AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

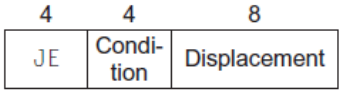
Procesory i Architektura Systemów Komputerowych

Architektura potokowa

IET
Katedra Elektroniki
Kraków 2015
dr inż. Roman Rumian

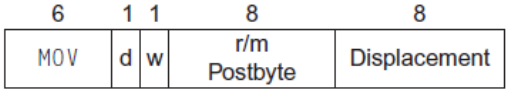
Typical x86 instruction formats

a. JE EIP + displacement

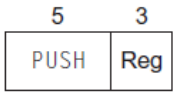


b. CALL

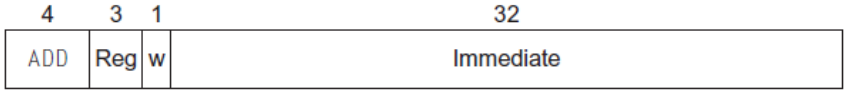
c. MOV EBX, [EDI + 45]



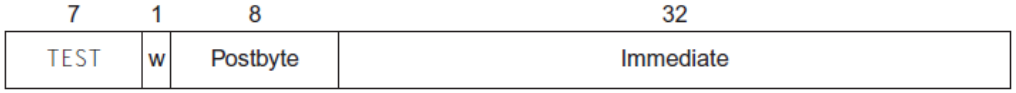
d. PUSH ESI



e. ADD EAX, #6765

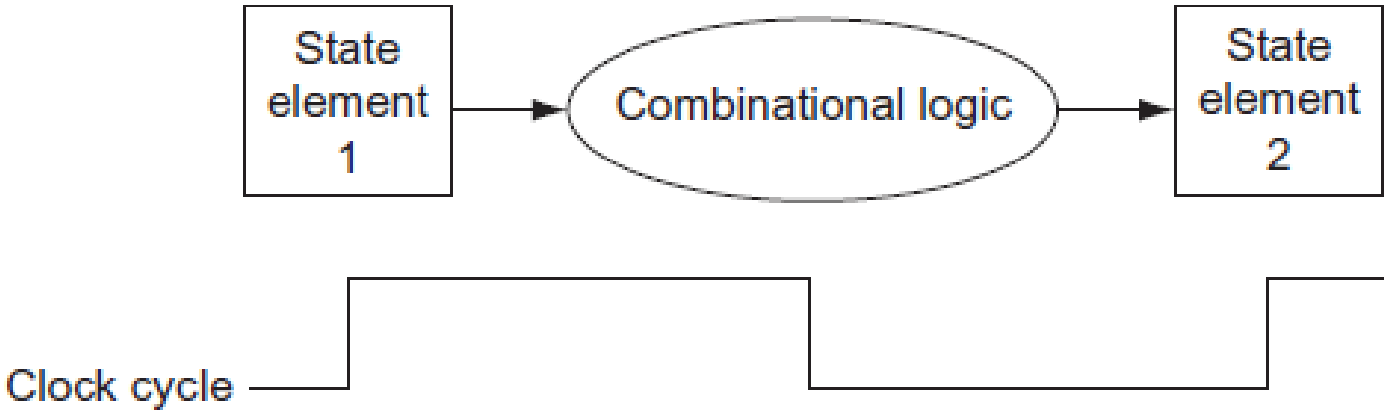


f. TEST EDX, #42



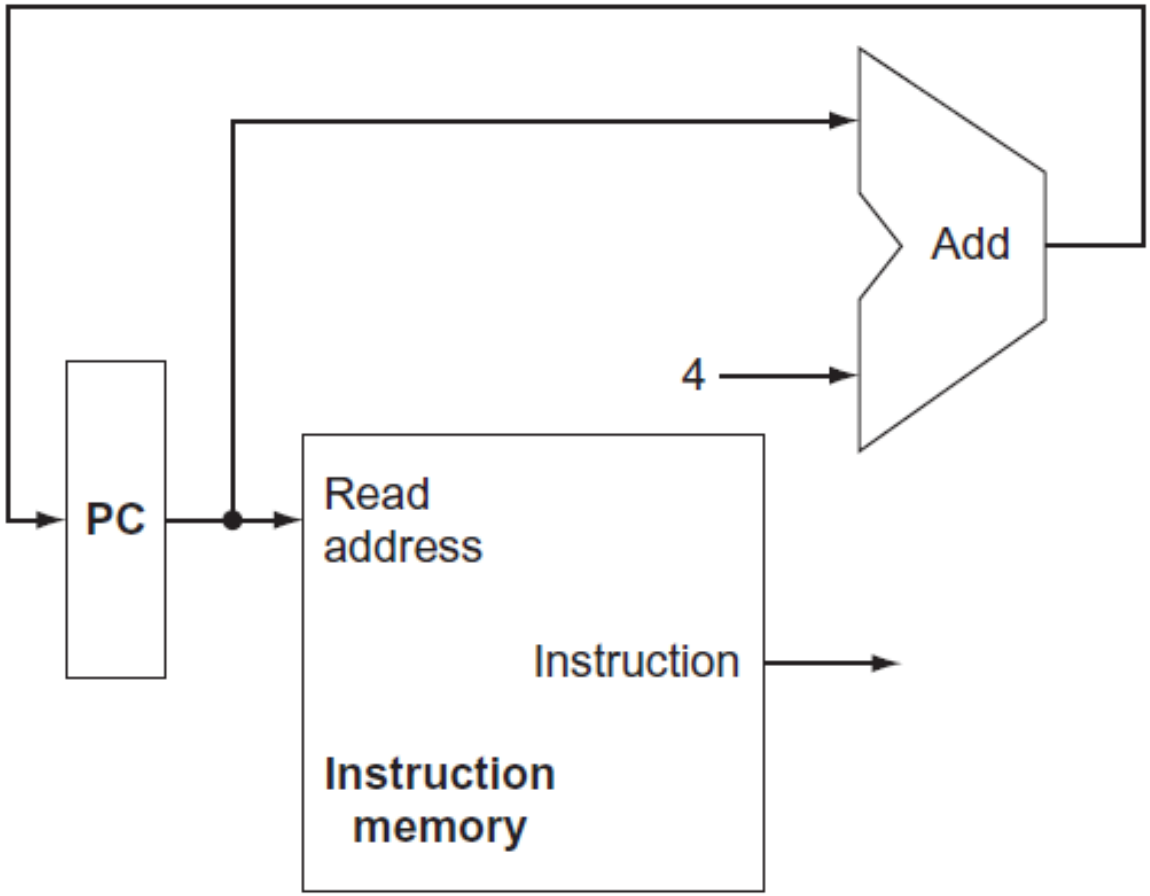
Some typical x86 instructions and their functions

