Overview of Microprocessor Designs

Chun-Jen Tsai

NYCU

10/6/2023

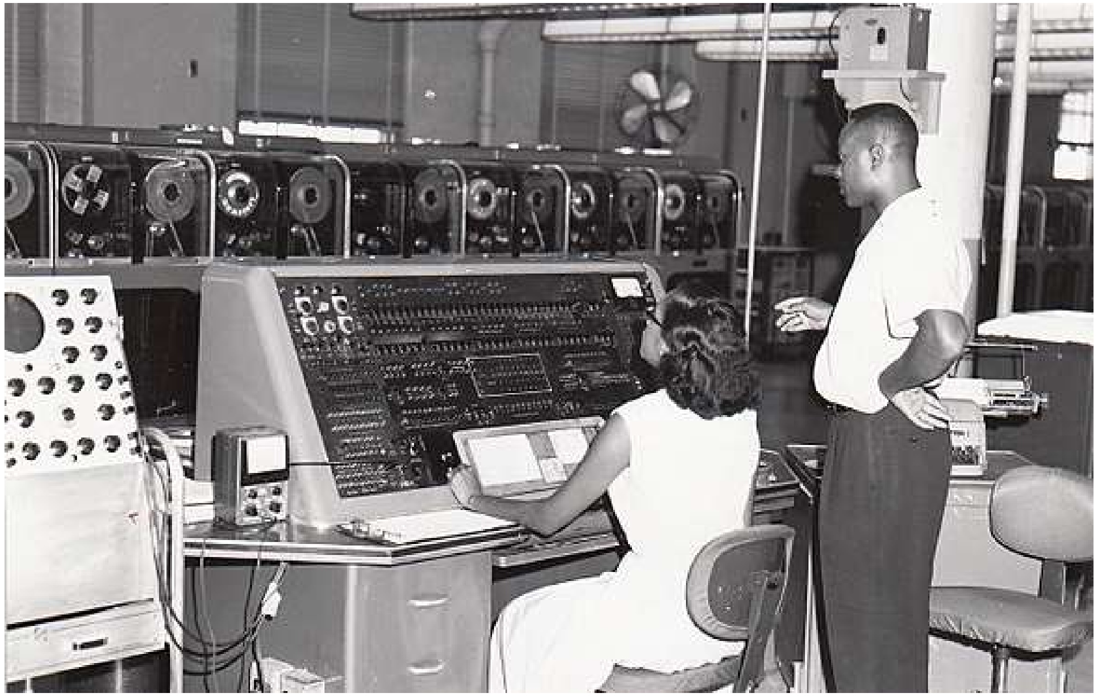
General Purpose Computers

- The first electronic general-purpose (programmable) computer is ENIAC
  - Built in 1946
  - Programmed by re-wiring functional boxes

- The first instruction-based computer is EDVAC
  - Realization of the “stored-program concept”
    - Proposed by Eckert et al.
    - Program bits stored in memory, just like data
  - EDVAC built in 1948 by configuring an ENIAC
  - von Neuman wrote the first ISA draft for EDVAC

UNIVAC I

- ❑ UNIVAC I is the first commercial ISA-based computer
  - Build by Eckert-Mauchly Computer Corporation in 1951
  - Performance: 0.0008 DMIPS/Mhz



† Picture taken by U.S. Census Bureau employees - https://www.census.gov/history/, Public Domain, https://commons.wikimedia.org/w/index.php?curid=61118834

From Architecture to Realization

- Architecture (a.k.a. ISA)
  - The design of instructions, addressing modes, and system interface of a microprocessor
- Implementation (a.k.a. microarchitecture)
  - The design of the circuit modules that implements the ISA
  - The high-level design description is mainly based on diagrams (hierarchical, timing, and interface diagrams, …, etc.)
  - For each bottom circuit module, the description are often synthesizable RTL code (Verilog, VHDL)
- Realization
  - Mapping of the synthesizable circuit description to a specific target technology (e.g., FPGAs or ASICs)

CISC vs RISC ISA

- In computer history, we have Complex Instruction Set Computer (CISC) before Reduced Instruction Set Computer (RISC)
  - What defines “complex” operation?
  - Why did we build “complex” computers before “simple” computers?
  - Which one is “better” (consider Intel i9 vs ARM Cortex A78)?
  - Does instruction set architecture design really matter?

Scalar ISA vs Vector ISA

- A vector ISA packs several numbers in a long bit-width register and perform the same operation on all data items concurrently, for example,

```
rv1 ← [a, b, c, d]; // 4 32-bit int, rv1 is of 128 bits
rv2 ← [e, f, g, h];
rv3 ← rv1 + rv2;    // [a+e, b+f, c+g, d+h]
```

  - Intel SSE, and the infamous AVX-512 instructions are vector extensions to the base scalar ISA
  - ARM defines scalar instructions in the base ISA, and vector instructions in the NEON coprocessor ISA

ISA Design

- ISA is not that important for modern computers
  - ISA designs have converged over several decades
    - Basic ISAs for different processors show little differences
    - Application-specific instructions are the main differentiators

- ISA does impact the microarchitecture complexity
  - The Dynamic/Static Interface[†] (DSI) determines what features of the processor are implemented in HW or SW
  - An operation can be implemented by compiler/programmer using instructions (dynamic) or wired into the processor (static)
    - Software interpreters and hardware microprogramming blurs the boundary of DSI

[†] S. W. Melvin and Y. N. Patt, “A clarification of the dynamic/static interface,” Proc. of th 12th Hawaii Int. Conf. on System Science, 1987.

DSI of a Processor

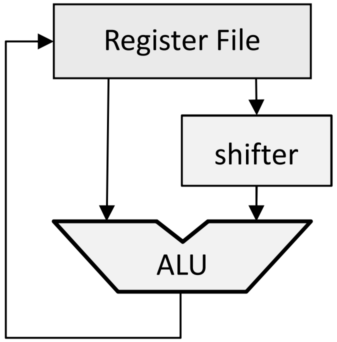
- ❑ DSI provides an important separation between the architecture and the implementation
  - ■ A CISC ISA places the DSI at the higher-level assembly language, and hardwired complex operations
  - ■ A RISC ISA lowers the DSI and expects to perform more of the optimizations above the DSI via the compiler

- ❑ DSI decides the microarchitecture features of a μP
  - ■ Java bytecode has an instruction for multi-dimensional memory allocation!
  - ■ Today, the DSI is chosen towards a simpler ISA architecture

Smart ISA Designs that Impact HW

- Instructions with additional barrel shifter
  - In ARM, two ALU ops in 1 cycle:

```
ADD r3, r2, r1, LSL#3  ; r3←r2+r1*8
```



- Conditional executions
  - In ARM, all ALU ops are conditional:

```
        CMP r0,#5
        BEQ Bypass    ;if (r0!=5)
        ADD r1,r1,r0  ;  r1←r1+r0
        SUB r1,r1,r2
Bypass   . . .
```

```
CMP   r0,#5
ADDNE r1,r1,r0
SUBNE r1,r1,r2
```

- Default branch direction bit
  - Compilers can encode the default jump direction in a loop
  - Example: PowerPC ISA

Smart ISA Designs Become Obsolete

- With new hardware design practices, these smart ISA designs may not be useful:
  - Barrel shifters along operand path hinder clock rates
  - Conditional execution still wastes CPU cycles
  - Default branch bits do not account for dynamic behavior

- In particular, these tricks are meaningless under superscalar microarchitecture
- For ISA design, simplicity is the king!

