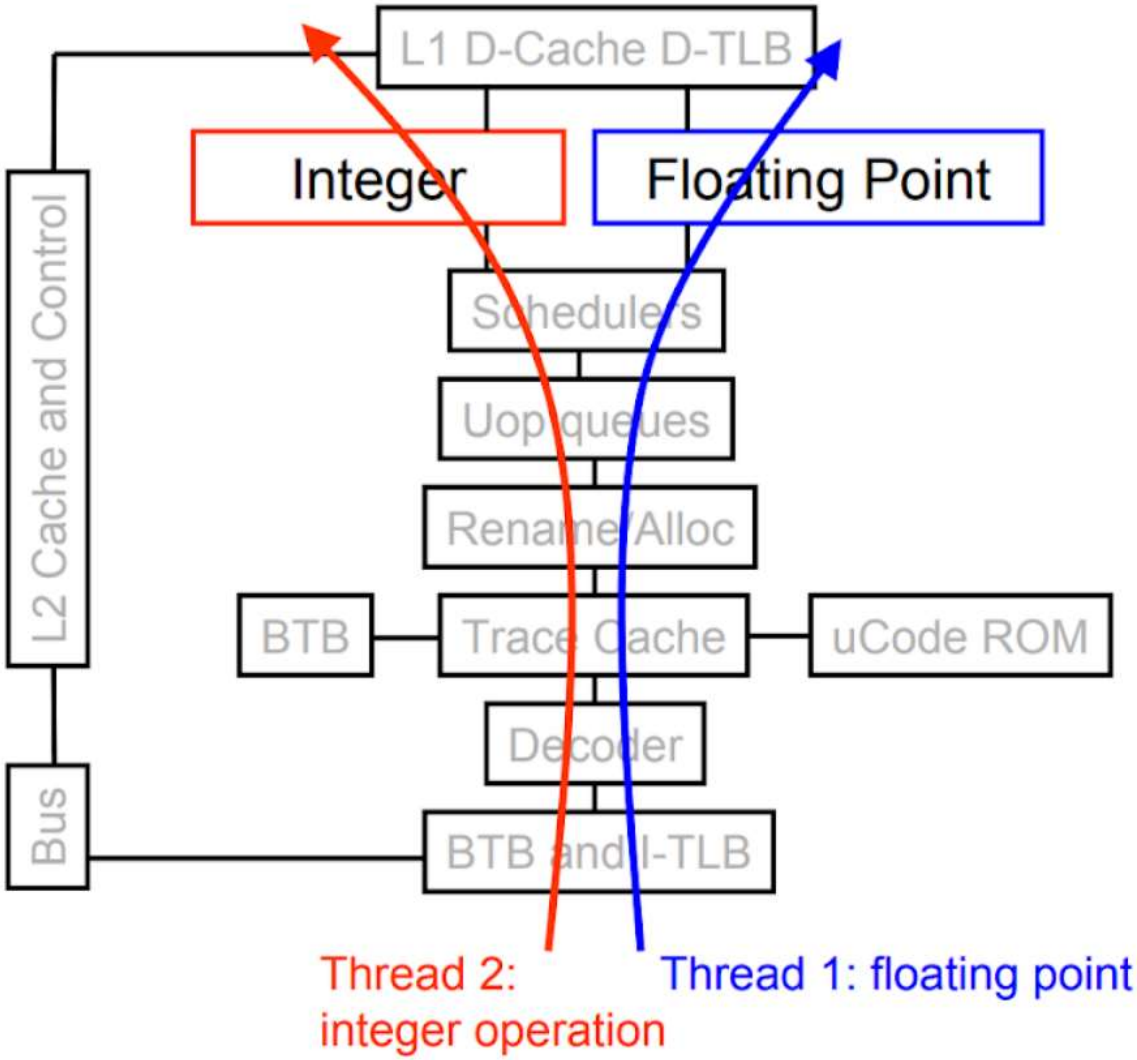
Source: Intel

17

ITU CPH

There's also SMT (Intel's **Hyperthreading**)

SMT processor: both threads can run concurrently



21

ITU CPH

There's also SMT (Intel's **Hyperthreading**)