Instruction	Function
je name	if equal(condition code) {EIP=name}; EIP-128 <= name < EIP+128
jmp name	EIP=name
call name	SP=SP-4; M[SP]=EIP+5; EIP=name;
movw EBX,[EDI+45]	EBX=M[EDI+45]
push ESI	SP=SP-4; M[SP]=ESI
pop EDI	EDI=M[SP]; SP=SP+4
add EAX,#6765	EAX= EAX+6765
test EDX,#42	Set condition code (flags) with EDX and 42
movsl	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

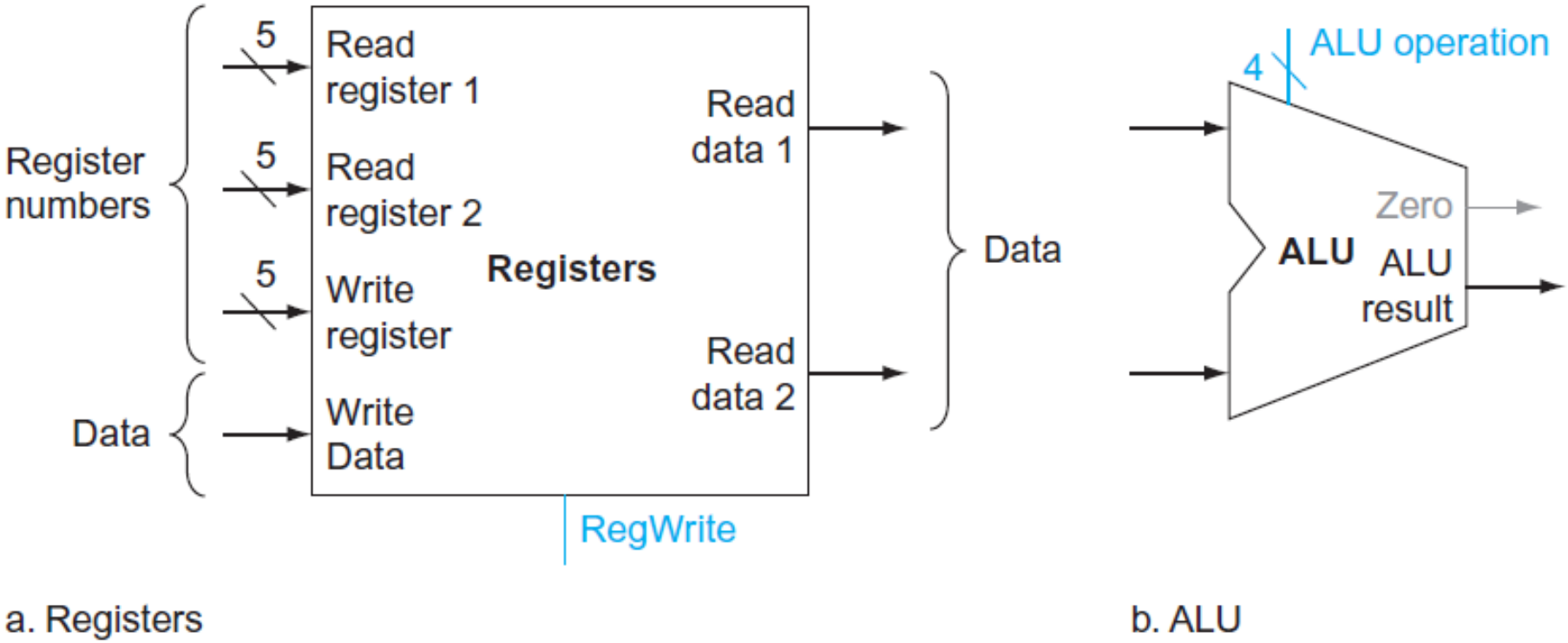


Combinational logic, state elements, and the clock are closely related.
In a synchronous digital system, the clock determines when elements with state will write values into internal storage. Any inputs to a state element must reach a stable value (that is, have reached a value from which they will not change until after the clock edge) before the active clock edge causes the state to be updated. All state elements, including memory, are assumed to be positive edge-triggered; that is, they change on the rising clock edge.

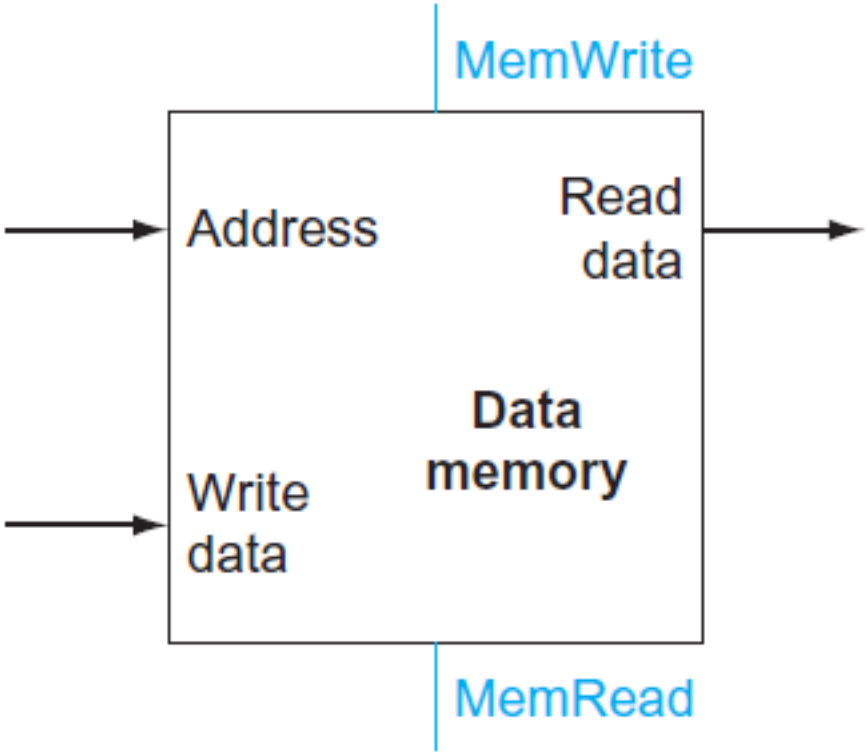
A portion of the datapath used for fetching instructions and incrementing the program counter.