Design Objectives of Processors

- There are several design objectives of a μP:
  - Performance
  - Power
  - Cost
  - Reliability
  - Security

- Which one has more impact on achieving these goals: ISA or microarchitecture?

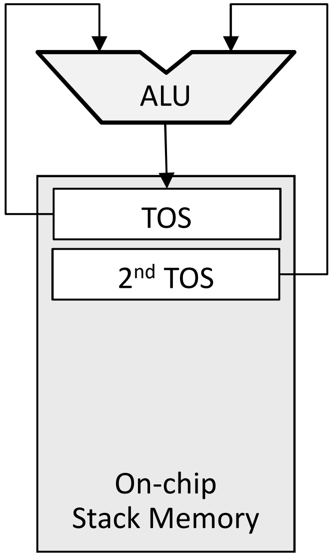
Pursuit of Performance

- ❑ The factors that affect the processor performance:

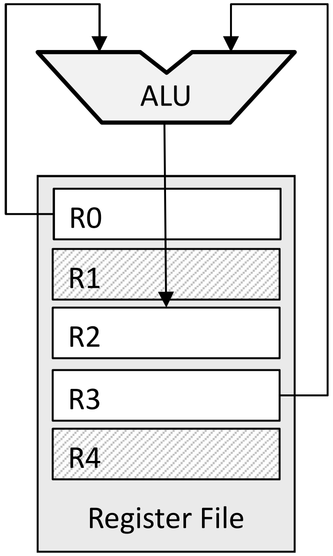
$$Exexution\ time = \underbrace{\frac{instruction}{program}}_{\text{Compiler \& OS}} \times \underbrace{\frac{cycle}{instruction}}_{\text{CPI: } \mu\text{Arch}} \times \underbrace{\frac{time}{cycle}}_{\mu\text{Arch \& fabrication tech.}}$$

Stack vs. Register Architecture

- A processor can adopt a stack- or register-based microarchitecture depending on where the intermediate results are stored:



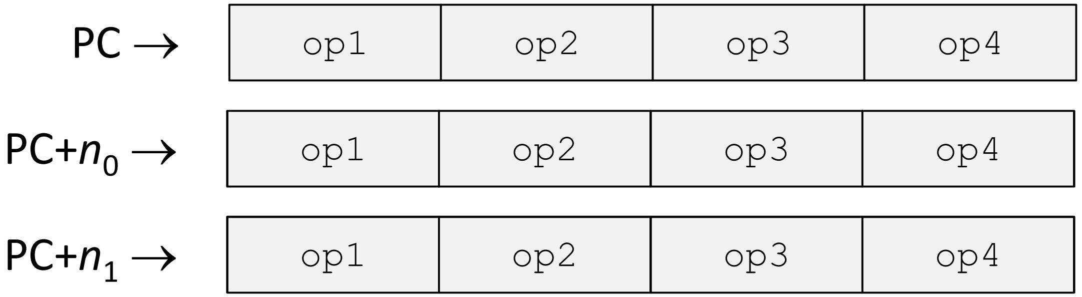
Stack-based datapath



Register-based datapath

VLIW Architecture

- Very Long Instruction Word (VLIW) ISA allows multiple operations to be packed into a bundle for execution:
  - Popular for DSP processors
  - Each bundle has a fixed number of “issue slots”
    - All operations in a bundle issued at the same clock cycle
  - Each issue slot encodes one instruction



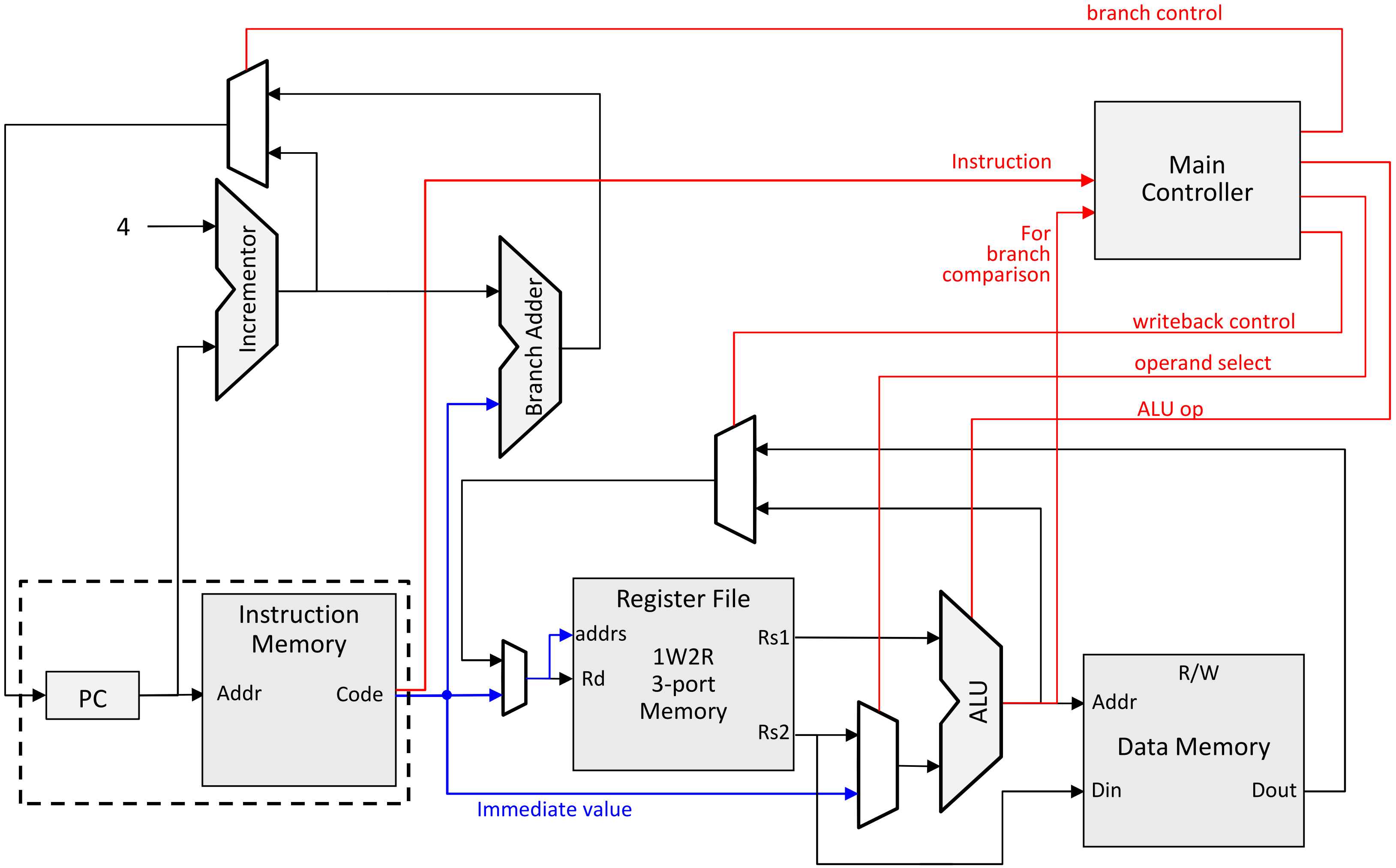
  - Easier to program by using “C intrinsics”

