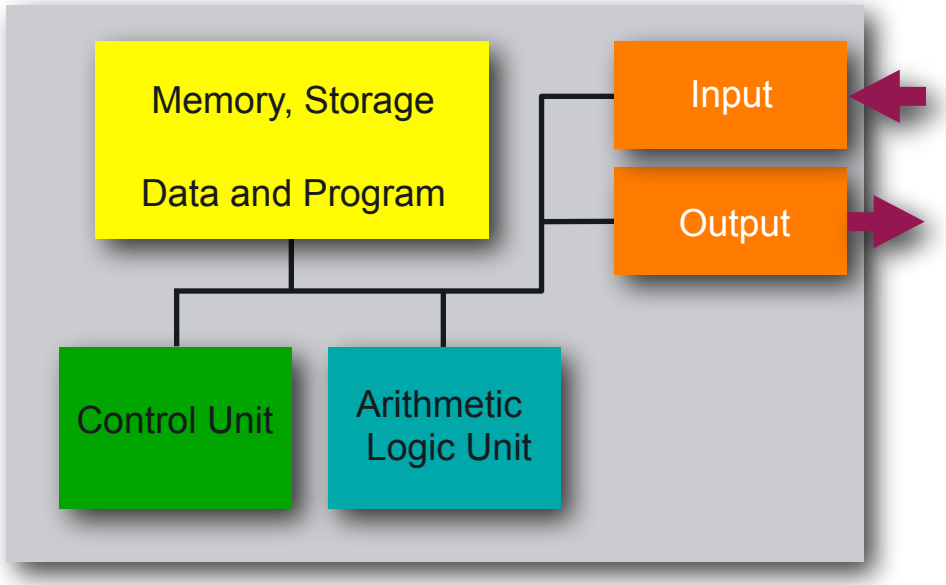
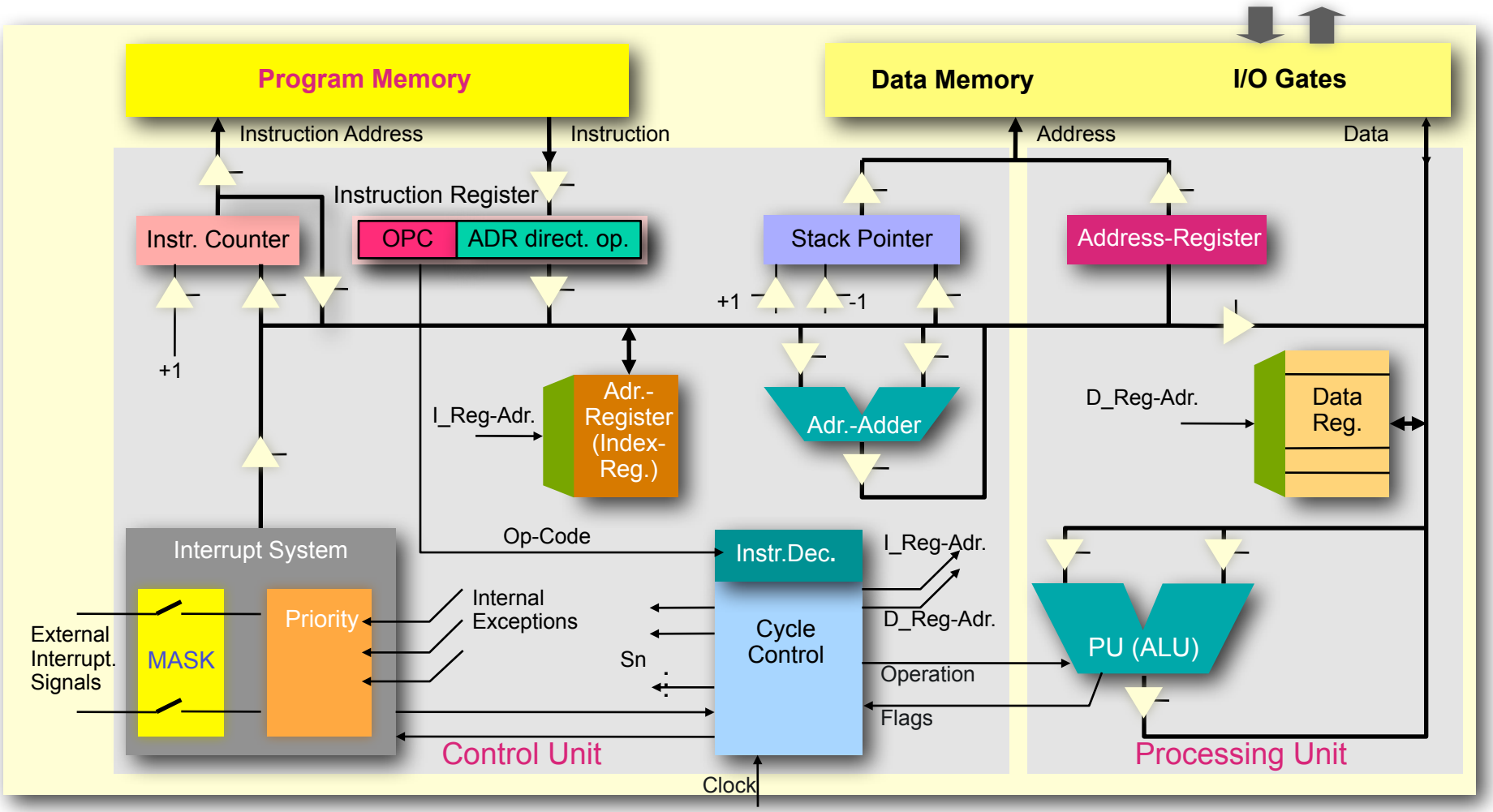
Major Components of a Processor

Main components:

- central processing unit, (CPU) including
  - control unit
  - arithmetic logic unit, (ALU)
- memory
- (I/O unit)
- bus system for connecting the components

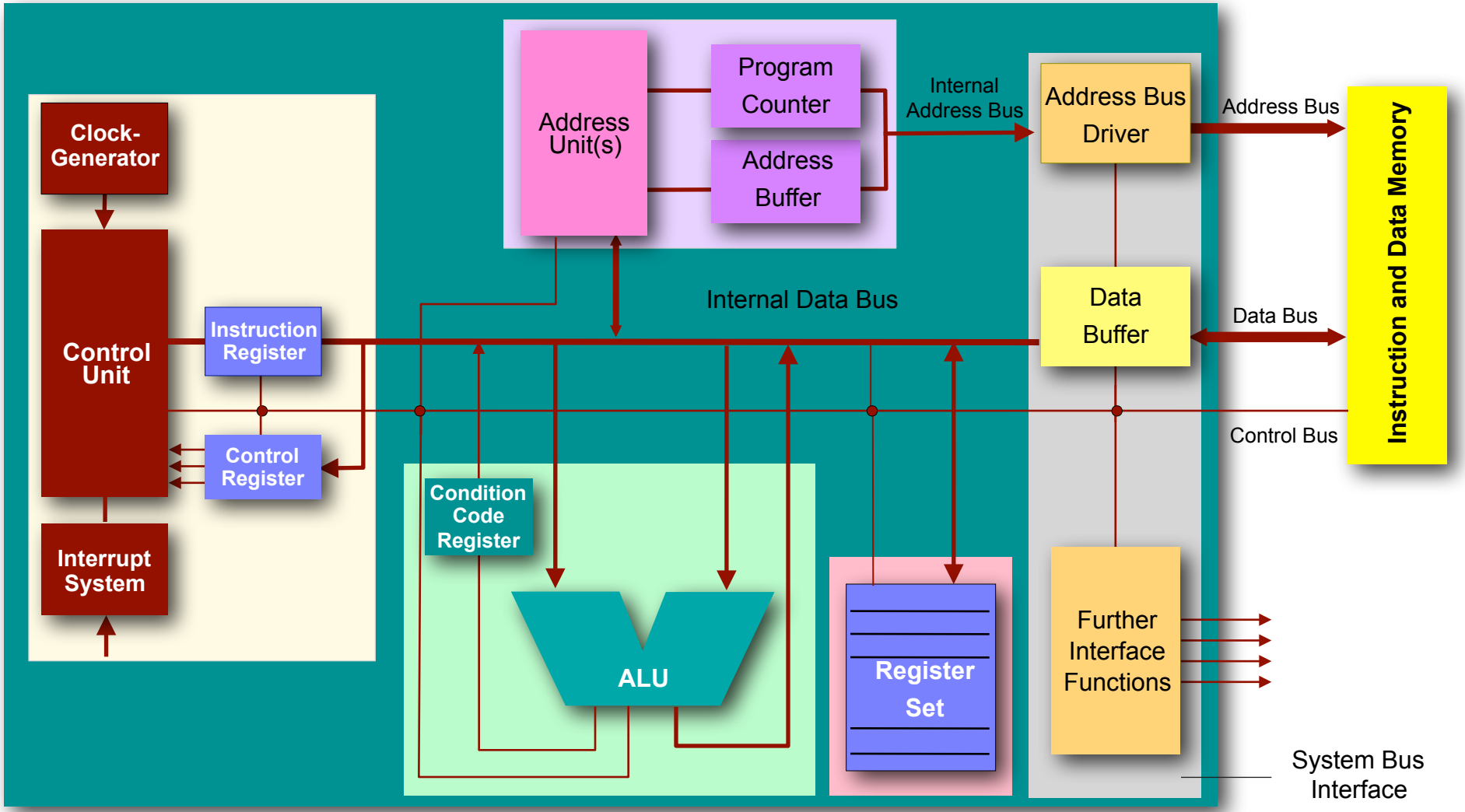


Generic Processor (1)



K+A-3_F 2

Generic Processor (2), Block Representation



K+A-3_F 3

Control Unit (CU)

- Checks/generates control signals for system components (processor internally and externally).
- Ensures executing of all system processes with precise timing.
- Generates status signals of the processor.
- Additionally important components:
  - Instruction register: buffers the actual executed instruction.
  - Interrupt system: handles all arrived interrupt requests.

Arithmetic Logic Unit (ALU)

- Executes all arithmetic and logic operations required from the CU → arithmetic logic unit (ALU).
- Informs the CU by means of status registers/flags about the state of the result.
- More complex processors often have one or more ALU(s) for processing special tasks, f.e. FPU, MMX.