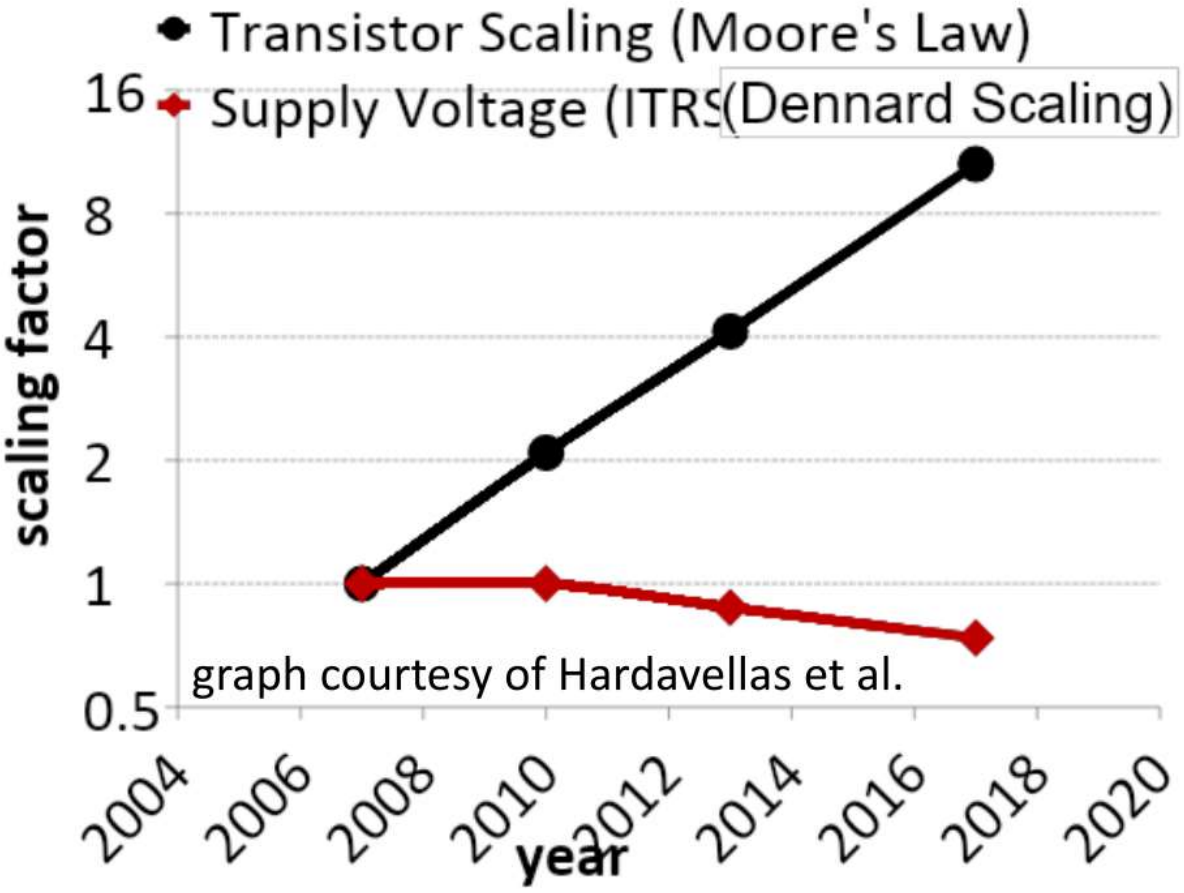
SMT not a "true" parallel processor

- Enables better threading (e.g. up to 30%)
- OS and applications perceive each simultaneous thread as a separate "virtual processor"
- The chip has only a single copy of each resource
- Compare to multi-core: each core has its own copy of resources