Scalar vs Superscalar Architecture

- Traditional scalar processors “execute” one instruction per clock cycle

- Superscalars execute multiple instructions per cycle:
  - Need multiple ALU functional units
  - Need multiple read/write buffers (registers)
  - Need internal queues for decoded instructions
  - Need control schemes to enforce data-dependency
  - Maybe in-order or out-of-order for instruction execution

1-Cycle Simple Scalar μArchitecture



Multiple Issue of Instructions

- ❑ Static multiple issue
  - ■ Typically for VLIW architecture
    - – Compiler/programmer groups instructions to be issued together
    - – Packages them into “issue slots”
    - – Compiler/programmer detects and avoids hazards
  - ■ VLIW compilers are very difficult to design

- ❑ Dynamic multiple issue
  - ■ CPU chooses instructions to issue each cycle
    - – Execution can be in-order or out-of-order
  - ■ Compiler can also help by reordering instructions
  - ■ CPU resolves hazards using advanced techniques at runtime

Speculative Execution of Instructions

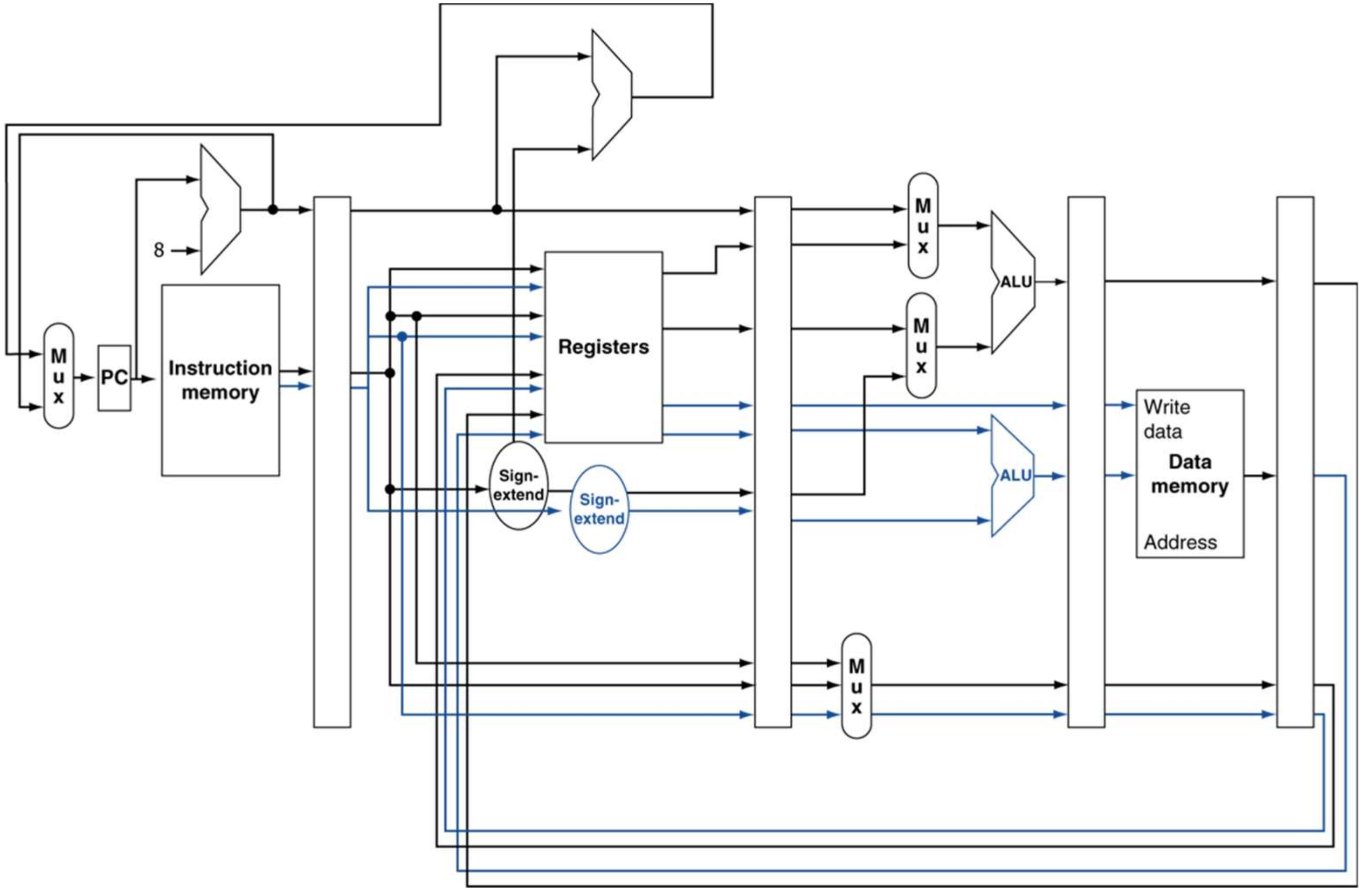
- Speculate on branch
  - Execute a branch instruction before the outcome is known
  - Roll back if path taken is different

- Speculate on data loading
  - Execute an indirect load before the address is determined
  - Pointer in register may be wrong due to out-of-order execution
  - Roll back if location is updated

- In general, can look ahead for instructions to execute
  - Buffer results until they are actually needed (for writeback)
  - Flush buffers on incorrect speculation

Static Double-Issue μArchitecture

- Two instructions will be concurrently executed:
  - Only one of them can be load/store