Address Unit (AU)

- Computes the address of an operand/instruction according to regulations of CU.
- Harvard architecture processors (each with separate data and program memory) or DSPs often have multiple AUs → simultaneous access on instruction and data respectively multiple data.
- Two special registers for intermediate storing:
  - Instruction pointer/program counter IP/PC: always contains the address of the memory cell, wherein the next instruction(part) has been saved.
  - Address buffer: contains the address of the next required operand, determined by addressing mode – specified by the instruction.

Register Set (Register File), System Bus Interface

- Register set (register file)
  - mostly consists of data registers, address registers and special registers,
  - temporary storing of frequently used data (operands) and (memory) addresses.
- System bus interface
  - includes diverse registers/latches for buffering data, addresses, control signals; furthermore bus drivers and so on.

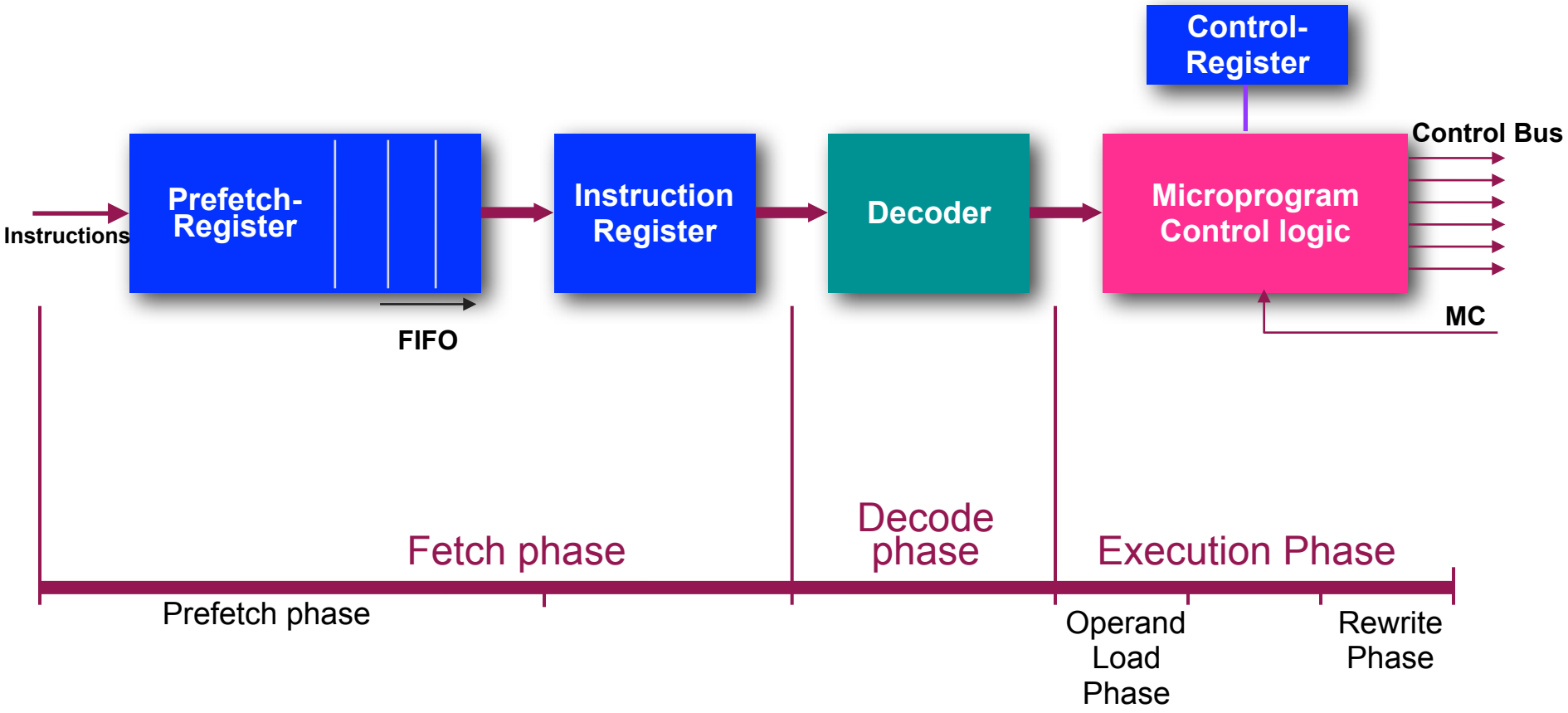
Performance Effects

- Operating “speed” (performance) of a processor is determined by
  - system clock,
  - speed of separate components,
  - interaction from architecture (HW) and instruction set (SW),
  - utilization/throughput of components,
  - …
- Increasing the performance is possible with modifications of these points, f.e.
  - increasing clock frequency,
  - coordinated, balanced concept of hardware and software,
  - increasing throughput with ideally assuming regular utilization of all components – they handle different parts of one or multiple instructions like a conveyor belt →„Pipelining“.

Functions of Control Unit

- CUs in general are viewed as sequential logic systems.
- They take over the sequential control in the arithmetic logic unit and the transport of operands/instructions from and to all registers, memory cells and interfaces.
- They define the following for the complete processor:
  - What has to be done in each moment of instruction execution?
  - Where are the actions depending from an instruction to be triggered?
- Instructions for action → program = sequence of machine instructions (macro instructions).
- The instructions are loaded step-by-step from the CU into the processor and then executed.

Phases of Instruction Processing (1)

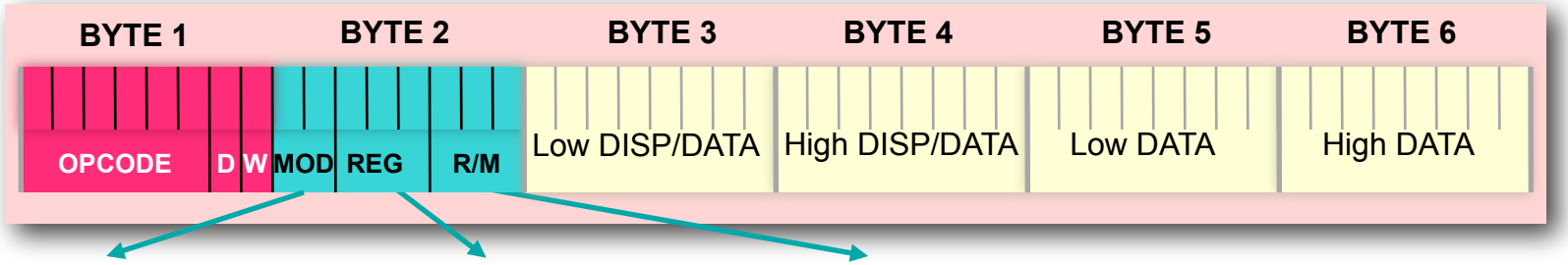


K+A-3_F 10

Phases of Instruction Processing (2)

1. Instruction Fetch, (IF) (1)

- Machine instruction has been loaded from (program) memory or from a register (see below FIFO) into the instruction register, whereas the operation code (opcode, OPC) is of interest at first.
  Example: instruction word of 8086



Mode (MOD) Field Encoding

Code	Explanation
00	Memory Mode, no displacement follows
01	Memory Mode, 8-bit displacement follows
10	Memory Mode, 16-bit displacement follows
11	Register Mode (no displacement)

Register (REG) Field Encoding

REG	W=0	W=1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Register/Memory (R/M) Field Encoding

MOD = 11			Effective Address Calculation			
R/M	W=0	W=1	R/M	MOD=00	MOD=01	MOD=10
000	AL	AX	000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16
001	CL	CX	001	(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16
010	DL	DX	010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16
011	BL	BX	011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16
100	AH	SP	100	(SI)	(SI) + D8	(SI) + D16
101	CH	BP	101	(DI)	(DI) + D8	(DI) + D16
110	DH	SI	110	Direct Access	(BP) + D8	(BP) + D16
111	BH	DI	111	(BX)	(BX) + D8	(BX) + D16

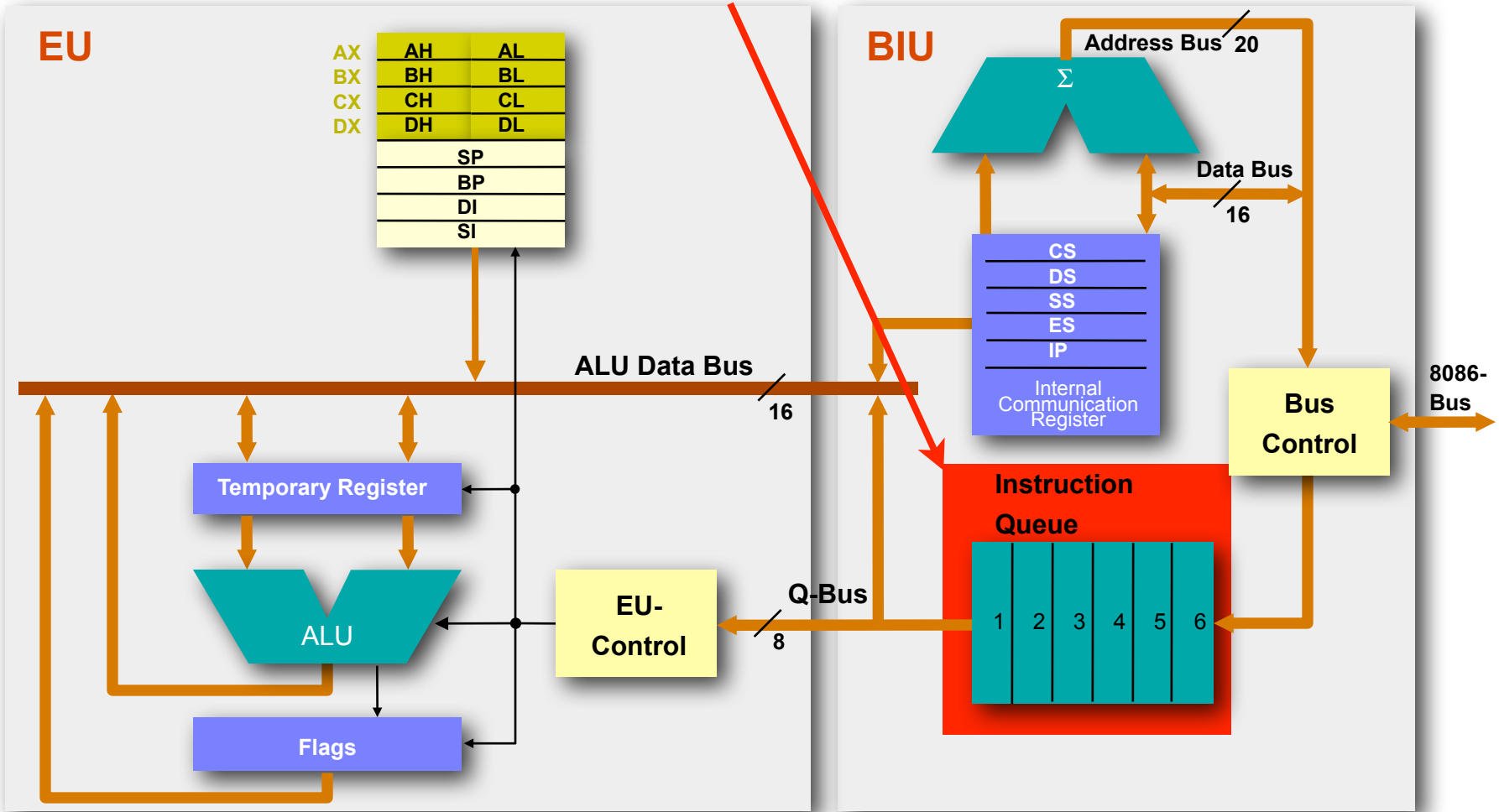
Phases of Instruction Processing (3)