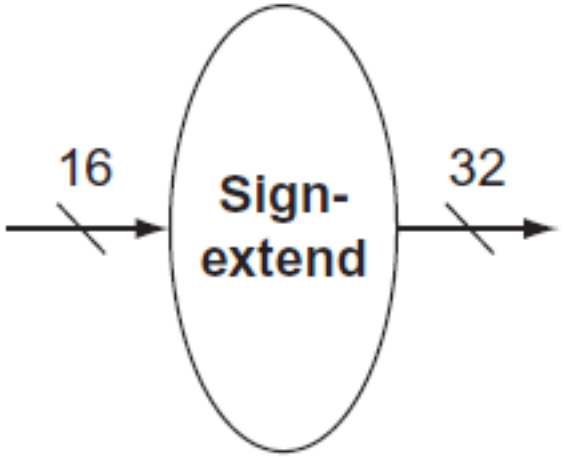
The two elements needed to implement R-format ALU operations are the register file and the ALU



The two units needed to implement loads and stores, in addition to the register file and ALU, are the data memory unit and the sign extension unit

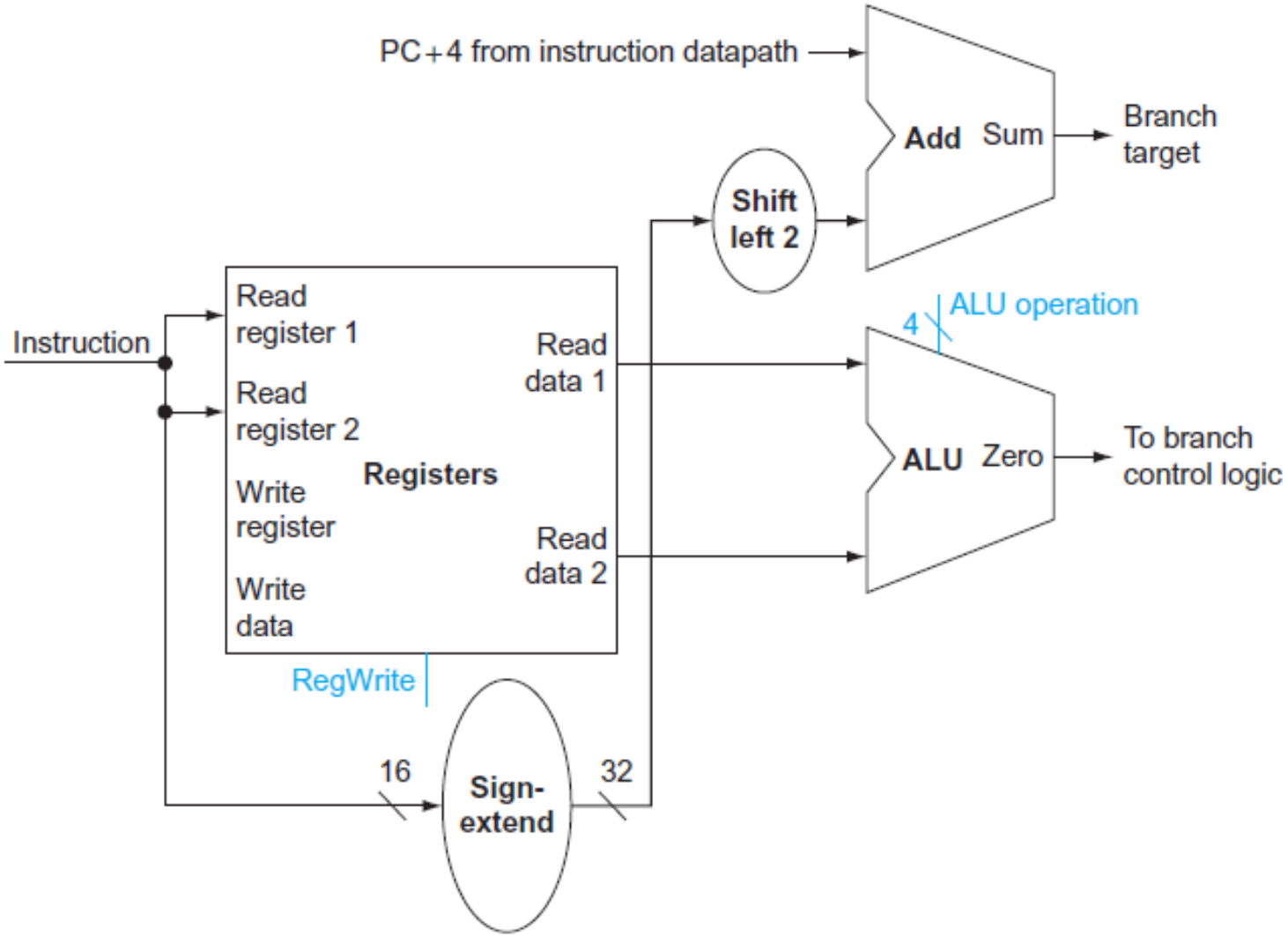


a. Data memory unit

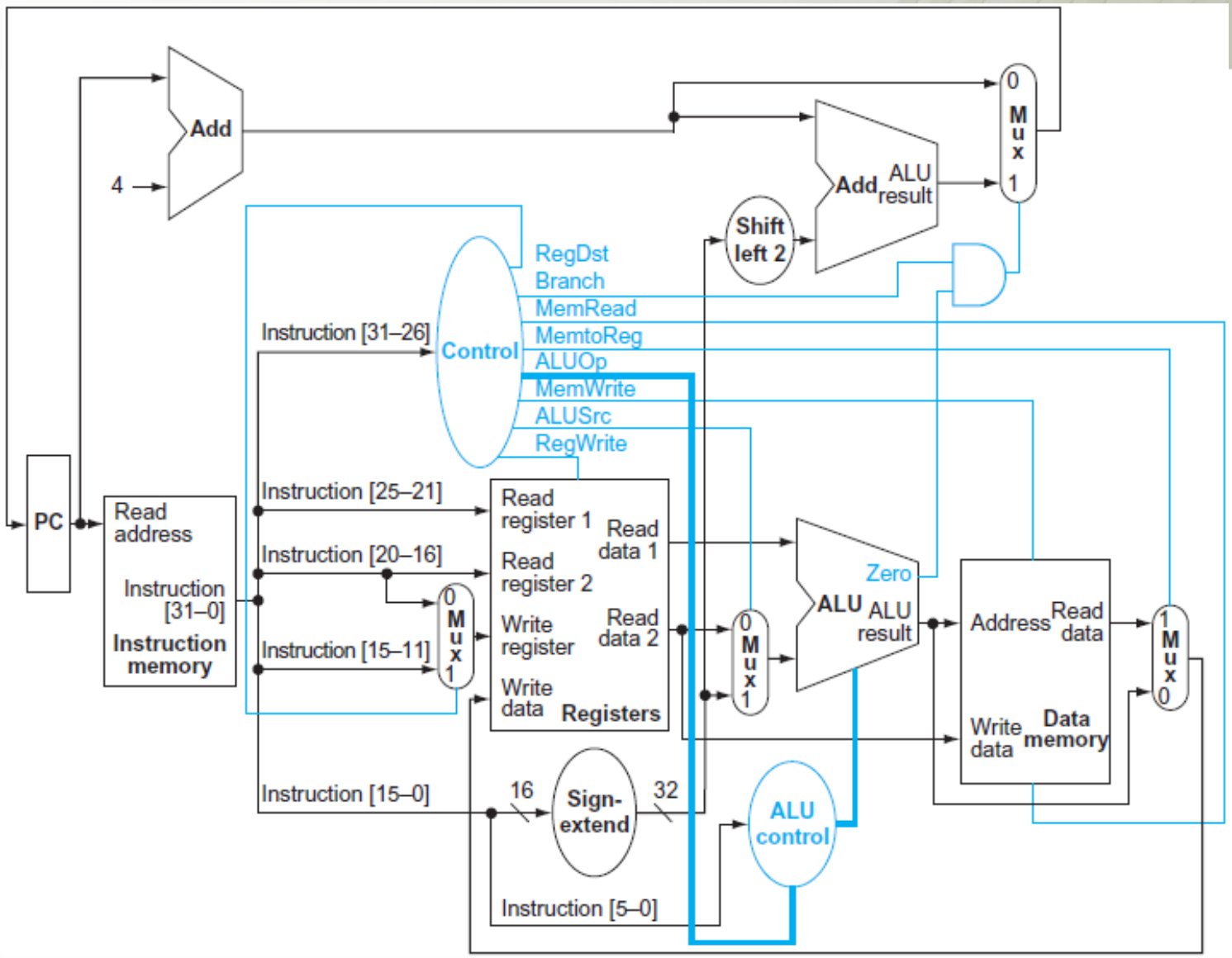


b. Sign extension unit

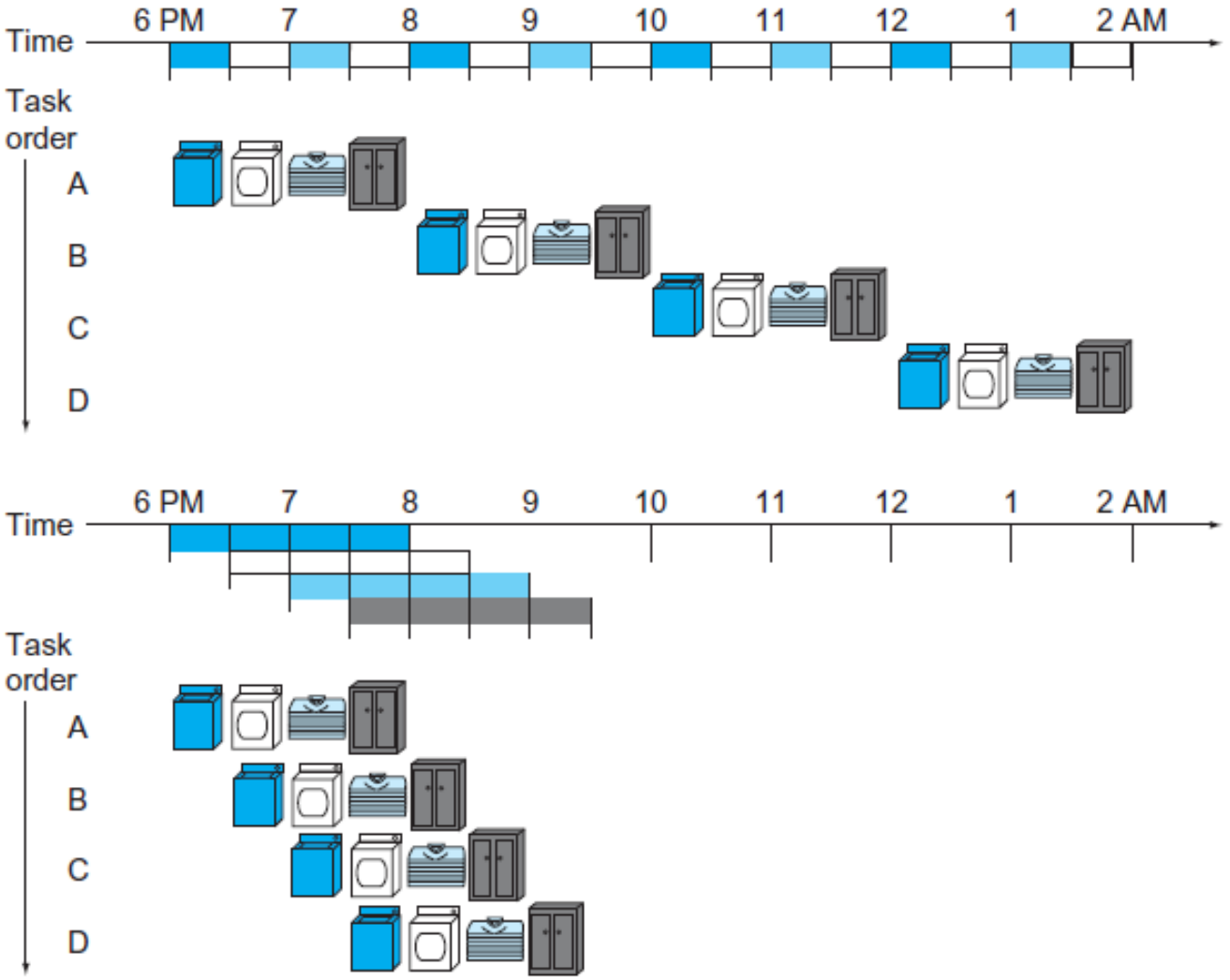
The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits