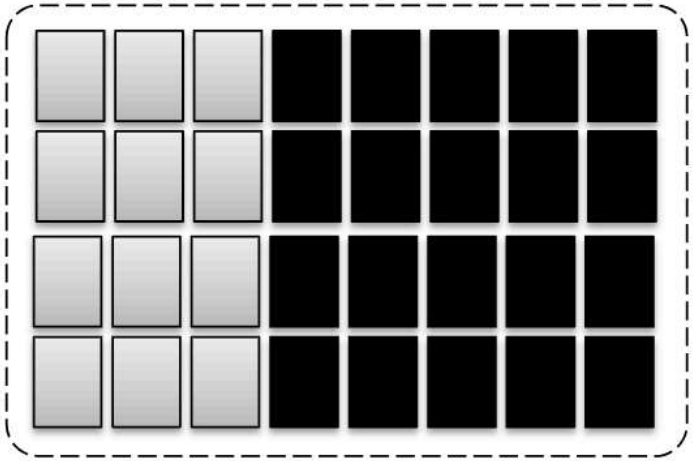
23

ITU CPH

Towards Dark Silicon



you have more cores than you can power.



you must schedule them. **new architectures** (Graphcore, Cerebras, …) no longer have this classic "CPU style".

heterogeneous systems: CPU + specialized hardware (FPGA, ASIC)

app-s

Can still pack more cores in a processor
Cannot fire all of them up simultaneously

UNIVERSITY OF COPENHAGEN

ITU CPH

un, fantastic, but) don't expect magic

Pitfall: Amdahl's Law

Execution time after improvement =

$$\frac{\text{affected execution time}}{\text{amount of improvement}} + \text{execution time unaffected}$$

$$T_{improved} = \frac{T_{affected}}{\text{improvement factor}} + T_{unaffected}$$

- adding more cores can lead to a speed-up…
- up to a point. eventually, $T_{unaffected}$ will dominate the other term in the sum.
  (anecdote: parallelizing mergesort)

takeaway: diminishing returns.

analogies:

- "too many cooks spoil the broth"
- project management joke:
  it takes 1 woman 9 months to grow 1 baby
  $\Rightarrow^{?}$ takes 9 women 1 months to grow 1 baby

embarrassingly parallel: task that can be divided cleanly and split off to threads of execution. (ideal)
reality: threads need to communicate & coordinate; incurs overhead.