1. Instruction Fetch (IF) (2)

- Often used, the so called „opcode prefetching“: Additonally to the current instruction, the following instruction(s) are loaded into a FIFO memory (instruction queue) from memory.
  - Advantage: better (uniform) utilization of the system bus.
  - Disadvantage: In case of program branching the queue content must be wasted and the now current instructions have to be loaded.
    → Possibly timing problems?

Phases of Instruction Processing (4)

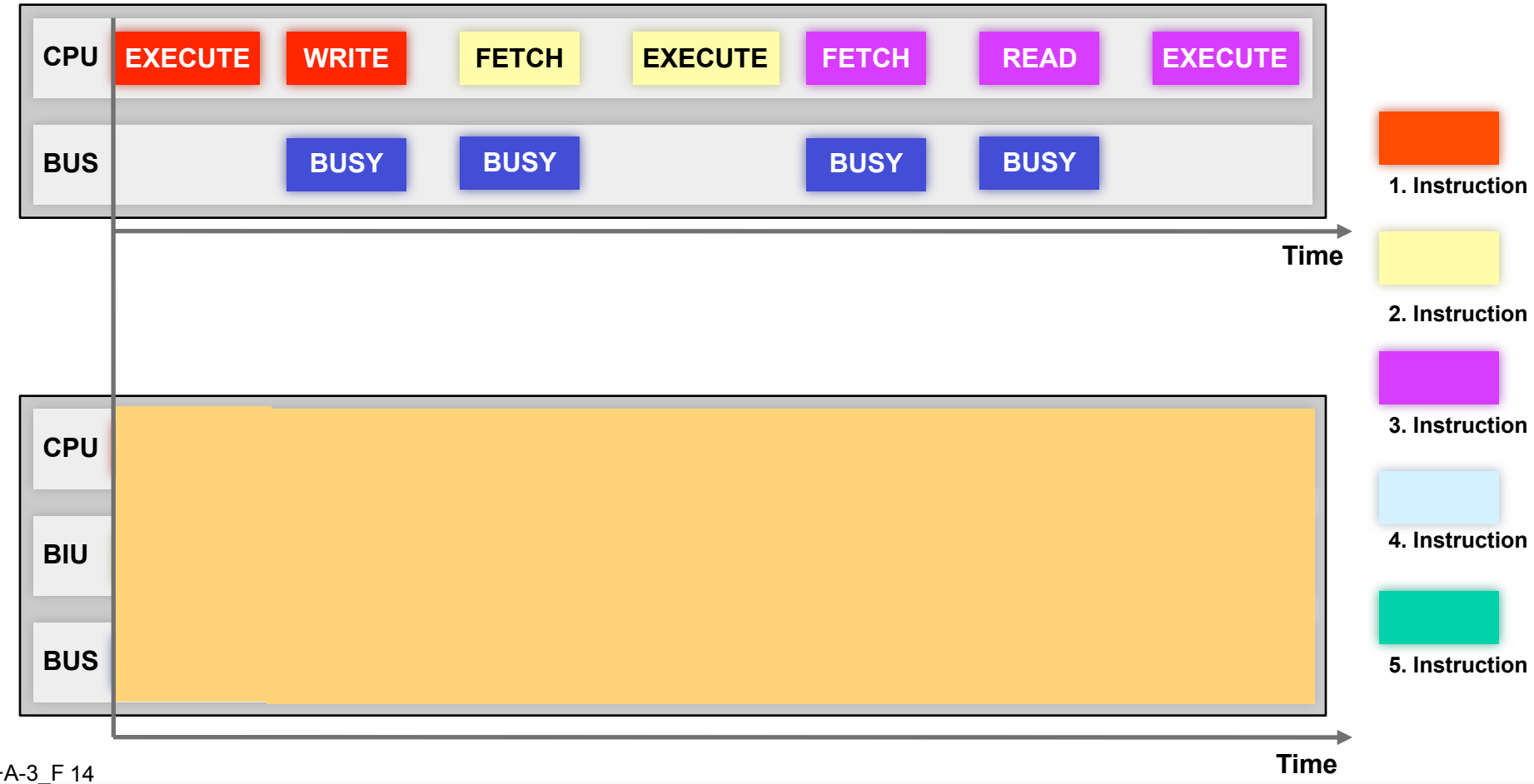
1. Instruction Fetch (IF) (3): prefetching in instruction queue



K+A-3_F 13

Phases of Instruction Processing (5)

1. Instruction Fetch (IF) (4): Superior bus load by prefetching.



Phases of Instruction Processing (6)

2. Instruction Decode, (ID)

- Opcode will be interpreted by instruction decoder.

  Example 1: ADD (add byte/word), Opcode 0000 00 dw → d and w define – according to their current value (0 or 1) – if the operands are available in memory and/or registers – and where the result has to be delievered to → mainly arithmetic processor resources.

  Example 2: MOV (move byte/word), opcode 1000 10dw
  d and w see above → pure displace operation.

- Important: How many operands are involved, which registers are required, how the address of memory operands has to be calculated?
- Preparing the actual execution of operation by providing the appropriate control signals.
- Loading the operands (according to processor/architecture – some processors do this later in the execution phase).

Phases of Instruction Processing (7)

3. Instruction Execute, (EX)

- Control signals now become active at the required components; the „actual“ operation is executed, according to specification operands are linked in the ALU; register contents de-/incremented, data transported, are interrupts enabled?
- Corresponding feedback signals (flags) inform the CU about states in the ALU.

4. Instruction Execute, (EX)

- Write back of the result to memory/register.

Phases of Instruction Processing (8)

- Each instruction contains of a number of subordinated activities → term „macro instruction“.
- The actual control of sequences/subordinated activities takes place in
  - either a hardwired sequential circuit,
  - or a microcoded control unit.
- Problems:
  - Instruction words may be of various lengths – depending on instruction → determined by operands and by each declared addressing mode.
  - Execution time of instructions is also different (associated number of required machine cycles or clock cycles).

→ Possibly leads to processing speed problems → CISC/RISC-architectures, pipelining.

General (1)

- After instruction set and data path (see ALU) architecture have been fixed → now control tasks of CU must be defined.
- For this is a FSM applicable.
- Each instruction passes diverse phases (see above), i.e. a finite sequence of states.
- Each state corresponds to a set of elementary hardware operations, that have to be executed during a clock cycle on the data path, on registers, memory, busses and so on.
- Assuming 2^n states of a processor, these states can be esteemed as control signals, leading over n state lines into the CU.

General (2)

- They control the ALU elementary operations, that have to be executed in each next clock cycle. Depending on the
  - machine instruction currently being processed (*p* bits for opcode and detailed operation specification) and
  - status information about states of registers (f.e. flags) (*u* bits), flowing back from the ALU.
- The elementary operations are activated by corresponding control lines (*m* bit), simultaneously the next state of the processor has to be determined.
- For processor control at least $s = n + p + u$ bits input signals have to be mapped to *m* output signals, with standard processors *m* is mostly > 100 (f.e. Intel P6 integer ALU: 120; floating point ALU: 285).

  This leads to a mapping problem: a table with approximately 2^s lines of length *m*.

Resource Requirements (1)

- A direct implementation of the above mentioned table – realized as a ROM allocation table – would be for an processor example
  - with 80 elementary states
    ld 80 = 6,321 → n = 7 bit used for presentation
    (2^6 = 64, 2^7 = 128)
  - with 8 bit wide opcode field and 6 additional function control bits
    → p = 8 + 6 = 14
  - 5 feedback bits from data path → u = 5
  - so s = 26
  - and with assumed 100 control lines for management of data path
    → m = 100

ROM requirement of 2^{26} words for each 100 bit → approx. 800 MByte!