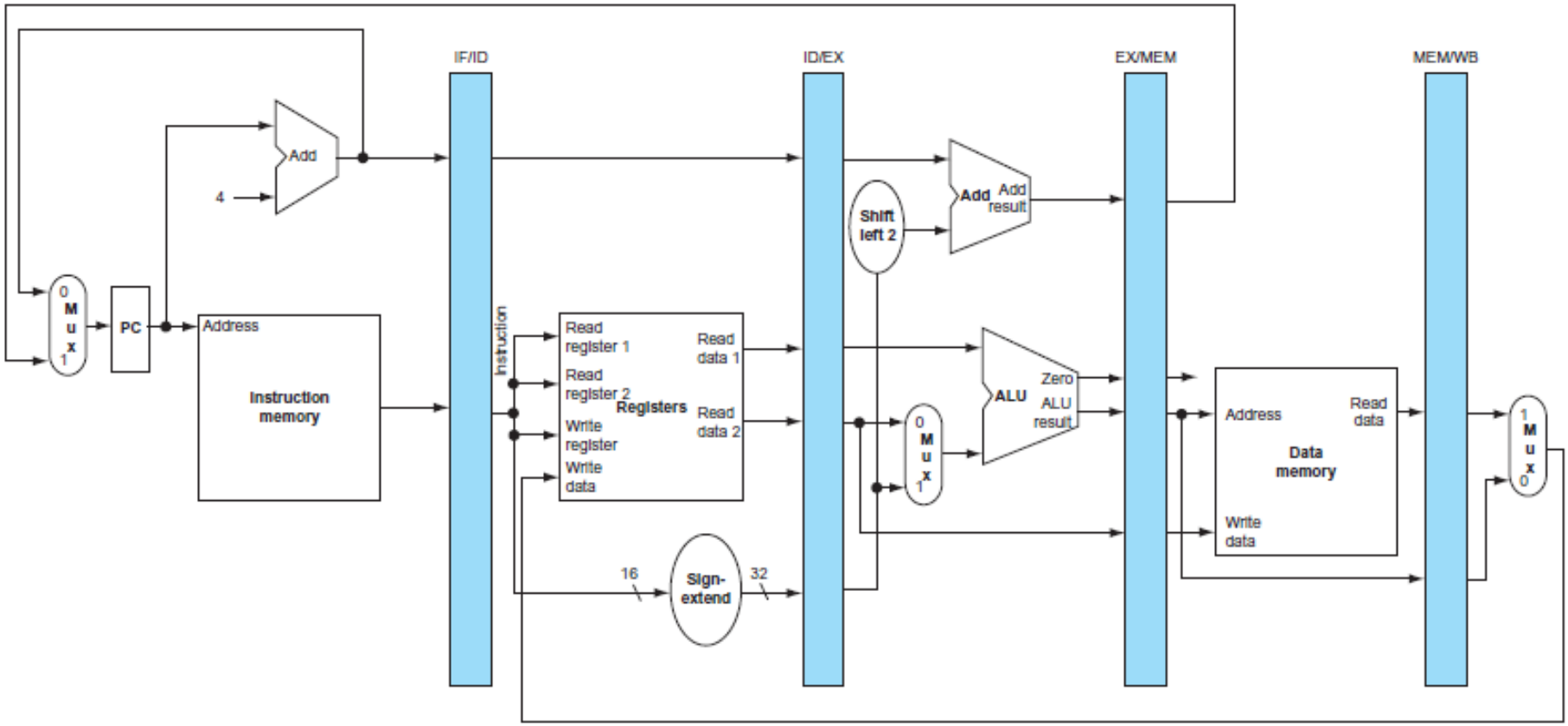
The simple datapath with the control unit



The laundry analogy for pipelining



The pipelined version of the datapath



Pipeline definitions

Pipelining exploits the potential **parallelism** among instructions. This parallelism is called **instruction-level parallelism (ILP)**.

Another approach is to replicate the internal components of the computer so that it can launch multiple instructions in every pipeline stage. The general name for this technique is **multiple issue**.
Launching multiple instructions per stage allows the instruction execution rate to exceed the clock rate or, stated alternatively, the CPI to be less than 1. It is sometimes useful to flip the metric and use IPC, or instructions per clock cycle. Hence, a 4 GHz four-way multiple-issue microprocessor can execute a peak rate of 16 billion instructions per second and have a best-case CPI of 0.25, or an IPC of 4. Assuming a five-stage pipeline, such a processor would have 20 instructions in execution at any given time. Today's high-end microprocessors attempt to issue from three to six instructions in every clock cycle.

There are two major ways to implement a multiple-issue processor, with the major difference being the division of work between the compiler and the hardware. Because the division of work dictates whether decisions are being made statically (that is, at compile time) or dynamically (that is, during execution), the approaches are sometimes called **static multiple issue** and **dynamic multiple issue**.

issue packet The set of instructions that issues together in one clock cycle; the packet may be determined statically by the compiler or dynamically by the processor.

superscalar An advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle by selecting them during execution.
dynamic pipeline scheduling Hardware support for reordering the order of instruction execution so as to avoid stalls.
static multiple issue An approach to implementing a multiple-issue processor where many decisions are made by the compiler before execution.
dynamic multiple issue An approach to implementing a multipl eissue processor where many decisions are made during execution by the processor.
issue slots The positions from which instructions could issue in a given clock cycle; by analogy, these correspond to positions at the starting blocks for a sprint.

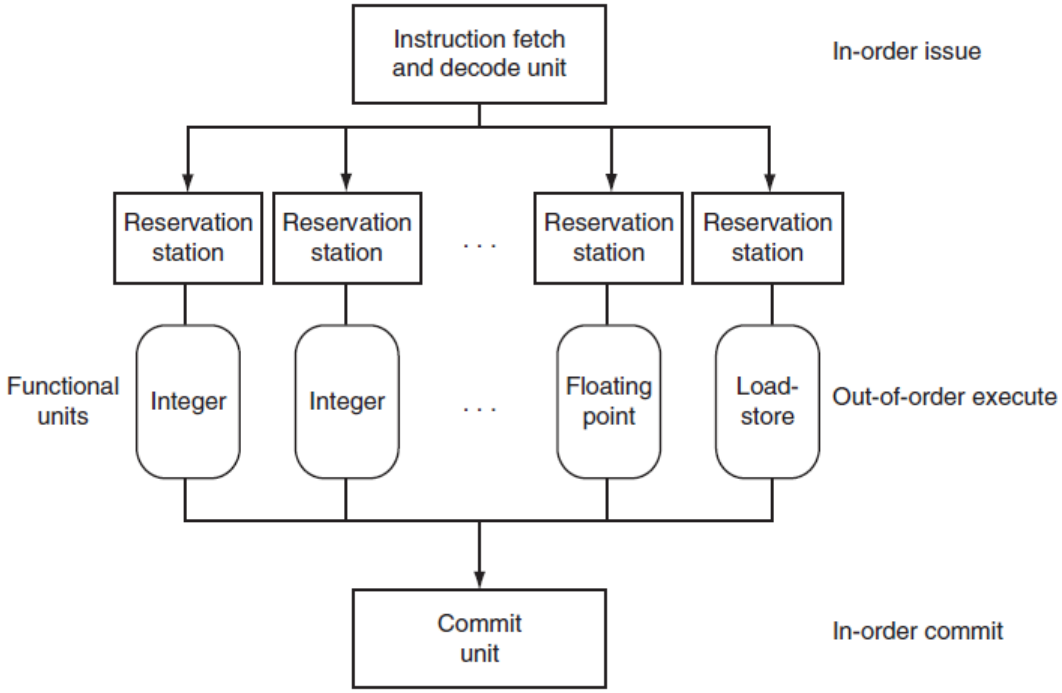
Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