UNIVERSITY OF COPENHAGEN

ITU CPH

slide by Hakim Weatherspoon, from CS 3410 course at Corn

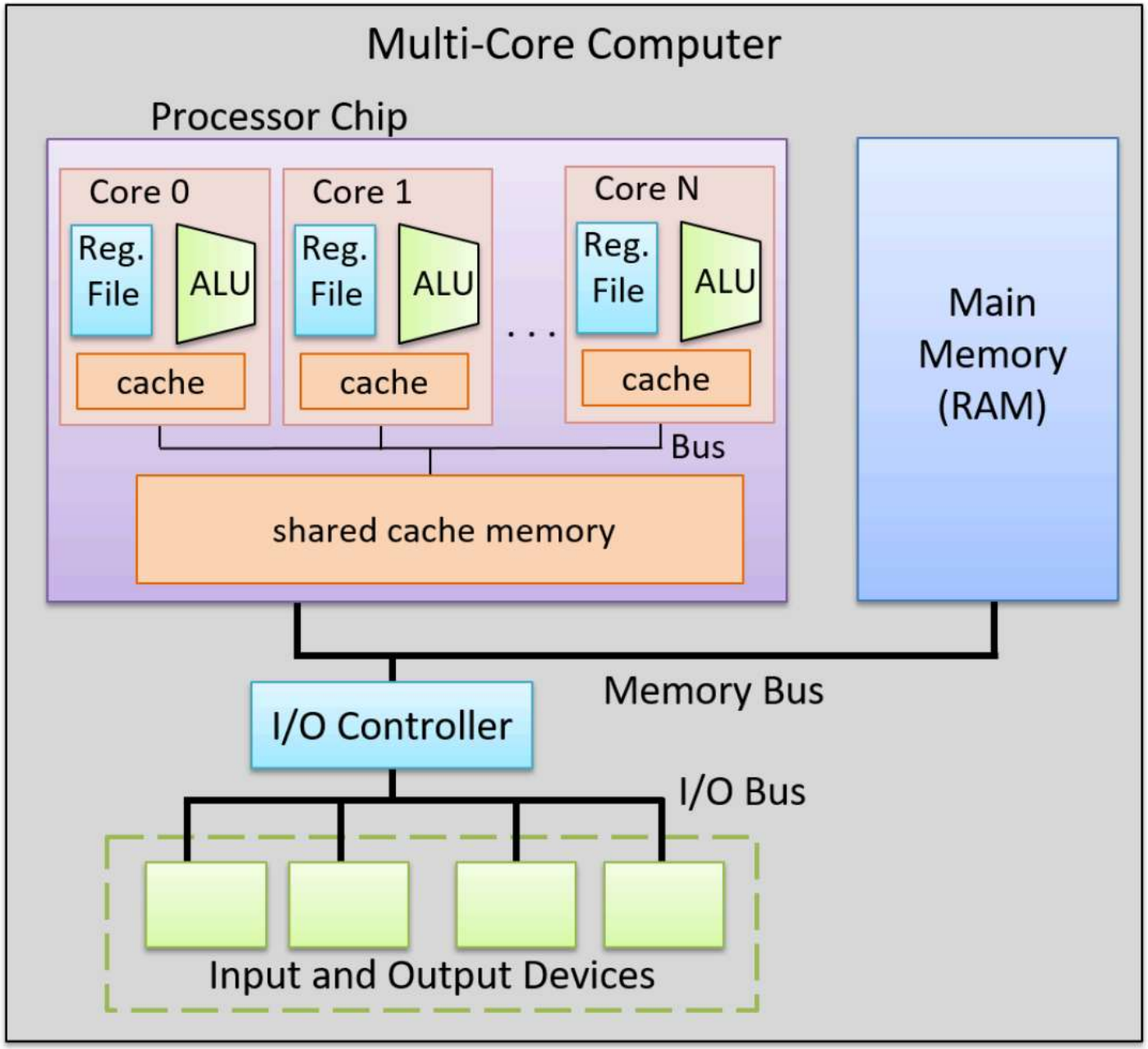


Figure 90. A computer with a multicore processor. The processor contains multiple complete CPU cores, each with its own private cache memory. The cores communicate with each and share a larger shared cached memory via on-chip buses.

ITU CPH

1. `LOCK` is not an instruction itself: it is an instruction prefix, which applies to the following instruction. That instruction must be something that does a read-modify-write on memory (`INC`, `XCHG`, `CMPXCHG` etc.) --- in this case it is the `incl (%ecx)` instruction which `inc`rements the `l`ong word at the address held in the `ecx` register.

   The `LOCK` prefix ensures that the CPU has exclusive ownership of the appropriate cache line for the duration of the operation, and provides certain additional ordering guarantees. This may be achieved by asserting a bus lock, but the CPU will avoid this where possible. If the bus is locked then it is only for the duration of the locked instruction.

2. This code copies the address of the variable to be incremented off the stack into the `ecx` register, then it does `lock incl (%ecx)` to atomically increment that variable by 1. The next two instructions set the `eax` register (which holds the return value from the function) to 0 if the new value of the variable is 0, and 1 otherwise. The operation is an **increment**, not an add (hence the name).