Dynamic Multiple Issue

- Superscalar processors
  - Instruction “grouping” is done by the CPU control logic
  - Execution can be in-order or out-of-order
  - Avoiding structural and data hazards

- Avoids the need for compiler scheduling
  - Compilers may still help
    - CPU analysis windows are small
    - Roll-backs are expensive
  - Code semantics ensured by the CPU
    - CPU has to ensure the results are precisely what the code says

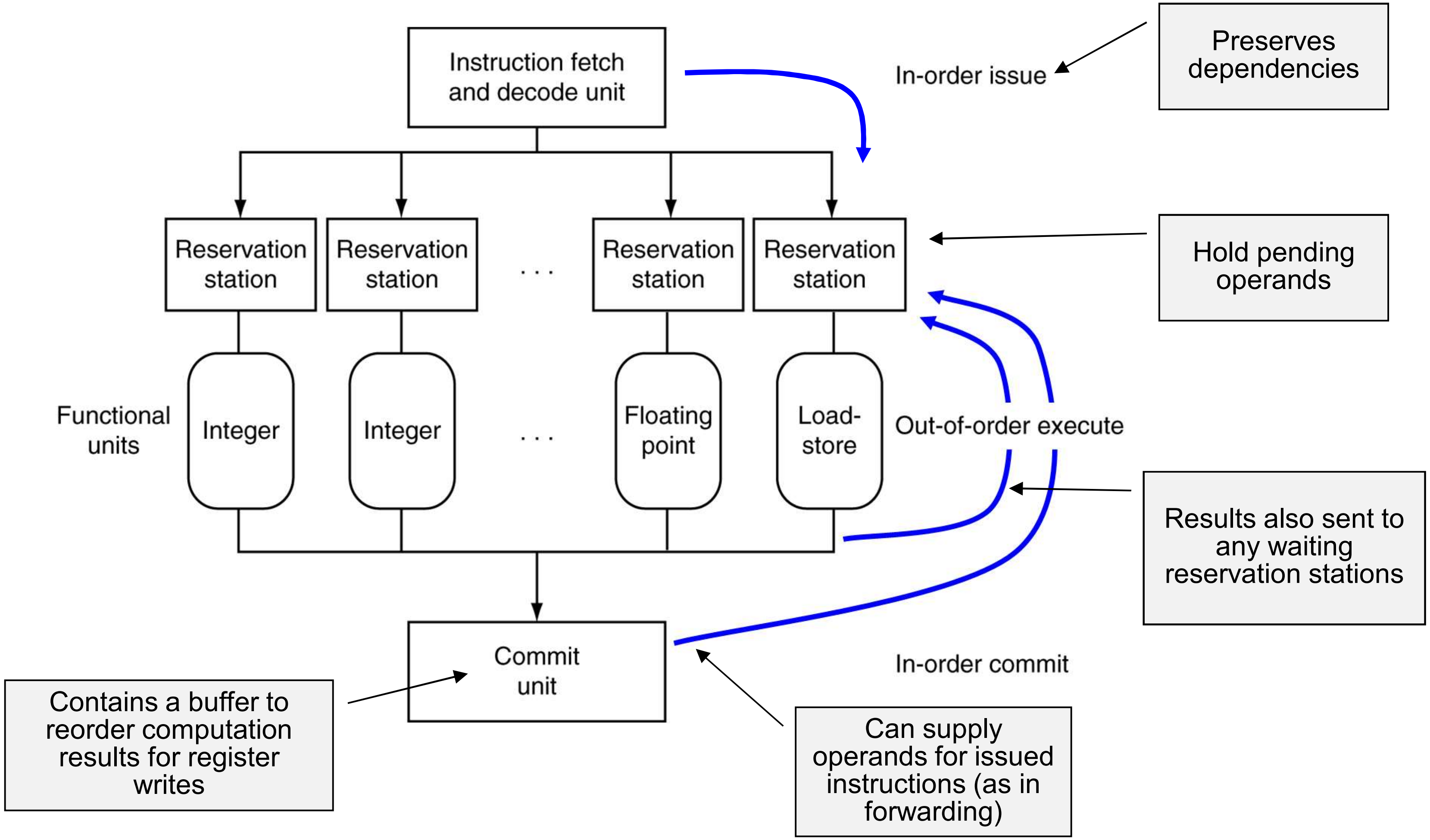
Dynamic Pipeline Scheduling

- ❑ Allow the CPU to execute instructions out-of-order to avoid stalls
  - Issue instructions in order
  - Execute instructions out of order
  - Commit results to registers in order

- ❑ Example: CPU can start `sub` and `slti` while `add` is waiting for `lw`

```
lw    $t0, 20($s2)
add   $t1, $t0, $t2
sub   $s4, $s4, $t3
slti  $t5, $s4, 20
```

Dynamically Scheduled CPU



Preserves dependencies

Hold pending operands

Results also sent to any waiting reservation stations

Contains a buffer to reorder computation results for register writes

Can supply operands for issued instructions (as in forwarding)

Register Renaming

- Typical ISA has a limited amount of registers (e.g. 32)
  - Architectural register number is limited by instruction coding
  - CPU can have more physical registers
  - An architectural register can be renamed to a physical register

- Reservation stations and reorder buffer effectively provide register renaming upon instruction issue
  - If operand is available in register file or reorder buffer
    - Copied to a reservation station
    - No longer required in the register; can be overwritten
  - If operand is not yet available
    - It will be provided to the reservation station by a functional unit
    - Register update may not be required

Speculation in Reservation Station

- Branch speculation
  - Predict branch and continue issuing
  - Don’t commit results until branch outcome determined

- Load speculation
  - Avoid load and cache miss delay
    - Predict the effective address
    - Predict loaded value
    - Load before completing outstanding stores
    - Bypass stored values to load unit
  - Don’t commit load until speculation cleared

Name-Dependence Hazards

- ❑ Name hazards of out-of-order, multiple-issue execution:
  - ■ Two instructions use the same register name/memory location
- ❑ Data and name dependencies are classified as follows:
  - ■ Write-After-Read (WAR) hazard: an anti-dependence of name

```
add   $t1, $t2, $t3
add   $t3, $t0, $t1
```