Static two-issue pipeline in operation

commit unit The unit in a dynamic or out-of-order execution pipeline that decides when it is safe to release the result of an operation to programmervisible registers and memory.

reservation station A buffer within a functional unit that holds the operands and the operation.

reorder buffer The buffer that holds results in a dynamically scheduled processor until it is safe to store the results to memory or a register.



The three primary units of a dynamically scheduled pipeline

Record of Intel Microprocessors in terms of pipeline complexity, number of cores, and power

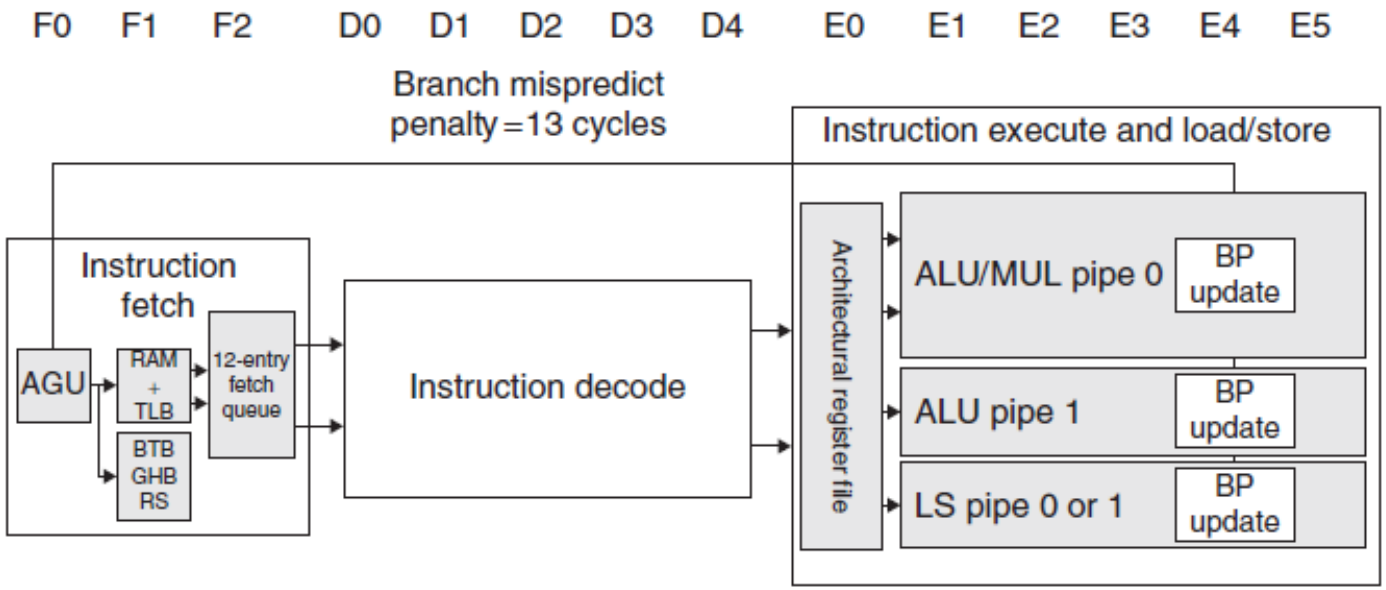
Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power	
Intel 486	1989	25 MHz	5	1	No	1	5	W
Intel Pentium	1993	66 MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	1	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

AGH

Specification of the ARM Cortex-A8 and the Intel Core i7 920

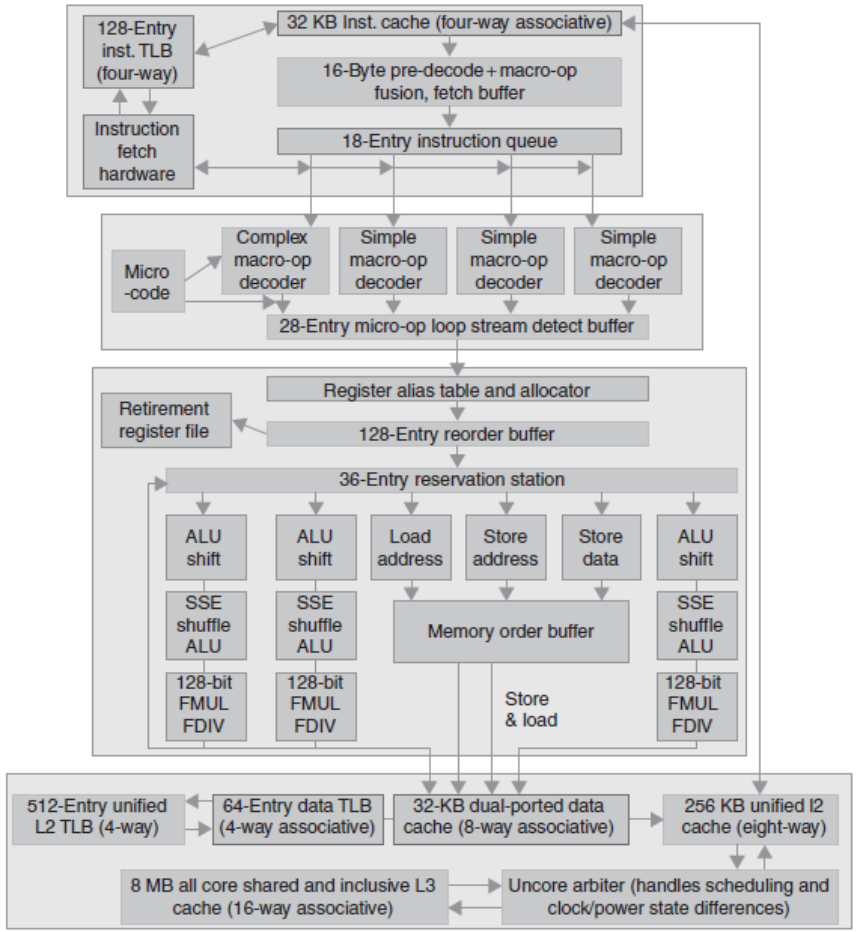
Processor	ARM A8	Intel Core I7 920
Market	Personal Mobile Device	Server, Cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple Issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline Stages	14	14
Pipeline schedule	Static In-order	Dynamic Out-of-order with Speculation
Branch prediction	2-level	2-level
1st level caches / core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
2nd level cache / core	128 - 1024 KiB	256 KiB
3rd level cache (shared)	–	2 - 8 MiB