Share Improve this answer Follow

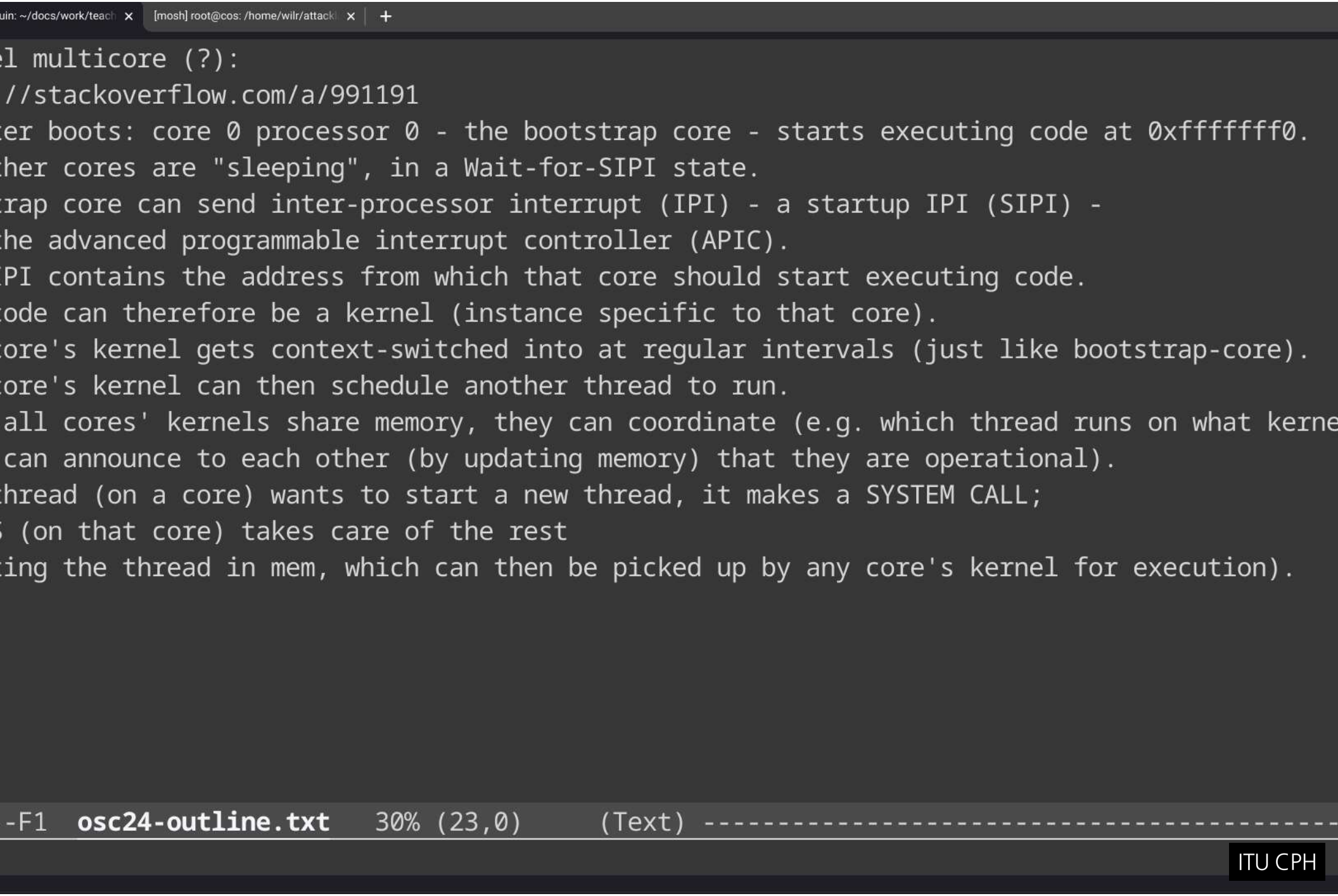
edited Jun 20, 2020 at 9:12
Community Bot
1 • 1