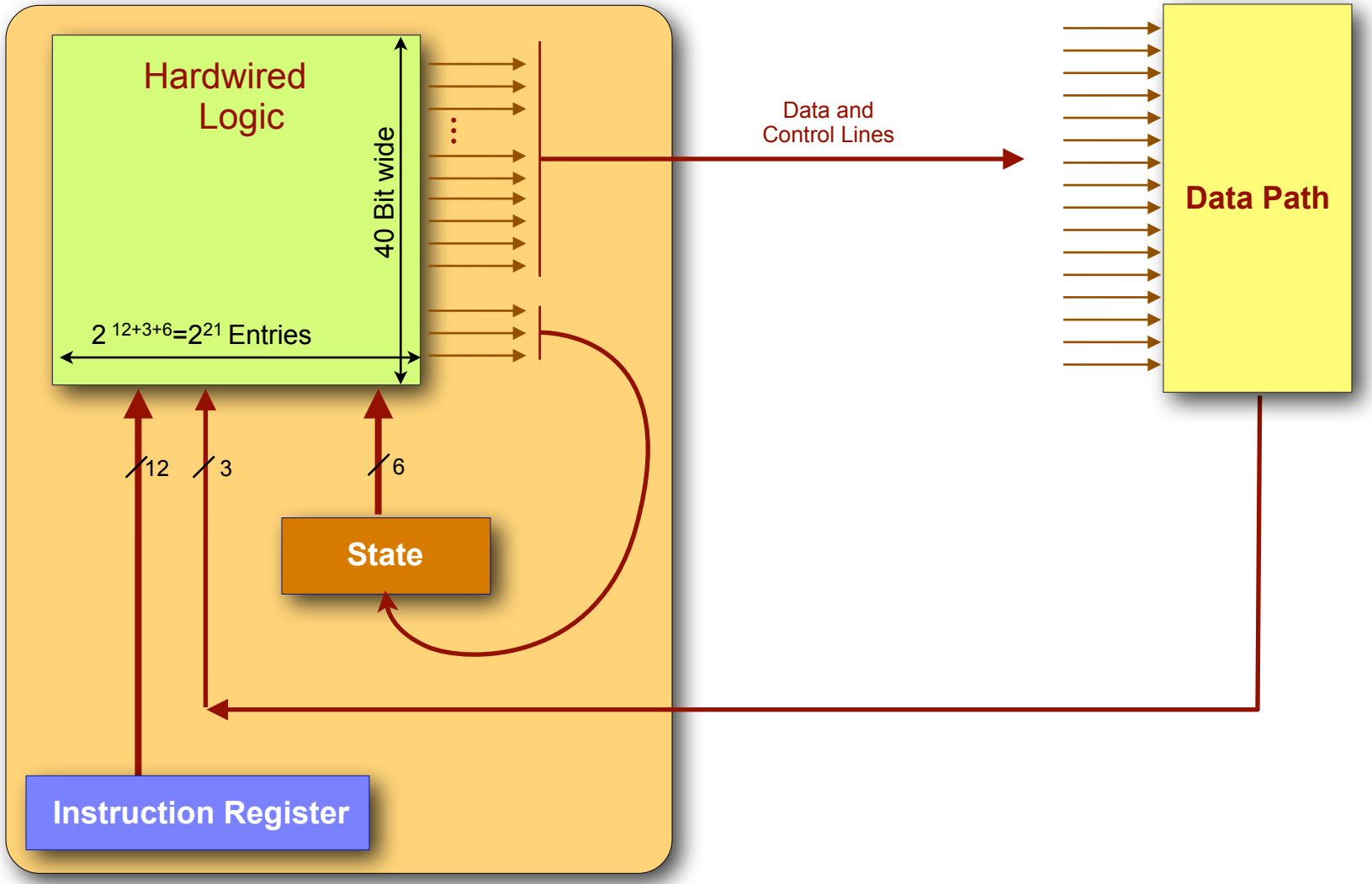
Resource Requirements (2)

- This table containes a lot of redundancy – f.e. the opcode bits are only used at the beginning of a new state selection instruction ➞ by reducing redundancy ➞ memory requirements for the table could be reduced.
- Hardware implementations of CU realized this way use PLAs ➞ application of logic minimization methods possible.

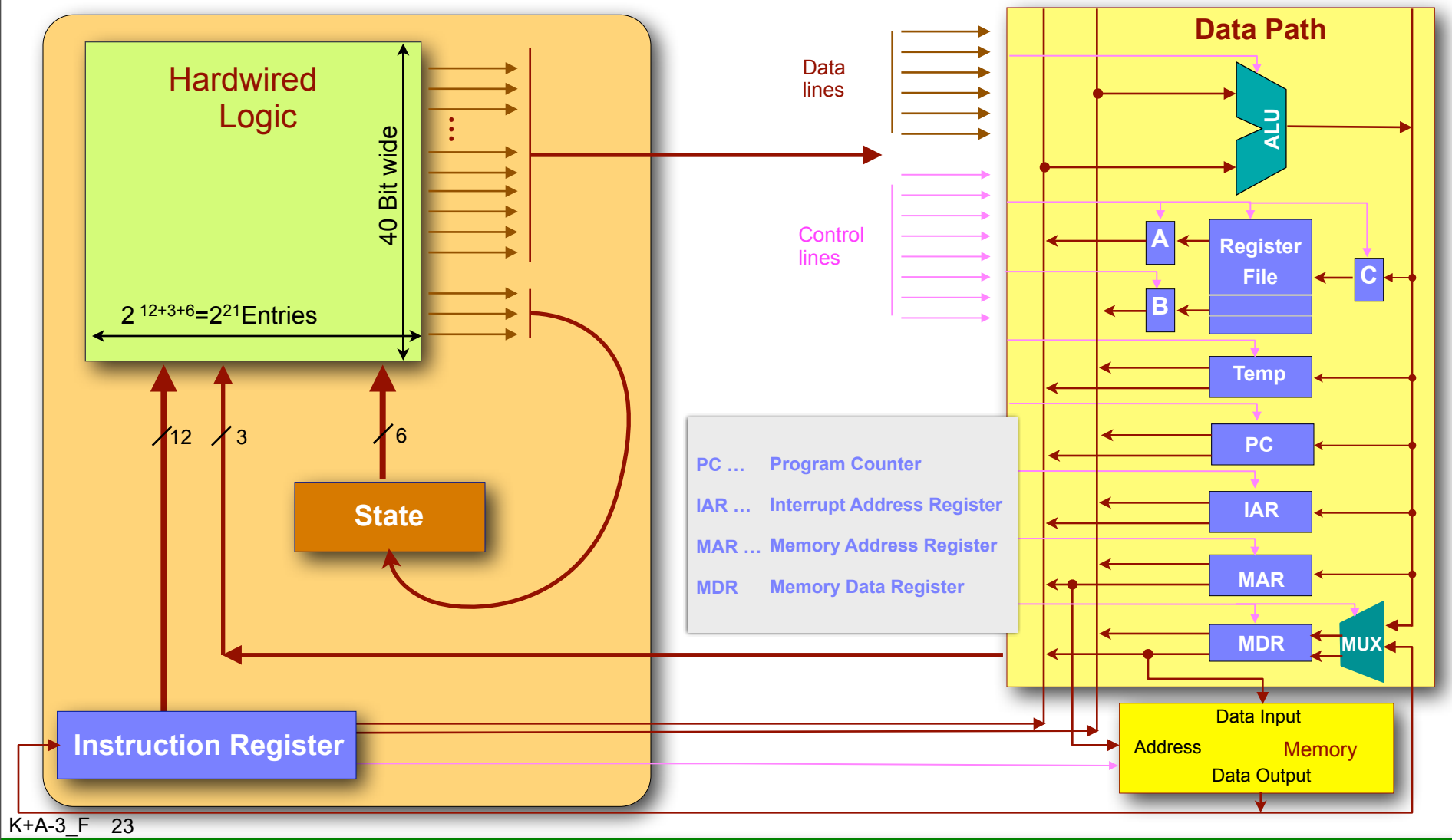
  → Redundancy decreases.

  → But logical complexity addressing control words increases.

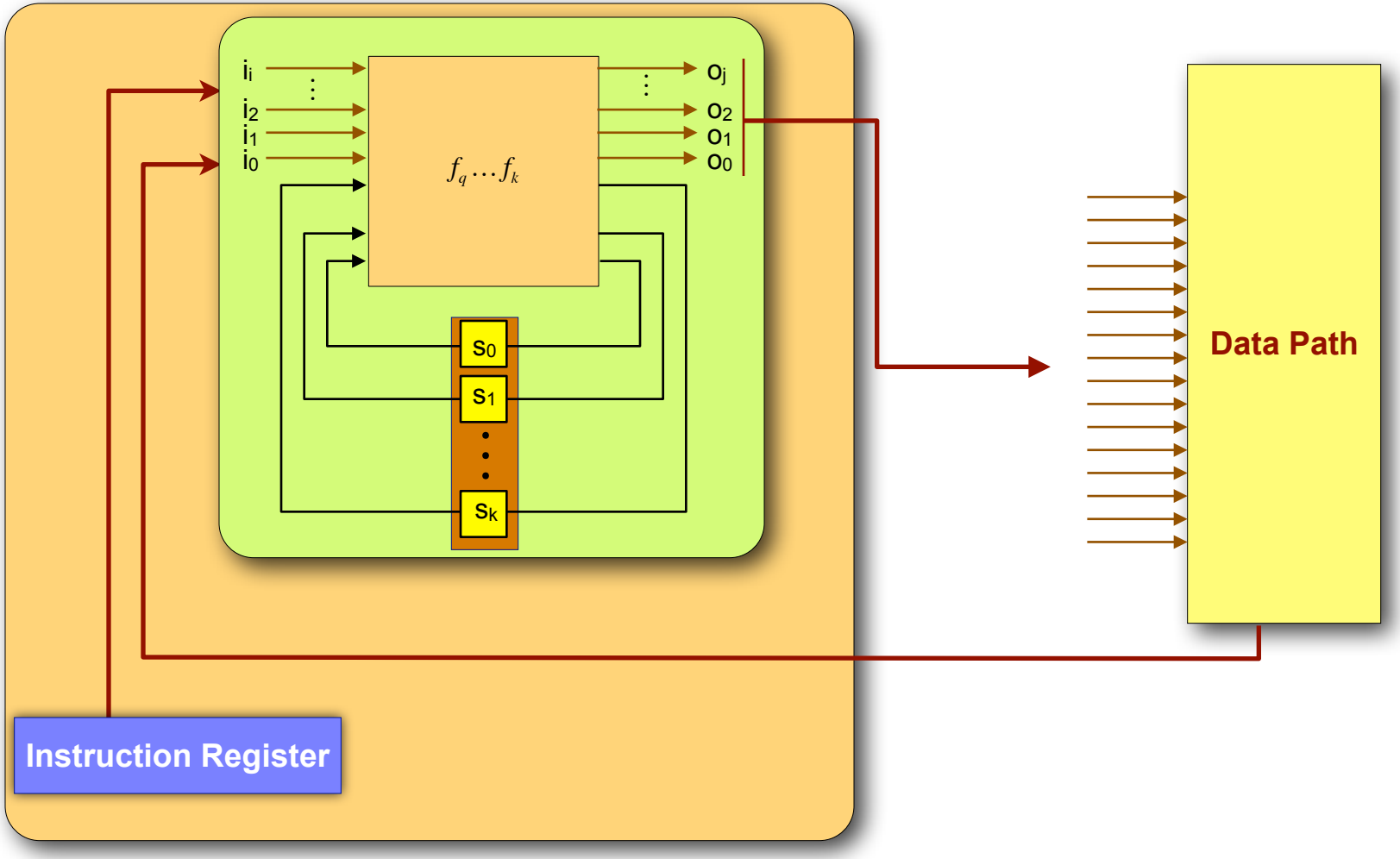
Principle of a Hardwired Control Unit (1)