The ARM Cortex-A8 pipeline



The A8 pipeline. The first three stages fetch instructions into a 12-entry instruction fetch buffer. The *Address Generation Unit* (AGU) uses a *Branch Target Buffer* (BTB), *Global History Buffer* (GHB), and a *Return Stack* (RS) to predict branches to try to keep the fetch queue full. Instruction decode is five stages and instruction execution is six stages.

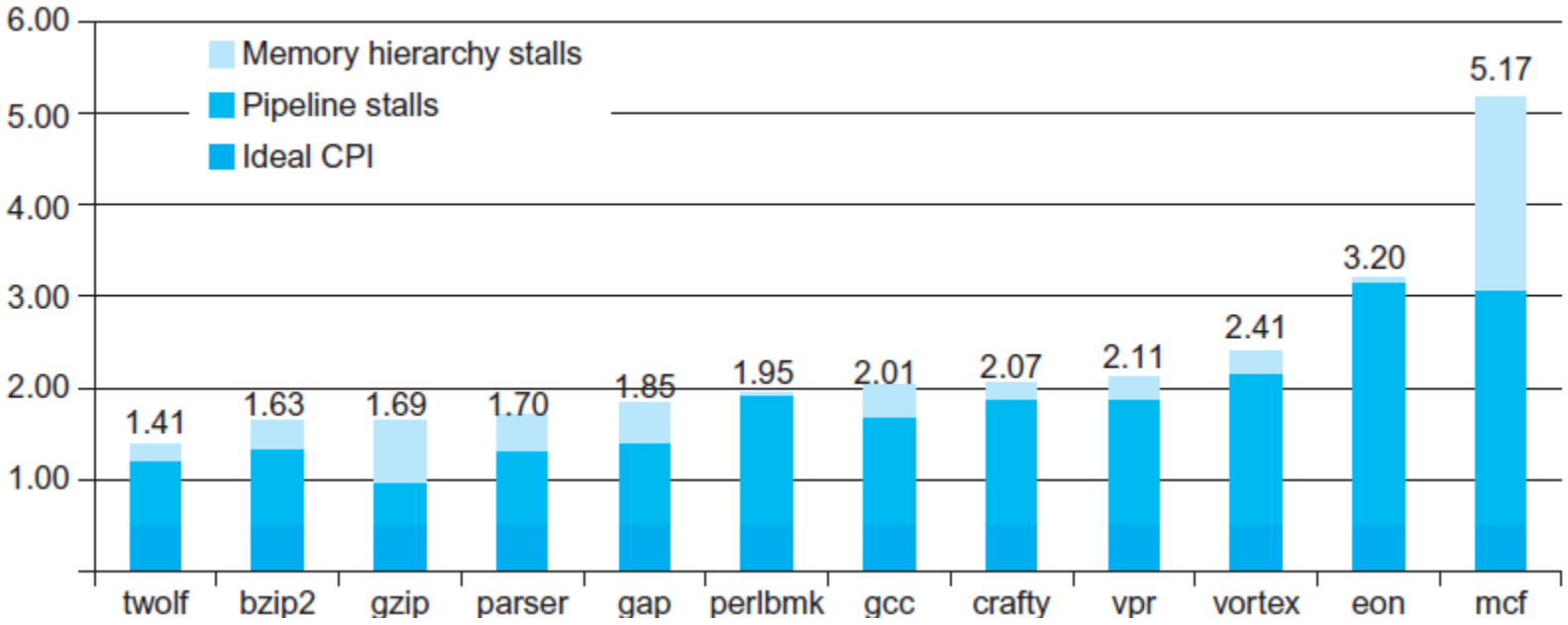
Intel Core i7 pipeline



The Core i7 pipeline with memory components. The total pipeline depth is 14 stages, with branch mispredictions costing 17 clock cycles. This design can buffer 48 loads and 32 stores. The six independent units can begin execution of a ready RISC operation each clock cycle.