answered Jan 17, 2012 at 8:46

Anthony Williams
68.2k • 15 • 136 • 160

So the instuction "mov $0,%eax" seems redundant? – gemfield Jan 18, 2012 at 3:23

ITU CPH

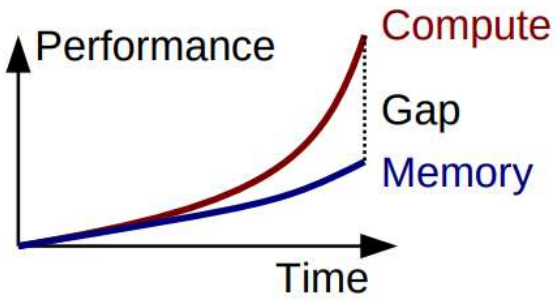

uin: ~/docs/work/teach
[mosh] root@cos: /home/wilr/attack
el multicore (?):
//stackoverflow.com/a/991191
ter boots: core 0 processor 0 - the bootstrap core - starts executing code at 0xfffffff0.
ther cores are "sleeping", in a Wait-for-SIPI state.
trap core can send inter-processor interrupt (IPI) - a startup IPI (SIPI) -
the advanced programmable interrupt controller (APIC).
IPI contains the address from which that core should start executing code.
code can therefore be a kernel (instance specific to that core).
core's kernel gets context-switched into at regular intervals (just like bootstrap-core).
core's kernel can then schedule another thread to run.
all cores' kernels share memory, they can coordinate (e.g. which thread runs on what kerne
can announce to each other (by updating memory) that they are operational).
thread (on a core) wants to start a new thread, it makes a SYSTEM CALL;
S (on that core) takes care of the rest
ting the thread in mem, which can then be picked up by any core's kernel for execution).
-F1 osc24-outline.txt 30% (23,0) (Text)
ITU CPH

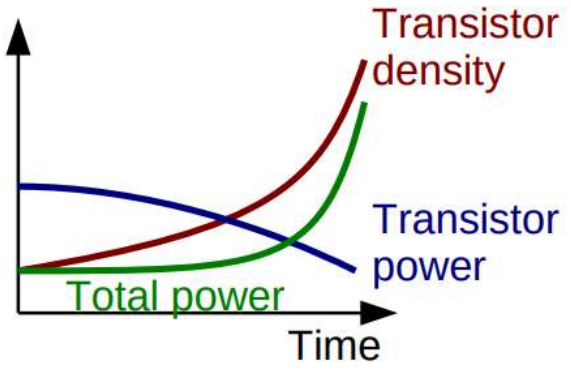
Heterogeneous Computing

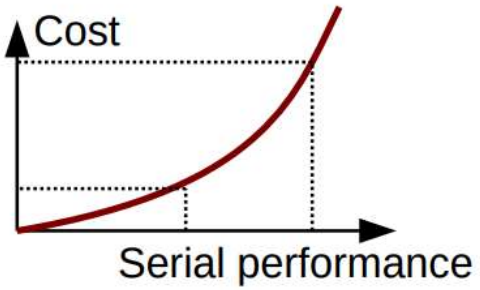
ITU CPH

Technology evolution

- Memory wall
  - Memory speed does not increase as fast as computing speed
  - Harder to hide memory latency
- Power wall
  - Power consumption of transistors does not decrease as fast as density increases
  - Performance is now limited by power consumption
- ILP wall
  - Law of diminishing returns on Instruction-Level Parallelism
  - Pollack rule: cost ≃ performance²







ITU CPH

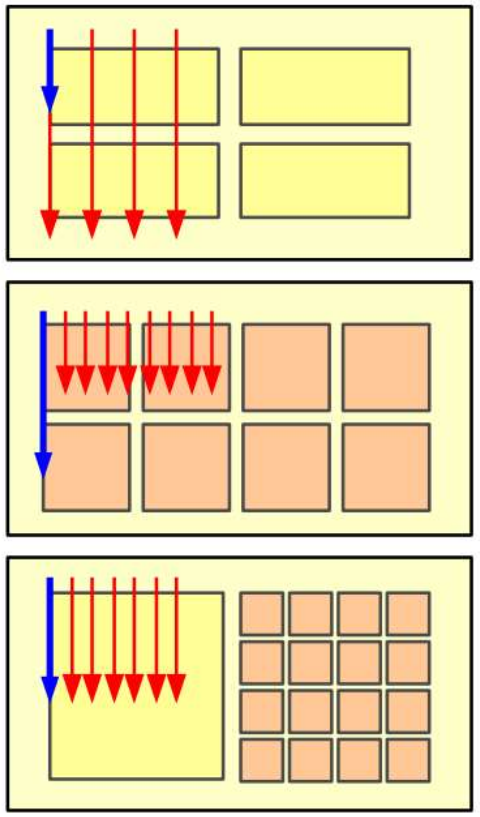
Why heterogeneous architectures?