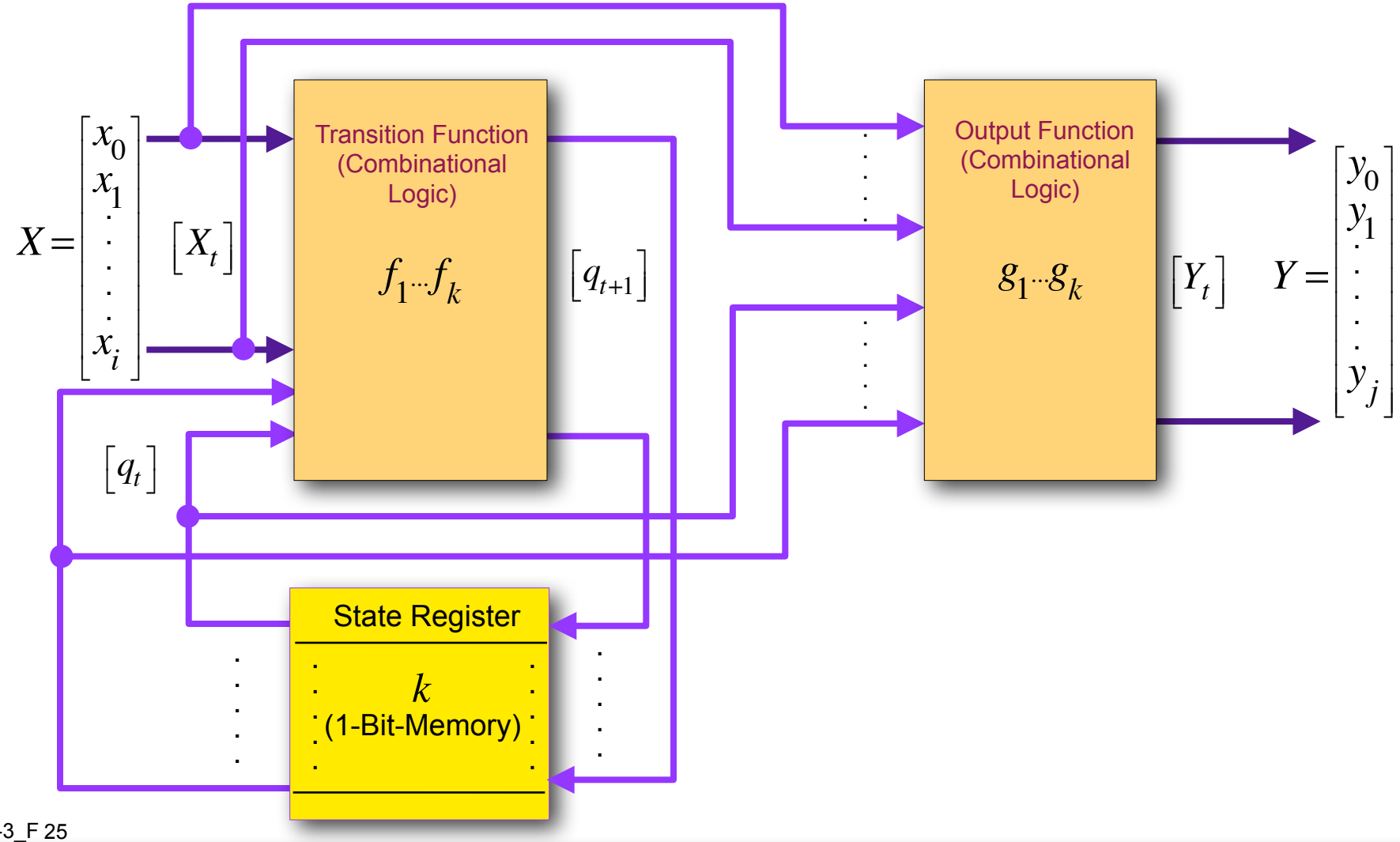
Prinziple of a Hardwired Control Unit (2)



Control Unit = Sequential Logic System



Sequential Logic System = Mealy Machine (Huffmann Model)



K+A-3_F 25

General (1)

- Controlling activities – necessary for execution of a macro instruction – are realized by a „micro program“ consisting of „micro instructions“.
- Micro instructions:
  - result from a single macro instruction, provide the real control bit pattern for the CU („control word“ + next address in µp memory).
  - are binary coded words as macro instructions, however by different „word widths“ (20 … >100 bit).
  - A single macro instruction activates mostly a complete sequence of micro instructions → micro program.

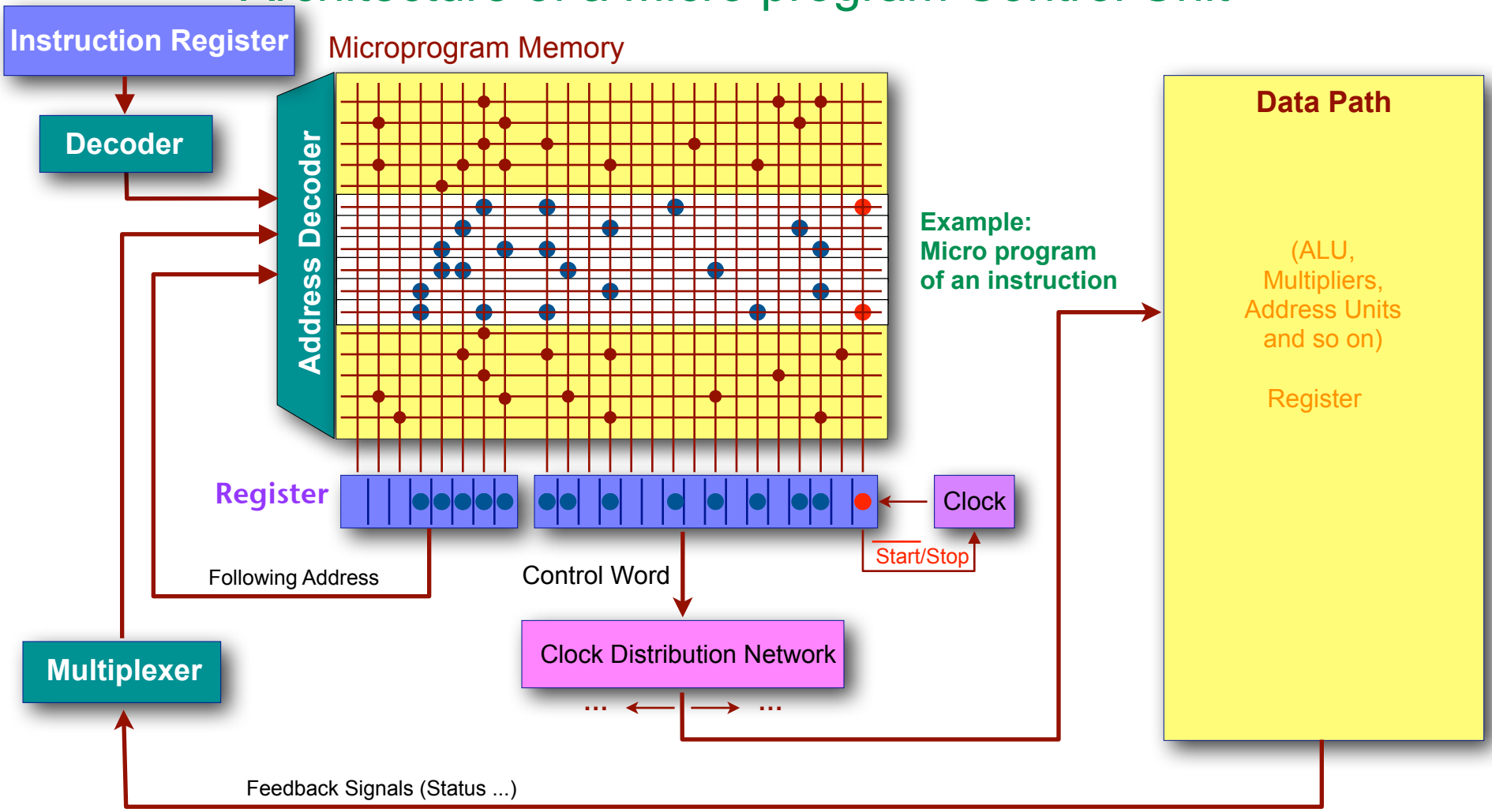
General (2)

- Micro program:
  - Representation of a hardware algorithm that has to be executed from the functional units of processor hardware.
  - Consists of micro instructions controlling each machine action with accurate timing.
  - Implemented on lowest hardware level of the processor.
  - A functional unit in the CU supplies the micro instructions retrievably and generates them immediately .
  - Supply takes place by means of an addressable micro code memory (ROM, EPROM) or with a PLA respectively, generated from a Mealy automate.
- Each macro instruction contains its own micro program.
- Corresponding to the complexity of the macro instruction → different number of micro instructions.

Operation Principle

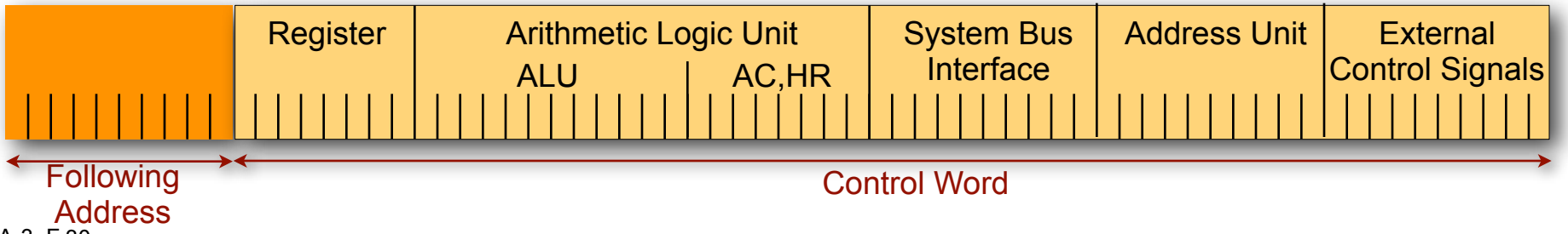
- The opcode of macro instruction will be interpreted („decoded“) from the instruction decoder → decoder supplies the initial address of micro program (in µp memory) which belongs to the current macro instruction.
- The micro instruction output (= lines/rows in µp memory) is driven by a clock signal.
- If the last micro instruction has been executed → a completion signal (“macro instruction executed”) is provided to the instruction decoder → clock signal of µp CU stopps.