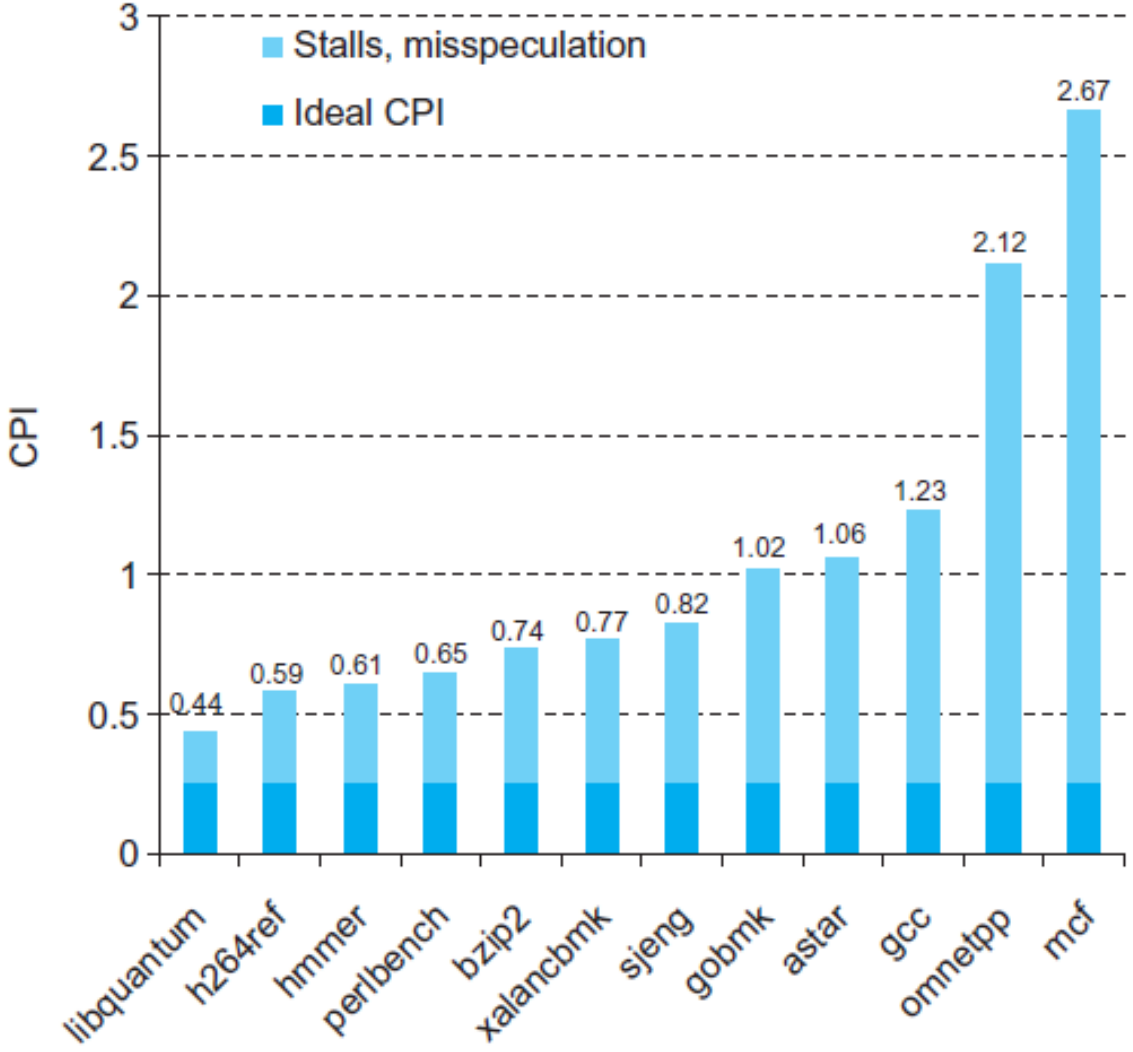
The eight steps an x86 instruction goes through for execution.

1. Instruction fetch—The processor uses a multilevel branch target buffer to achieve a balance between speed and prediction accuracy. There is also a return address stack to speed up function return. Mispredictions cause a penalty of about 15 cycles. Using the predicted address, the instruction fetch unit fetches 16 bytes from the instruction cache.
2. The 16 bytes are placed in the predecode instruction buffer— The predecode stage transforms the 16 bytes into individual x86 instructions. This predecode is nontrivial since the length of an x86 instruction can be from 1 to 15 bytes and the predecoder must look through a number of bytes before it knows the instruction length. Individual x86 instructions are placed into the 18-entry instruction queue.
3. Micro-op decode—Individual x86 instructions are translated into microoperations (micro-ops). Three of the decoders handle x86 instructions that translate directly into one micro-op. For x86 instructions that have more complex semantics, there is a microcode engine that is used to produce the micro-op sequence; it can produce up to four micro-ops every cycle and continues until the necessary micro-op sequence has been generated. The micro-ops are placed according to the order of the x86 instructions in the 28-entry micro-op buffer.
4. The micro-op buffer performs loop stream detection—If there is a small sequence of instructions (less than 28 instructions or 256 bytes in length) that comprises a loop, the loop stream detector will find the loop and directly issue the micro-ops from the buffer, eliminating the need for the instruction fetch and instruction decode stages to be activated.
5. Perform the basic instruction issue—Looking up the register location in the register tables, renaming the registers, allocating a reorder buffer entry, and fetching any results from the registers or reorder buffer before sending the micro-ops to the reservation stations.
6. The i7 uses a 36-entry centralized reservation station shared by six functional units. Up to six micro-ops may be dispatched to the functional units every clock cycle.
7. The individual function units execute micro-ops and then results are sent back to any waiting reservation station as well as to the register retirement unit, where they will update the register state, once it is known that the instruction is no longer speculative. The entry corresponding to the instruction in the reorder buffer is marked as complete.
8. When one or more instructions at the head of the reorder buffer have been marked as complete, the pending writes in the register retirement unit are executed, and the instructions are removed from the reorder buffer.



CPI on ARM Cortex A8 for the Minnespec benchmarks, which are small versions of the SPEC2000 benchmarks. These benchmarks use the much smaller inputs to reduce running time by several orders of magnitude. The smaller size significantly *underestimates* the CPI impact of the memory hierarchy

CPI of Intel Core i7 920 running SPEC2006 integer benchmarks



Flynn taxonomy of computer architectures

Single Instruction, Single Data (SISD) There is only one stream of instructions and one stream of data; ignoring co-processors and additional PEs added to produce identical, redundant computation (e.g., to cope with errors), there can only be one useful PE in such a processor.

Single Instruction, Multiple Data (SIMD) There is only one stream of instructions but there are many streams of data; thus we can have many PEs each executing the same instruction at a given point in time but using different data in each case. Examples of SIMD computers include the Cray series of vector supercomputers and the Graphics Processing Units (GPUs) that render 3D images within most modern graphics cards.

Multiple Instruction, Single Data (MISD) It is hard to justify the existence of MISD processors, in the sense that it is hard to see how such a device could compute useful results: what we are describing is a situation where we take a single data stream and operate on it using many different instruction streams.
Examples of MISD processors are rare but application areas are typically those that require some form of duplicate computation. For example, a computer to forecast weather might run several simulation models on the same dataset and average the results to get a more reasonable result.

Multiple Instruction, Multiple Data (MIMD) There are many streams of instructions and many streams of data; essentially we have many PEs executing different instructions on different data at the same time. In a sense, this is the most general form of parallel processor. Examples include a cluster of workstations (where the PEs are physically separate computers) connected by a network in order to collectively perform some task; the Beowulf design developed by Donald Becker at NASA is an example of how one can construct such system inexpensively. However, recent advances in processor design have led us to so-called multi-core designs whereby one has several processor cores on one microchip; the combined system is essentially a small-scale MIMD computer.

This taxonomy is usually extended to include the so-called **Single Program, Multiple Data (SPMD)** model. This is similar to SIMD in the sense that each processor is presented with a single set of instructions to execute. However, unlike SIMD a given PE within an SPMD processor can obtain an identifier (i.e., a processor number of some sort) and make conditional decisions based on that identifier. So, for example, PE number one might take one control-flow path within the single program while PE number two might take another. Hence although they are both given a single program, they might execute different instructions within it.