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

Time to run sequential portions

Time to run parallel portions

- Latency-optimized multi-core (CPU)
  - Low efficiency on parallel portions: spends too much resources
- Throughput-optimized multi-core (GPU)
  - Low performance on sequential portions
- Heterogeneous multi-core (CPU+GPU)
  - Use the right tool for the right job
  - Allows aggressive optimization for latency **or** for throughput



M. Hill, M. Marty. Amdahl's law in the multicore era. IEEE Computer, 2008.

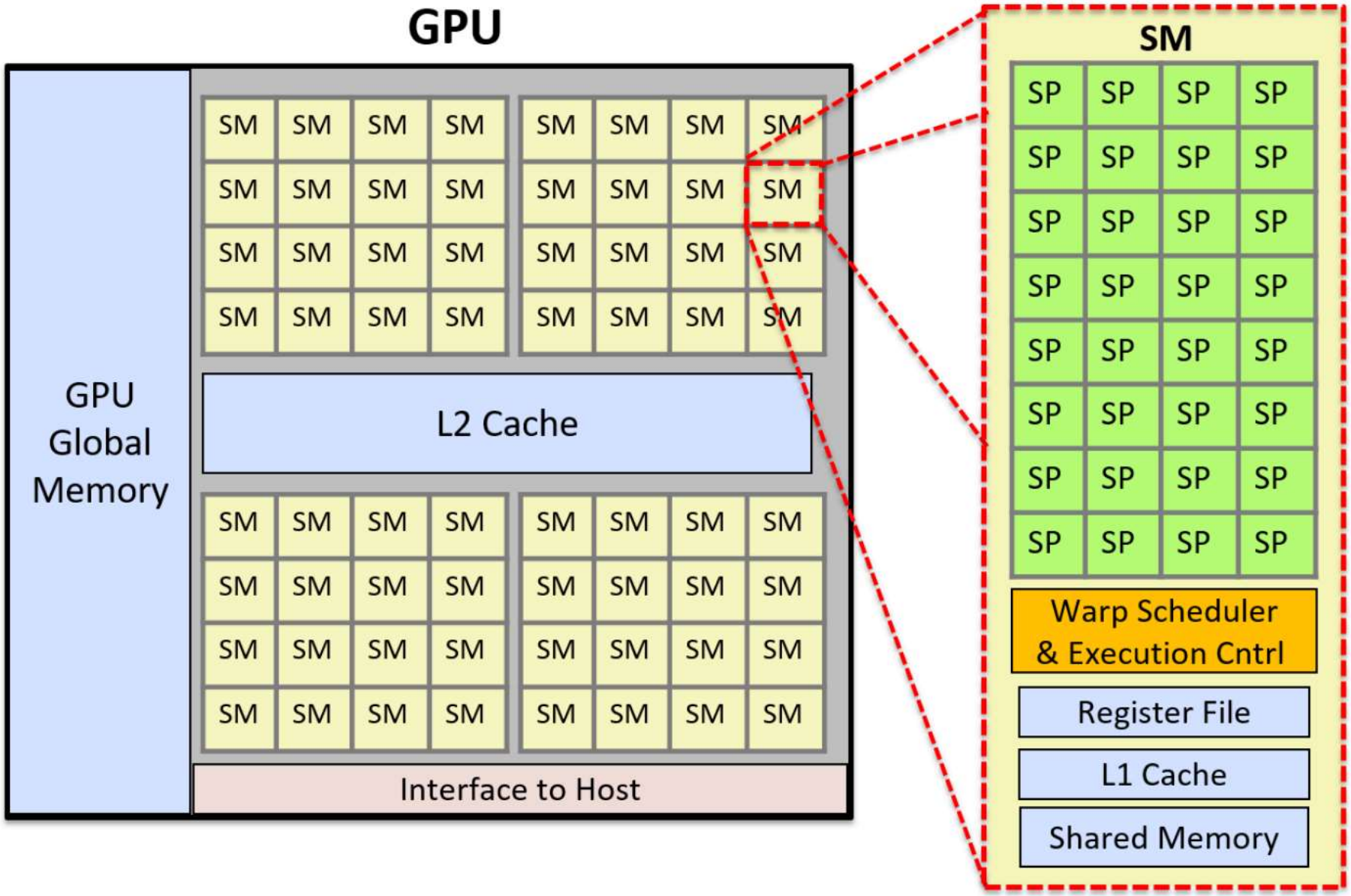


Figure 216. An example of a simplified GPU architecture with 2,048 cores. This shows the GPU divided into 64 SM units, and the details of one SM consisting of 32 SP cores. The SM's warp scheduler schedules thread warps on its SPs. A warp of threads executes in lockstep on the SP cores.