Architecture of a Micro program Control Unit



Basic Architecture of a Micro instruction

- The control word is divided into separate fields.
- The single bits correspond to micro operations, i.e. selection (select), enabling (enable/disable) signals for data paths and required components (f.e. the last bit of each control word affects the clock in the µp CU).
- By logic combination of signals with processor clock → generation of specific control clocks, whose edges define the switching time → thereby predetermined chronology /timing sequence of component control.
- Components themselves supply feedback signals → that implies (f.e. over MUX) state-depended generating of subaddresses for the µp-memory → state-depended branches in the µp are possible.



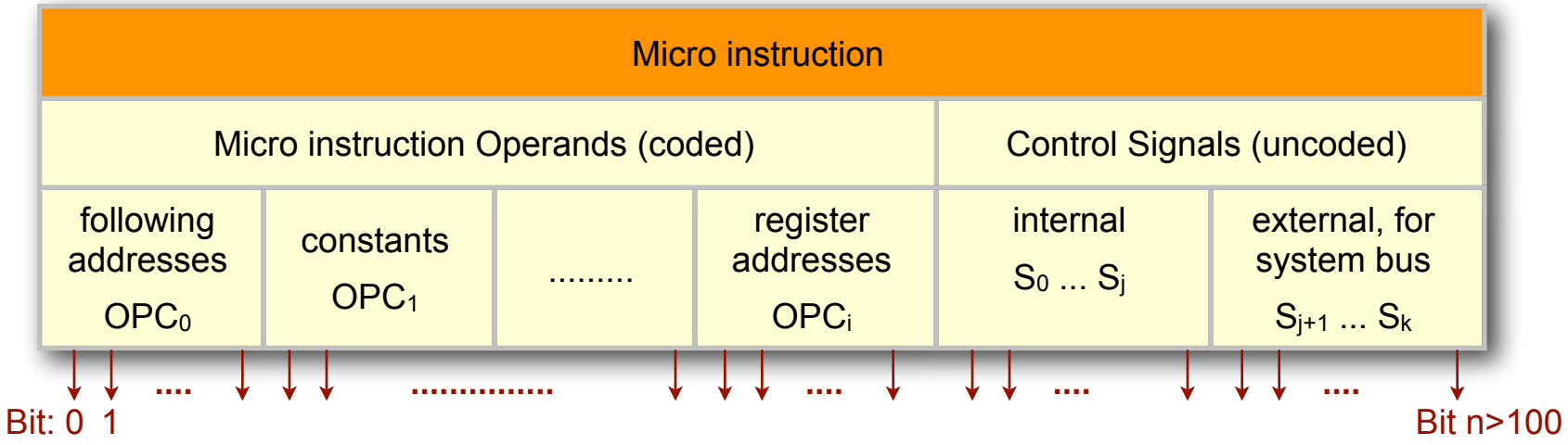
K+A-3_F 30

Horizontal Microinstruction Structure

- A specific control function is associated to nearly all bits of the instruction format → wide micro instruction format (f.e. more than 100 bits).

→Large micro instruction memory is necessary.

- Micro instruction operands are provided in encoded form OPC_0 to OPC_i in the micro instruction register (MIR).
- Control signals S_0 … S_k are active at processor's internal action point (f.e. „gates") and on external system bus.



Reduction of the Micro instruction Width

- To reduce the width of micro instructions (and thereby the size of micro instruction memory) → coding of control signals (assigned to the separate functional units) into groups (can act concurrently, too).

→ The shortened format is also called „quasi-horizontal“ microinstruction format → subsequent decoding is required → additional gate delays.

- Extreme case: „vertical“ micro instruction format → the control signal groups (generated with quasi-horizontal format) on their part affect the code for even finer micro instructions → even smaller word width (f.e. 40 bit).

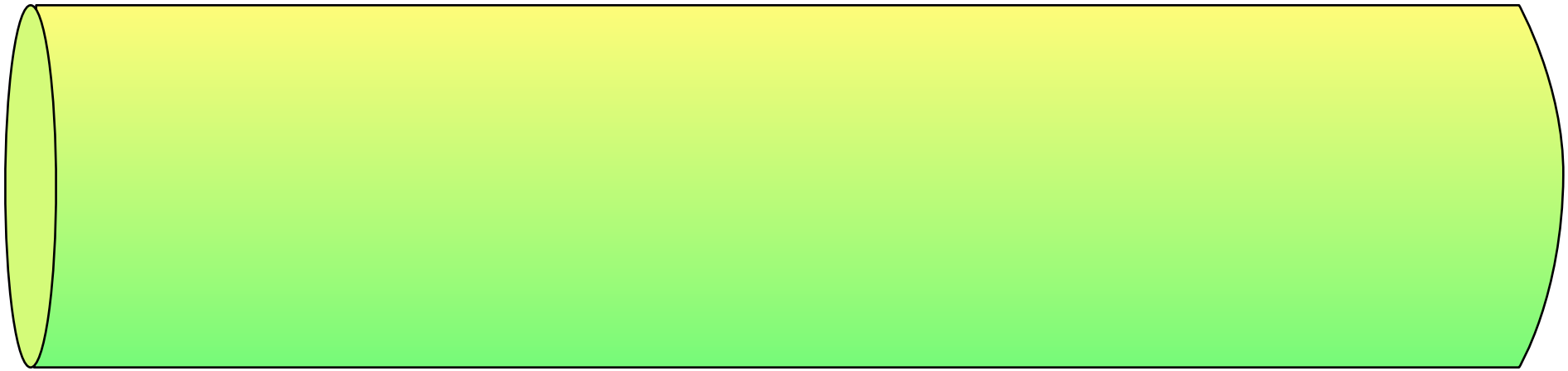
Advantages and Disadvantages

- Micro programmed control units
  - Primary in CISC architectures.
  - Micro instructions are fetched from a micro code memory.
  - Modifying micro code is rather simple possible → easy processor update (f.e. modified instruction set).
- Hardwired control units
  - With each clock a new micro instruction will be provided → CU very fast → especially suited for controlling pipelines into RISC and super scalar processors (with least possible low CPI values).
  - Micro instructions are generated by Mealy automates → more hardware, but faster; greater HW effort because of a reduced instruction set (micro instruction effort) is partially compensated.
  - "Sequential control" → hardwired → not changeable/only with large effort changeable → instruction set of the processor cannot be changed.

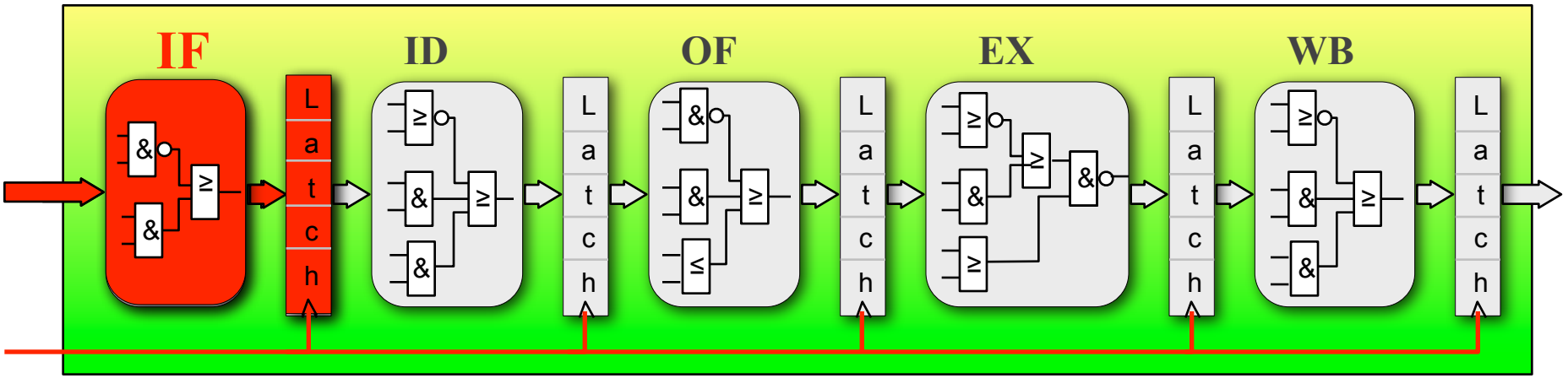
Principle of a Pipeline (1)

- The execution of an instruction occurs in phases (see above 3.1.1) and takes as long as all phases have been passed → this leads to an inevitable restriction of execution speed.
- Each phase requires specific hardware.
- A pipeline can be considered as the serial configuration of this „phase HW“.
- Pipelining is an implementation method, whereby several instructions at the same time are executed with overlap, it uses parallelism between the instructions in a sequential instruction stream.
- I.e. each pipeline level (pipe stage, pipe segment) represents (in hardware) an instruction execution phase – in such a way finishing a part of the instruction.