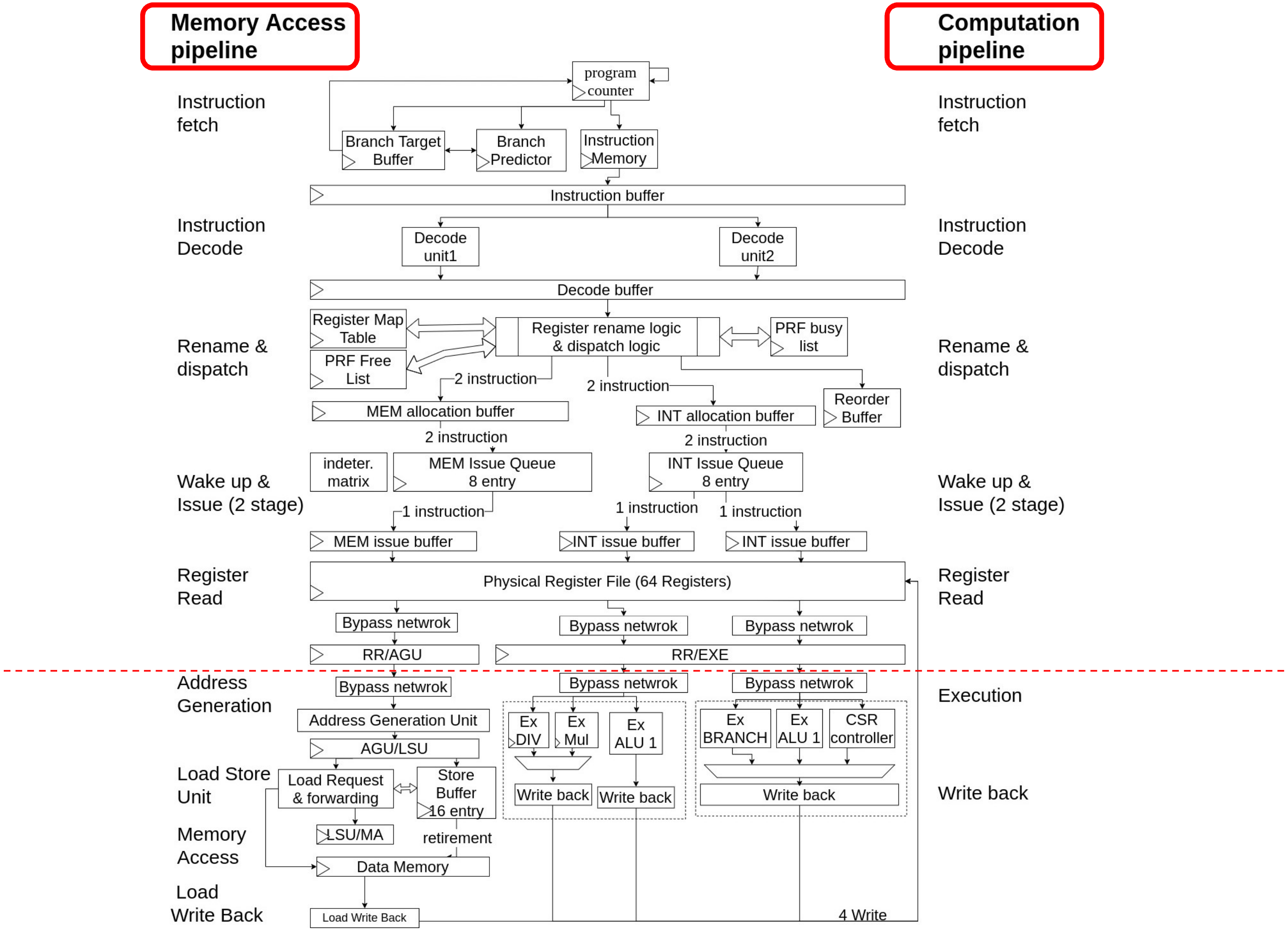
  - ■ Write-After-Write (WAW) hazard: an output name dependence

```
add   $t3, $t2, $t1
add   $t3, $t0, $t1
```

  - ■ Read-After-Write (RAW) hazard: a true data dependence

```
add   $t3, $t0, $t1
add   $t1, $t2, $t3
```

Superscalar Example (Falco)



Memory Access pipeline

Computation pipeline

Instruction fetch

program counter

Branch Target Buffer

Branch Predictor

Instruction Memory

Instruction buffer

Instruction Decode

Decode unit1

Decode unit2

Decode buffer

Rename & dispatch

Register Map Table

PRF Free List

Register rename logic & dispatch logic

PRF busy list

2 instruction

2 instruction

Reorder Buffer

MEM allocation buffer

INT allocation buffer

2 instruction

2 instruction

Wake up & Issue (2 stage)

indeter. matrix

MEM Issue Queue 8 entry

INT Issue Queue 8 entry

1 instruction

1 instruction

1 instruction

MEM issue buffer

INT issue buffer

INT issue buffer

Register Read

Physical Register File (64 Registers)

Bypass netwrok

Bypass netwrok

Bypass netwrok

RR/AGU

RR/EXE

Address Generation

Execution

Bypass netwrok

Bypass netwrok

Bypass netwrok

Address Generation Unit

Ex DIV

Ex Mul

Ex ALU 1

Ex BRANCH

Ex ALU 1

CSR controller

AGU/LSU

Load Store Unit

Load Request & forwarding

Store Buffer 16 entry

Write back

Write back

Write back

Write back

Memory Access

LSU/MA

retirement

Data Memory

Load Write Back

Load Write Back

4 Write

Evolution History of ARM μArch.

- ❑ Acorn Computers, established in 1978
  - ■ ARM1 (1985) → … → ARM3 (1989)

- ❑ ARM, established in 1990
  - ■ Classical ARMs:
    - – ARM6 (1992) → ARM7 (1993) → ARM9 (1998) → ARM11 (2002)
    - – ARM6 was used in the Apple Newton PDA (1993)
  - ■ Cortex ARMs:
    - – Cortex A8(2005) → … → Cortex A78(2020) → Cortex A710(2021)
    - – Cortex M3(2004) → … → Cortex M55(2020)
    - – Cortex X1(2020) → Cortex X2(2021)

Pipeline Stage Design Choices

- In previous design, the “processor state” are stored in the PC and the register file
  - A 2-stage pipeline seems a natural implementation (assuming the instruction/data memory being latch-like combinational memory devices)
  - A synchronous memory would be a “state” element too
- Back in 1980’s, it is customary to cut the processor pipeline into three stages
  - Example: ARM 7 (ARMv4 architecture)
- Modern in-order execution RISC processors contains at least five pipeline stages