ITU CPH

5.1.2. GPU architecture overview

GPU hardware is designed for computer graphics and image processing. Historically, GPU development has been driven by the video game industry. To support more detailed graphics and faster frame rendering, a GPU device consists of thousands of special-purpose processors, specifically designed to efficiently manipulate image data, such as the individual pixel values of a two-dimensional image, in parallel.

The hardware execution model implemented by GPUs is **single instruction/multiple thread** (SIMT), a variation of SIMD. SIMT is like multithreaded SIMD, where a single instruction is executed in lockstep by multiple threads running on the processing units. In SIMT, the total number of threads can be larger than the total number of processing units, requiring the scheduling of multiple groups of threads on the processors to execute the same sequence of instructions.

As an example, NVIDIA GPUs consist of several streaming multiprocessors (SMs), each of which has its own execution control units and memory space (registers, L1 cache, and shared memory). Each SM consists of several scalar processor (SP) cores. The SM includes a warp scheduler that schedules **warps**, or sets of application threads, to execute in lockstep on its SP cores. In lockstep execution, each thread in a warp executes the same instruction each cycle but on different data. For example, if an application is changing a color image to grayscale, each thread in a warp executes the same sequence of instructions at the same time to set a pixel's RGB value to its grayscale equivalent. Each thread in the warp executes these instructions on a different pixel data value, resulting in multiple pixels of the image being updated in parallel. Because the threads are executed in lockstep, the processor design can be simplified so that multiple cores share the same instruction control units. Each unit contains cache memory and multiple registers that it uses to hold data as it's manipulated in lockstep by the parallel processing cores.

Figure 216 shows a simplified GPU architecture that includes a detailed view of one of its SM units. Each SM consists of multiple SP cores, a warp scheduler, an execution control unit, an L1 cache, and shared memory space.

ITU CPH

4. CUDA

(Compute Unified Device Architecture)[3] is NVIDIA's programming interface for GPGPU computing on its gra
s. CUDA is designed for heterogeneous computing in which some program functions run on the host CPU, a
run on the GPU device. Programmers typically write CUDA programs in C or C++ with annotations that spe
kernel functions, and they make calls to CUDA library functions to manage GPU device memory. A CUDA **ke
on** is a function that is executed on the GPU, and a CUDA **thread** is the basic unit of execution in a CUDA pr
hreads are scheduled in warps that execute in lockstep on the GPU's SMs, executing CUDA kernel code on t
stored in GPU memory. Kernel functions are annotated with `__global__` to distinguish them from host functi
`__device__` functions are helper functions that can be called from a CUDA kernel function.

emory space of a CUDA program is separated into host and GPU memory. The program must explicitly alloc
PU memory space to store program data manipulated by CUDA kernels. The CUDA programmer must either
tly copy data to and from the host and GPU memory, or use CUDA unified memory that presents a view of m
hat is directly shared by the GPU and host. Here is an example of CUDA's basic memory allocation, memory
cation, and explicit memory copy functions:

```
returns" through pass-by-pointer param dev_ptr GPU memory of size bytes
eturns cudaSuccess or a cudaError value on error

Malloc(void **dev_ptr, size_t size);
```

ITU CPH

Add a comment

There are in fact two different CUDA assembly languages.

PTX is a machine-independent assembly language that is compiled down to SASS, the actu
opcodes executed on a particular GPU family. If you build .cubins, you're dealing with SASS
CUDA runtime applications use PTX, since this enables them to run on GPUs released afte
original application.

Also, function pointers have been in CUDA for a while if you're targeting sm_20 (Fermi/GTX
series).

Share Follow

answered Sep 8, 2011 at 2

ChrisV

3,413 ● 17 ● 19

Add a comment

ITU CPH