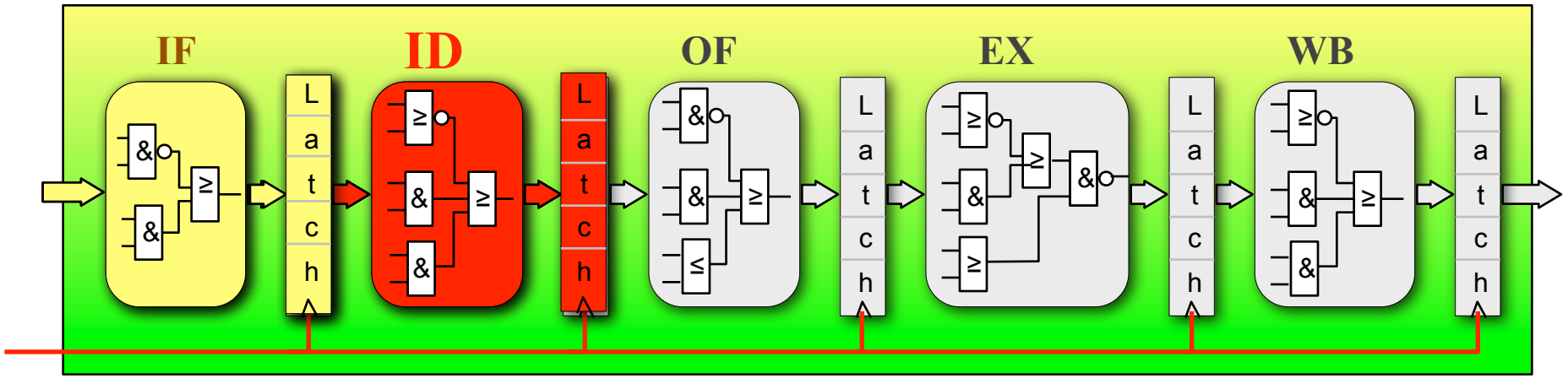
Principle of a Pipeline (2)



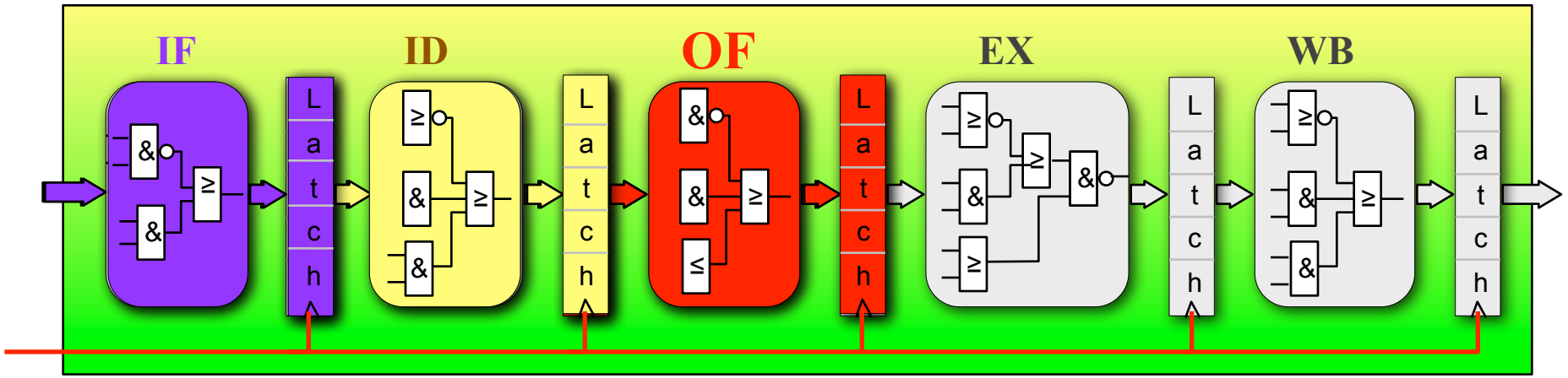
Principle of a Pipeline (2)



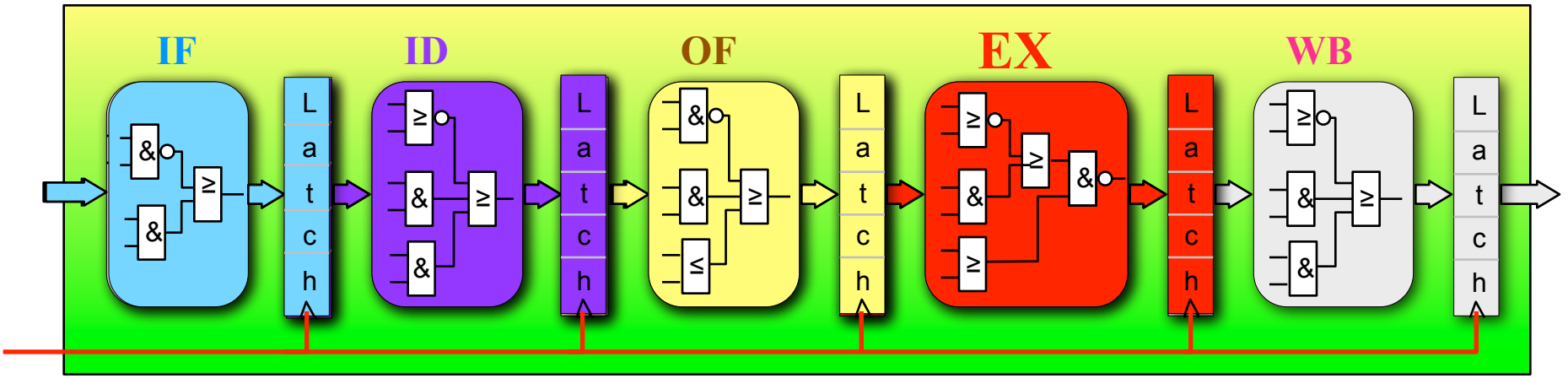
Principle of a Pipeline (2)



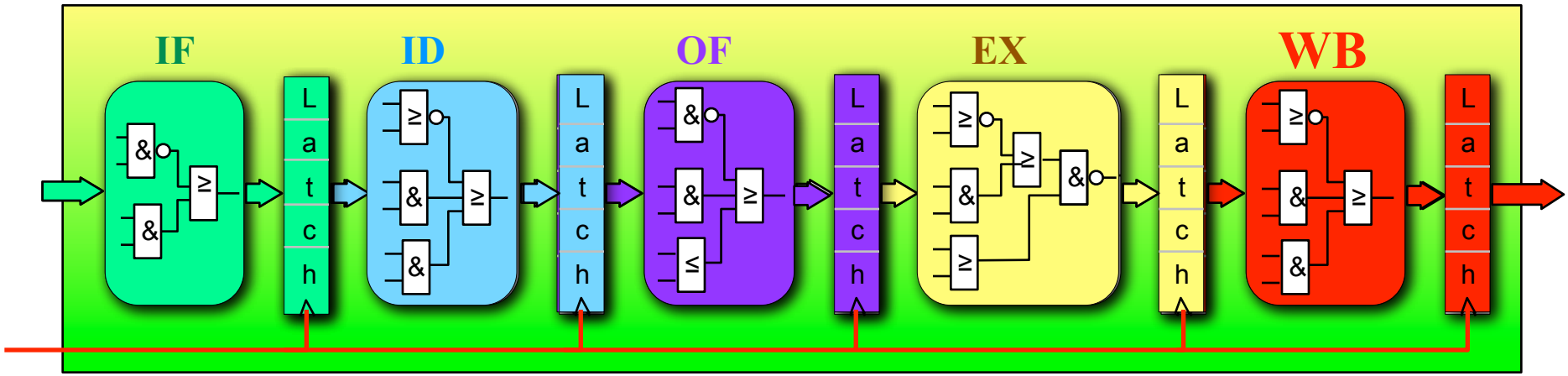
Principle of a Pipeline (2)



Principle of a Pipeline (2)



Principle of a Pipeline (2)



Principle of a Pipeline (3)

- As soon as IF stage is clear → the next instruction has to be fed in there … and so on. → Instructions enter the pipeline with IF → they are processed along its stages → leaving the pipeline after WB (completely executed).
- The required time for transport from one stage to the next is a machine cycle (mc).
- Processing inside the stages may take various time → the length of an mc is determined by the time required by the slowest pipe stage (because all stages continue at the same time).
- Because of overlap of the phases the processing speed can be improved → ideally, in an n-stage pipeline are n instructions to be processed.
- Timing mode of overlapping is controlled by clock.

Principle of a Pipeline (4)

- The throughput of the pipeline is thereby defined, how often instructions are outgoing from the pipeline.
- Pipeline stages can vary in complexity, depending on the complexity of the corresponding instruction execution phase:
  - F.e., some processors execute the actions *instruction fetch* and *decode* in one phase, the appropriate required HW will be more complex, so the needed time may be, too.
  - Some processors have only very small („fine granular“) phases.
- The simpler a phase the faster the processor can be clocked.

Aim of Designing a Pipeline

- Reaching a „balance“ between complexity (length, time requirement) of pipeline stages and their count → when this is reached, then it is under ideal conditions (f.e. no wait cycles):

$$time\ per\ instruction = \frac{instruction\ execution\ time\ \text{without pipeline}}{number\ of\ pipeline\ stages}$$

→ The increased efficiency with the pipeline would be = number of stages.

- Actually, increased efficiency is smaller (f.e. 10%), because stages are not balanced ideally + overhead with pipelining itself + delay by inserted latches.
- Also pipelining increases the throughput of processor instructions, (number of terminated instructions per time unit) → reducing the middle execution time per instruction (the program runs faster), but it reduces not the execution time of a single instruction.

→ Limits of pipelining: pipeline depth or clock frequency.

Short vs. Long Pipelines

- Short Pipelines consist of 3 … 5 stages, very long pipelines have more than 12.
- The single operations, which are executed by the processor in each stage differ depending on processor type – the easier these operations are, the faster the pipeline may be clocked.
- More complex operations thereby can be subdivided into multiple sequenced pipeline stages.
- “Super pipelines”: pipelines with very much – but simple stages.

Pipelines: Example (1)

- 3 stage pipeline:
  - Instruction fetch, (IF): The instruction is loaded from the instruction cache into the instruction register.
  - Instruction decode, (ID): The control unit decodes the instruction.
  - Executing phase, (EX): The required operation (arithmetic or logic) is executed by the arithmetic logic unit, the result is transferred into the register set (write back, WB).

  Depending on the kind of implementation, the operands have been fetched in ID or EX.

- 4 stage pipeline
  - IF
  - ID
  - EX
  - separate WB

Pipelines: Example (2)

- 5 stage pipeline:
  - IF
  - ID: CU decodes instruction. Operands (in registers) are loaded. Branch/jump instructions are terminated (soonest) at the end of this phase → passing the following stages passively. If necessary, the pipe can be loaded from the branch/jump address again.
  - EX: Operation is executed from the arithmetic logic unit.
  - Memory access, MA, MEM: LOAD/STORE instructions access the internal data cache (if a data has not been found in cache → loading it from main memory → additional clock required). All other instructions pass this stage passively. The processing of STORE instructions has been completed with this phase. LOAD instructions have an additional phase → WB.
  - WB: The result of the operation is stored in a register. Instructions without storing the result pass this phase passively.