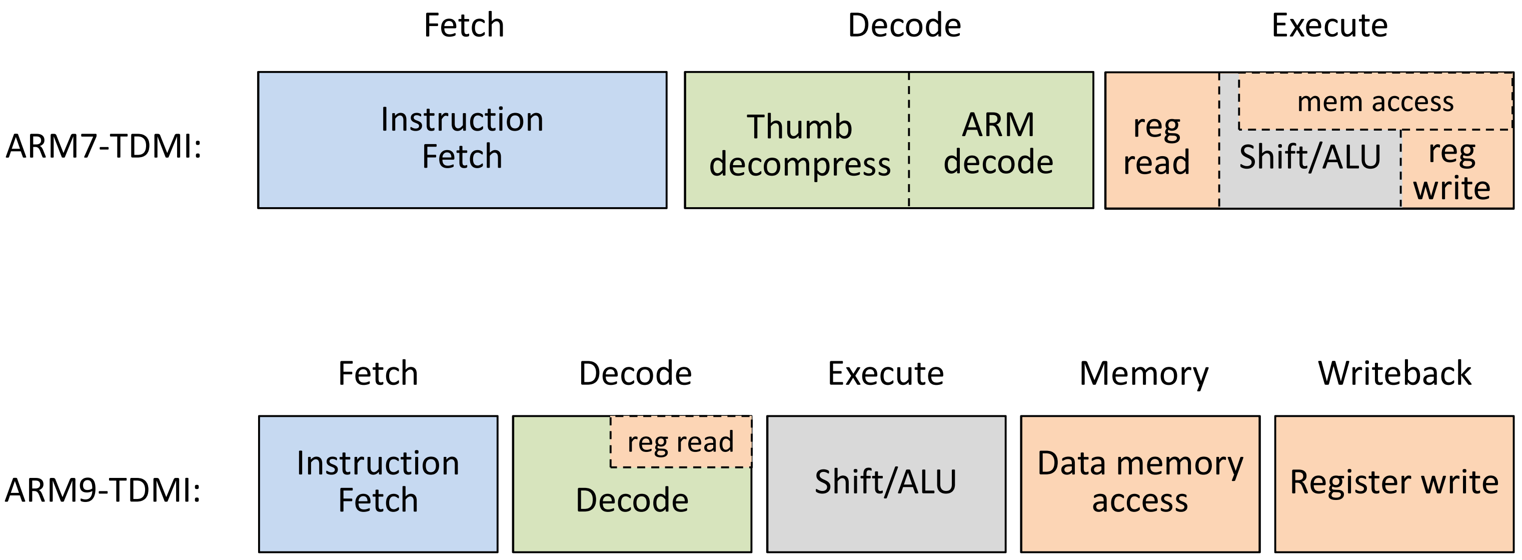
Classical RISC Pipeline Stages

- The execution of an instruction has several steps
  - Not all instructions require the same number of steps – a big problem for CISC processors

- For a RISC processor the classical steps are:
  - Fetch – fetch instruction from instruction memory
  - Decode – decode instruction to generate control signal
  - Execute – execute ALU operations
  - Memory – access data memory
  - Writeback – write results back to register file

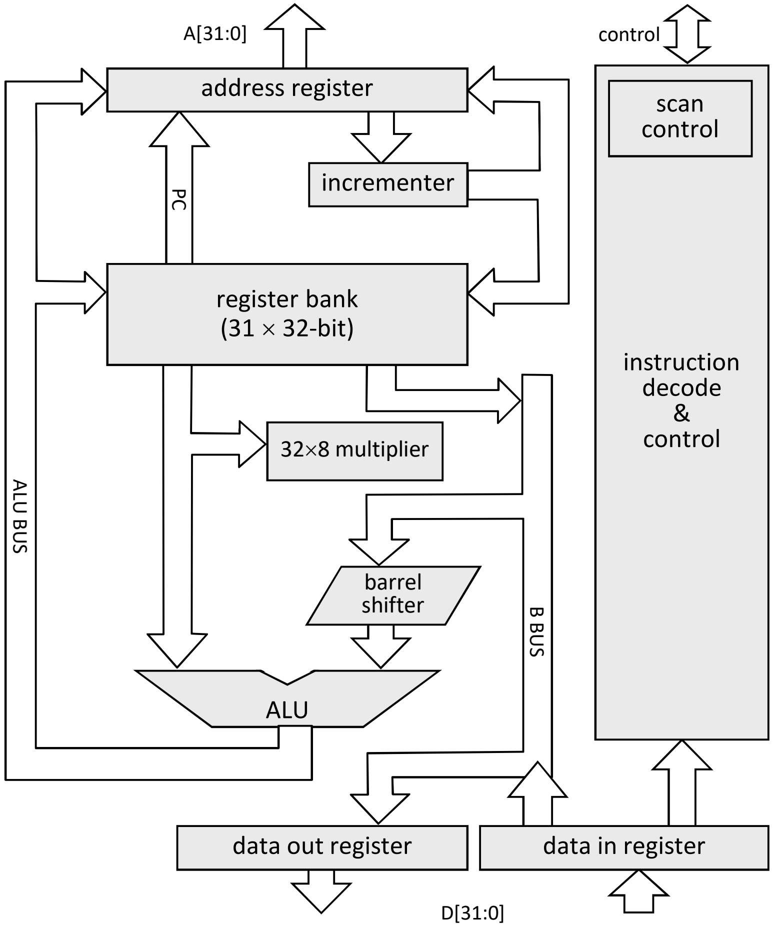
ARM Cores: 3-Stage vs. 5-Stage

- Pipeline stage organizations of a 3-stage ARM7 versus a 5-stage ARM9:



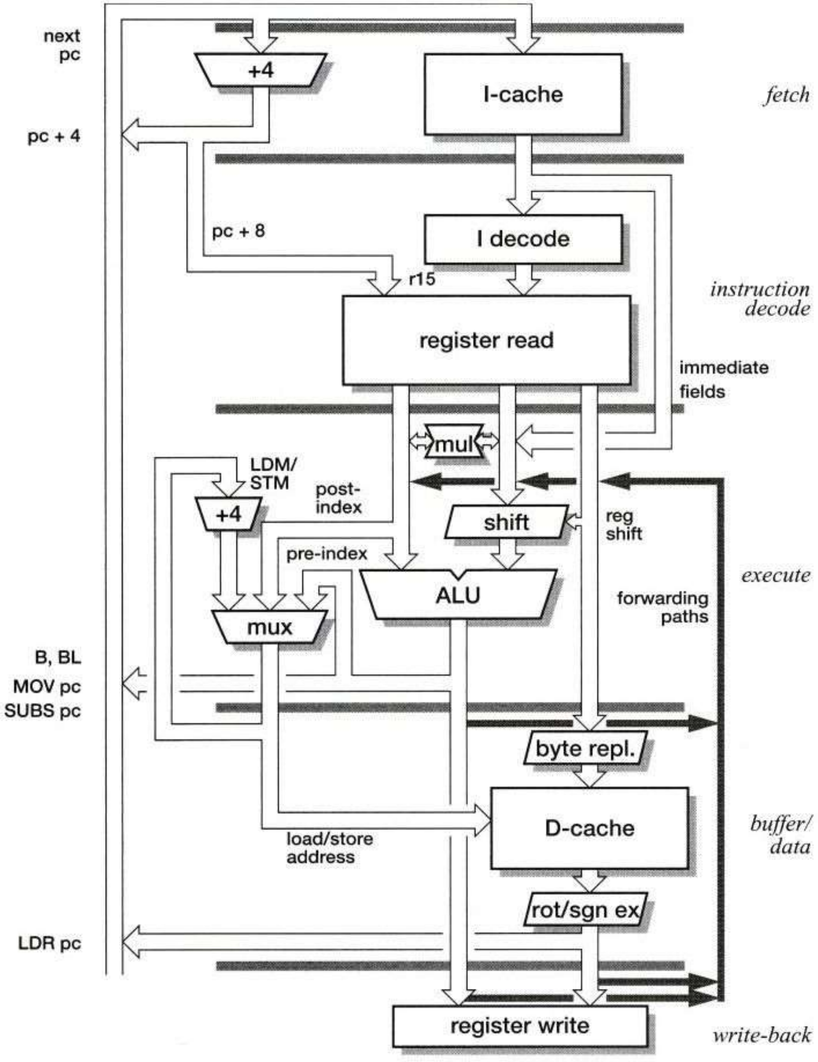
Classical ARM 7 μArchitecture

- ❑ One of the most famous three-stage pipeline RISC
  - ARM ISA v4T
  - Von Neumann architecture
  - ~ 80% market share in 2G cell phones
  - The predecessor of ARM Cortex-M μArchitecture
  - Used in the original iPod (customized edition)



ARM 9 μArchitecture

- First 5-stage pipeline from ARM
- Harvard architecture
- Add facilities to remove pipeline hazard

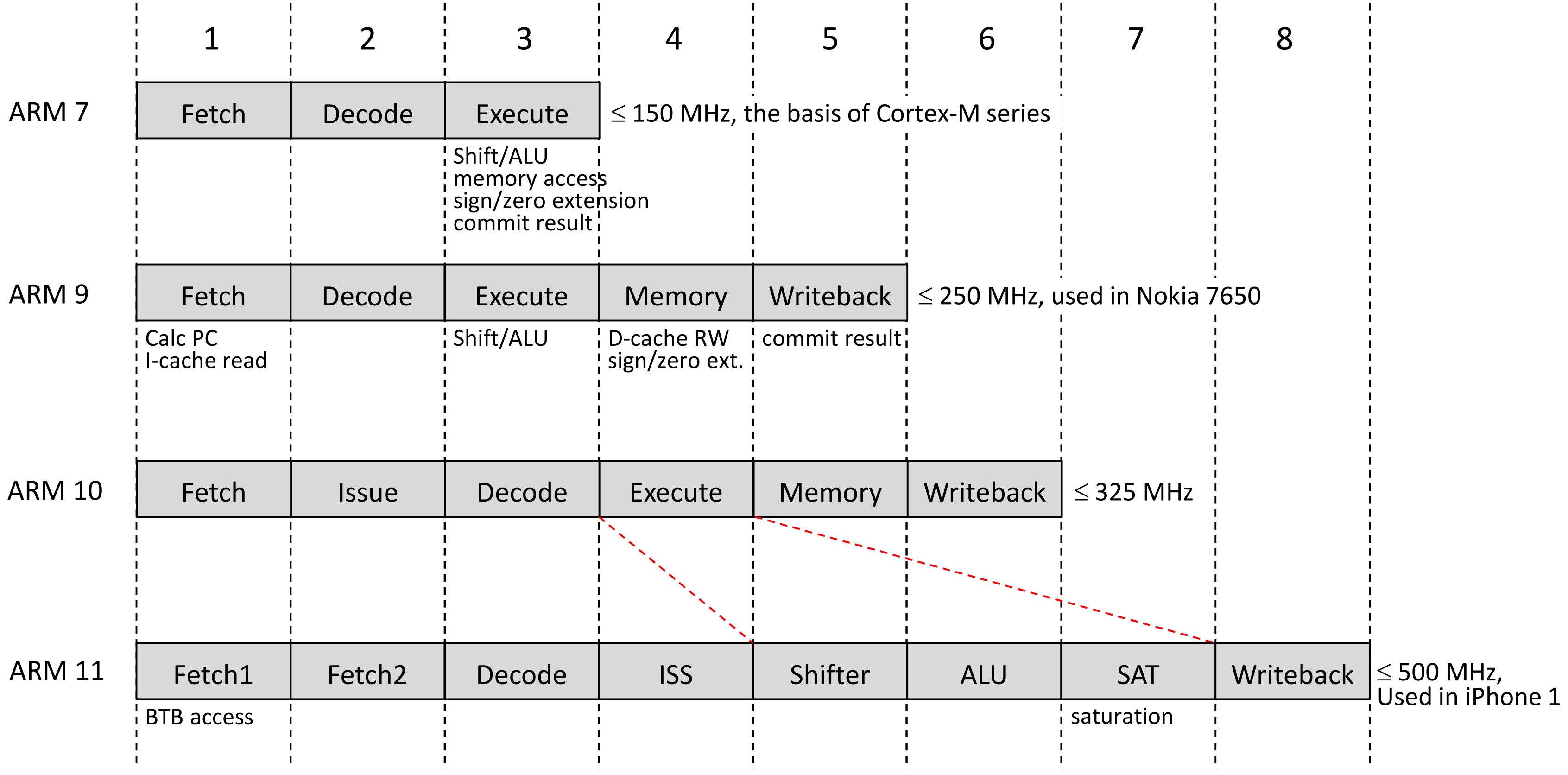


ARM 11 μArchitecture

- ❑ The final version of the classical ARMs
- ❑ Key μArchitecture features:
  - ■ In-order execution, out-of-order completion (certain ops)
  - ■ Non-blocking cache
  - ■ Parallel ALUs
  - ■ Parallel Loads/Stores

Classical ARM Pipeline Evolutions

- ❑ ARM classical pipelines before Cortex-A series:



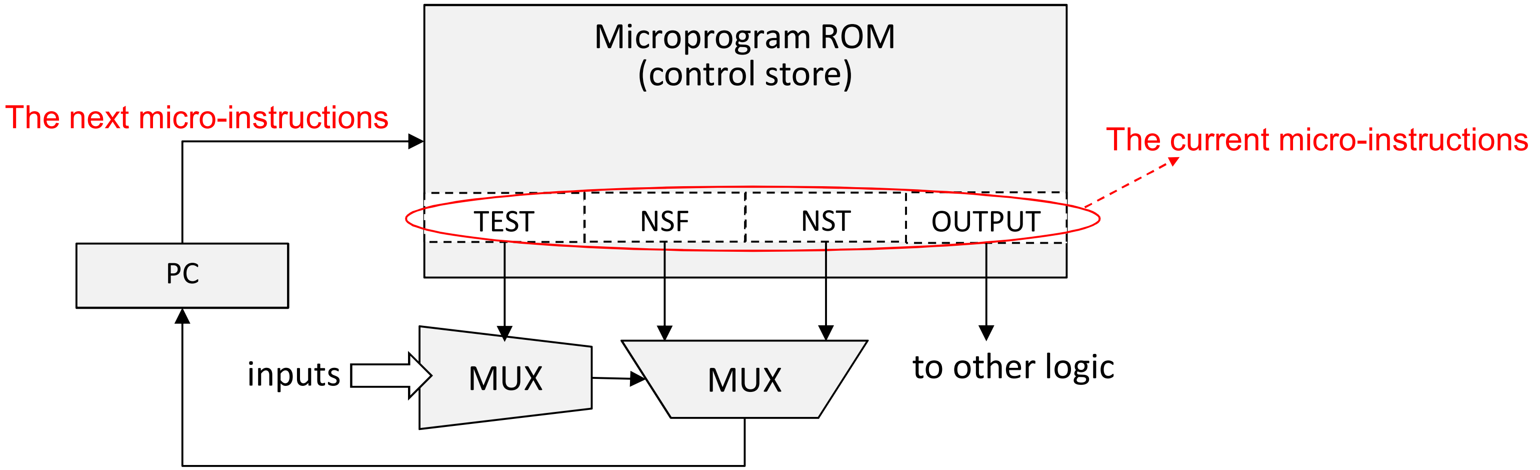
	1	2	3	4	5	6	7	8	
ARM 7	Fetch	Decode	Execute	≤ 150 MHz, the basis of Cortex-M series					
			Shift/ALU memory access sign/zero extension commit result						
ARM 9	Fetch	Decode	Execute	Memory	Writeback	≤ 250 MHz, used in Nokia 7650			
	Calc PC I-cache read		Shift/ALU	D-cache RW sign/zero ext.	commit result				
ARM 10	Fetch	Issue	Decode	Execute	Memory	Writeback	≤ 325 MHz		
ARM 11	Fetch1	Fetch2	Decode	ISS	Shifter	ALU	SAT	Writeback	≤ 500 MHz, Used in iPhone 1
	BTB access						saturation		

Microprogramming for CISC

- Hardwiring the controller FSM to execute a multi-cycle CISC ISA instruction is expensive
  - Design/debugging cost is quite high
- A more flexible approach is to use microprogramming†
  - Using a tiny (in-circuit) computer to execute the FSM
  - Each microinstruction specifies
    - The outputs to be generated
    - Where to find the next microinstruction (i.e. state transitions)
  - Moore FSMs are more suitable for microprogramming

Microcode Sequencer

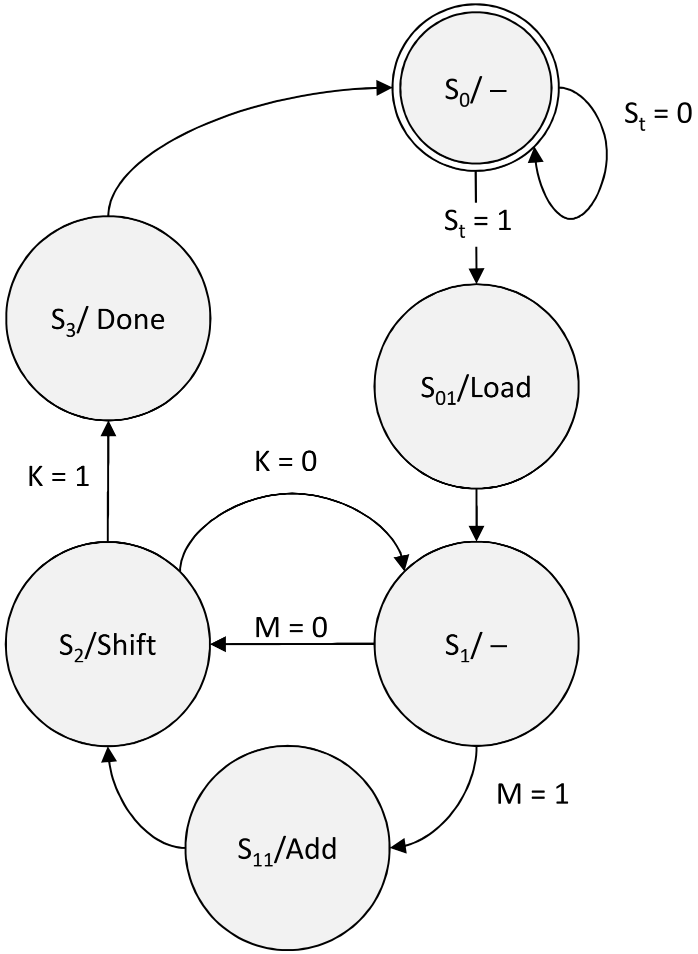
- Microprogrammed controllers are called sequencers
  - The control signals and the next state information are stored in a memory (i.e. ROM) inside the controller
  - The memory is called “control store” or “microcode memory”



TEST – which input is used to select next state?
NSF – next state if input is false
NST – next state if input is true
OUTPUT – control output signals

Example: Multiplier Controller

❑ The FSM and its microprogram of a 4-bit multiplier

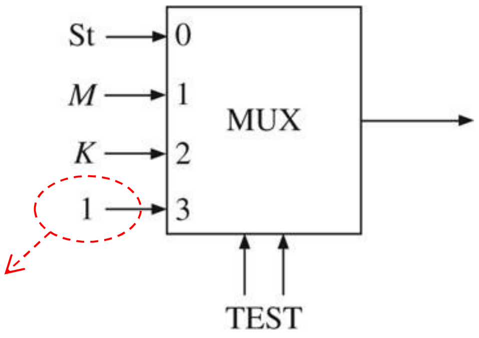


- Microprogram:

State	ABC	TEST	NSF	NST	Load	Add	Shift	Done
S_0	000	00	000	001	0	0	0	0
S_{01}	001	11	010	010	1	0	0	0
S_1	010	01	100	011	0	0	0	0
S_{11}	011	11	100	100	0	1	0	0
S_2	100	10	010	101	0	0	1	0
S_3	101	11	000	000	0	0	0	1

output signals

- Input select multiplexer:



for always-true conditional box

Discussions

- Most smartphone processors are based on the ARM ISAs, but the microarchitecture may be different
  - Apple A/M-series: fully developed by Apple Inc.
  - Qualcomm Kryo: some are developed by Qualcomm
  - Samsung Exynos: some are developed by Samsung
  - ARM Cortex A/X-series: developed by ARM, used by MediaTek, Huawei, Qualcomm, Samsung, and many other companies
    → where are the differences ?

- With the arrival of the RISC-V ISA, companies will design their own CPUs for marketing differentiation