Principle of a 5 Stage Pipeline

Instruction	Clock Cycle								
	1	2	3	4	5	6	7	8	9
instruction i	IF	ID	EX	MEM	WB				
instruction i+1		IF	ID	EX	MEM	WB			
instruction i+2			IF	ID	EX	MEM	WB		
instruction i+3				IF	ID	EX	MEM	WB	
instruction i+4					IF	ID	EX	MEM	WB

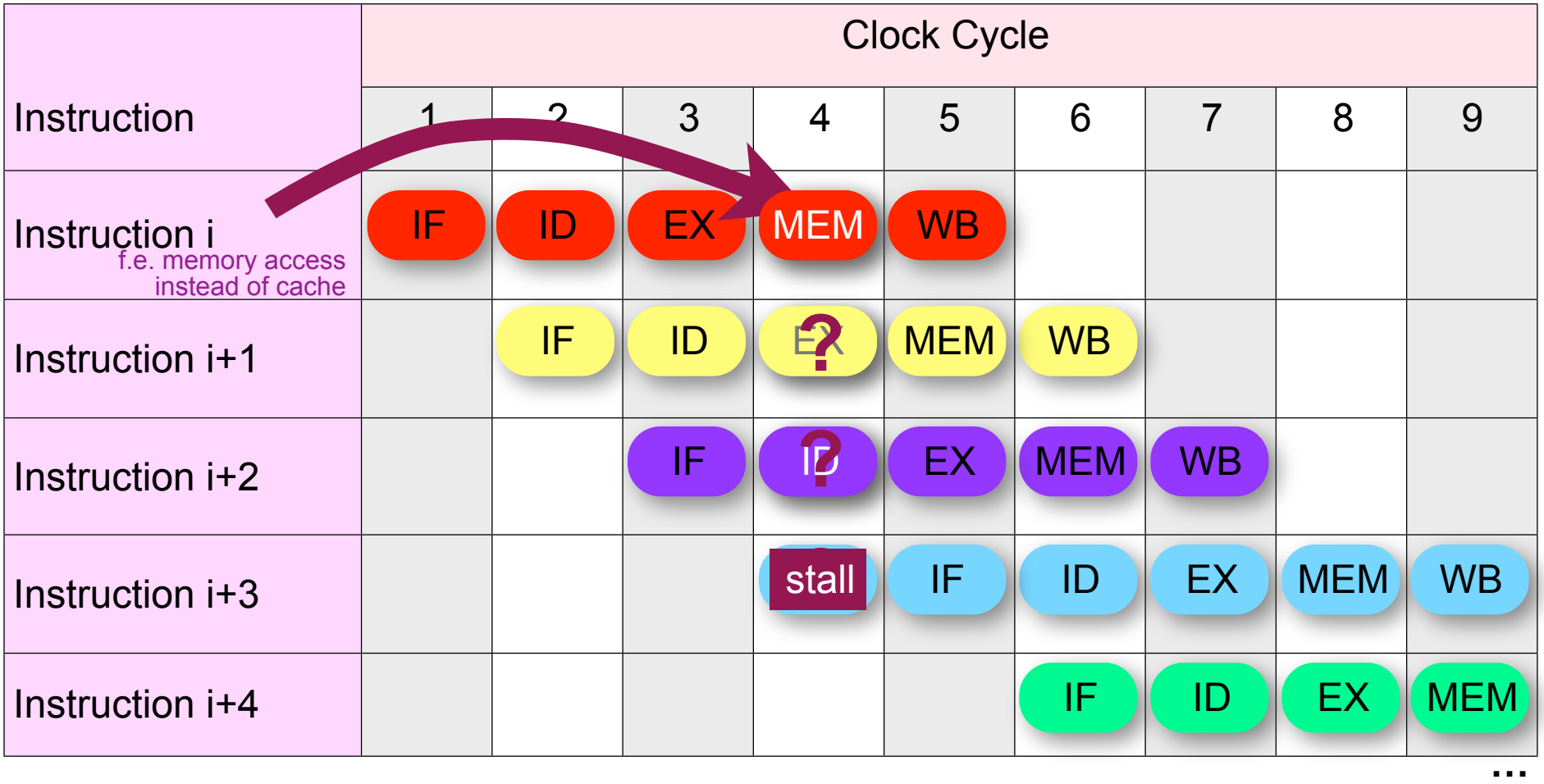
Reasons for Pipeline Hazards

- Structural hazard, (structural/resource dependency)
  Always appears, if 2 instructions simultaneously want to access the same processor component, which exists only once (bus, operating unit, register (in this case „name dependency“)).
- Data hazard, (data dependency)
  Occurs, if in an current instruction a data should be processed, which had not yet been calculated by previous instructions.
- Control hazard, (control dependency)
  Occurs associated with conditional branching, if – in an instruction – the predetermined condition has not been interpreted early enough.
- Hazards may cause wait cycles (stalls) in a pipeline.

Structural Hazards: Reasons, Effects

- Many reasons for this hazards, typical example: Memory access, f.e., if the CPU has only one memory port (Princeton architecture = program and data memory in one physical storage) or it has only one commonly used cache for program and data.
- The pipeline has to wait for one clock cycle, if the instruction contains a memory reference – or if a data has not been found in the internal cache whilst the MEM phase → CPU cannot fetch the next instruction, because the data access uses the memory port → wait cycle (stall) is inserted.
- While the instruction, which should be fetched, has to wait, all other instructions can be continued normally.
- After the memory port is free, the instruction, suspended until now, will be read into the pipeline.

Pipeline with Wait Cycle (Stall) by Structural Hazard



Data Hazards: Reasons, Effects, Example

- Primary purpose/effect of pipelining: reducing the total processing time of a program by temporally overlapped processing of the sequential listed instructions of this program.
- Data and control hazards may result from that.
- Data hazards occur, if a dependence exists between instructions that are located so close, that caused by the overlapping of the pipeline the sequence of access to the operands is changing (compared to the normal sequential order.
- Example:
  - ADD instruction writes a register, which is source operand for the SUB instruction.
  - ADD instruction completes writing of data into the register file only three clock cycles later, after SUB already started reading it.

Pipeline with Data Hazard

Example

ADD	R1,R2,R3
SUB	R4, R1, R5

ADD: **R1** stores result of addition from register contents of R2 and R3

SUB: R4 stores result os subtraction from register contents of **R1** and R5

Important: Of course, the value of R1 has to be known already at the time of reading in instruction SUB.

write data R1

instruction	clock cycle								
	1	2	3	4	5	6	7	8	9
ADD	IF	ID	EX	MEM	WB				
SUB		IF	ID	EX	MEM	WB			

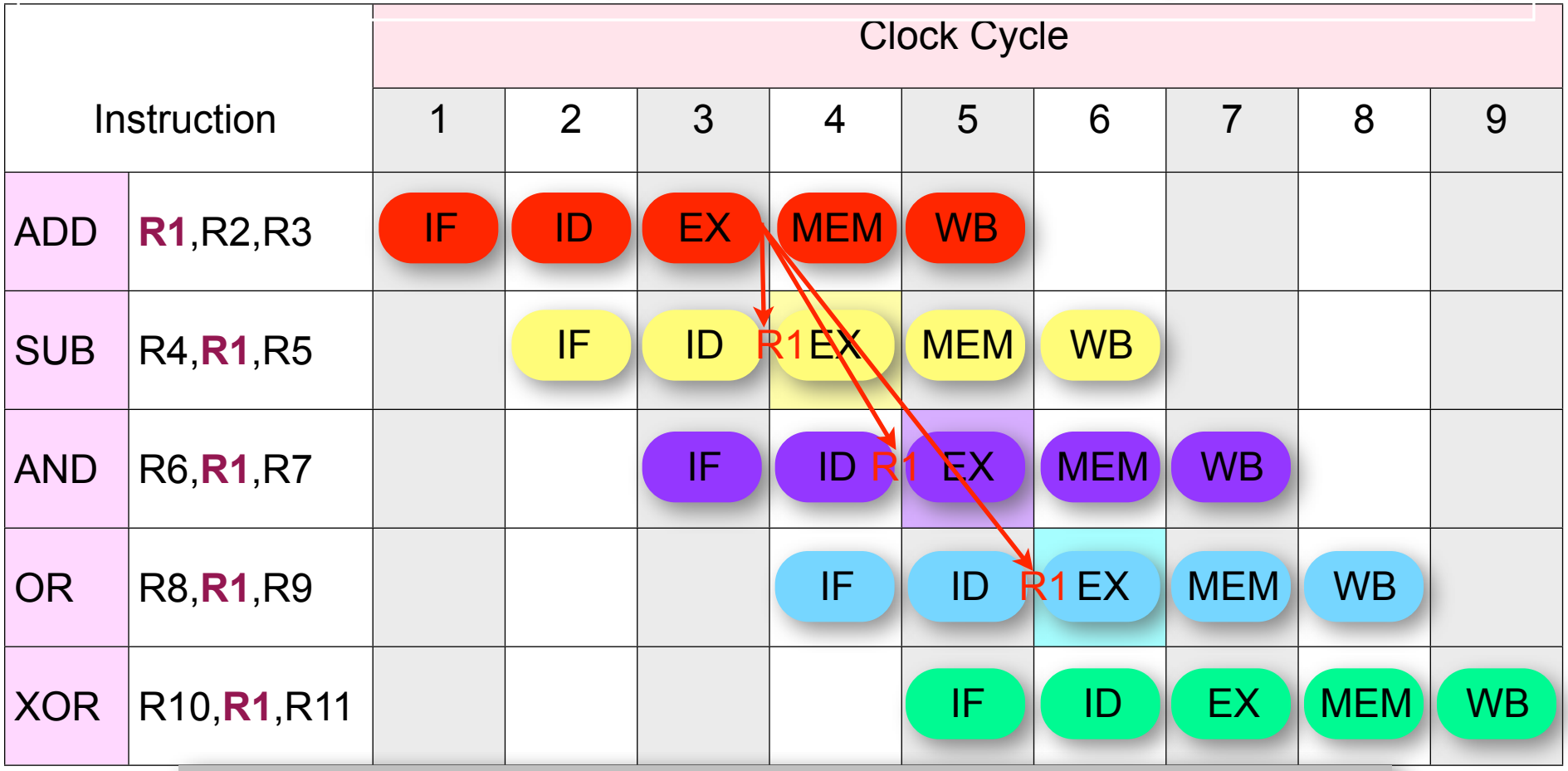
read data R1, R5

R1?

Pipeline with Data Hazard: Solution

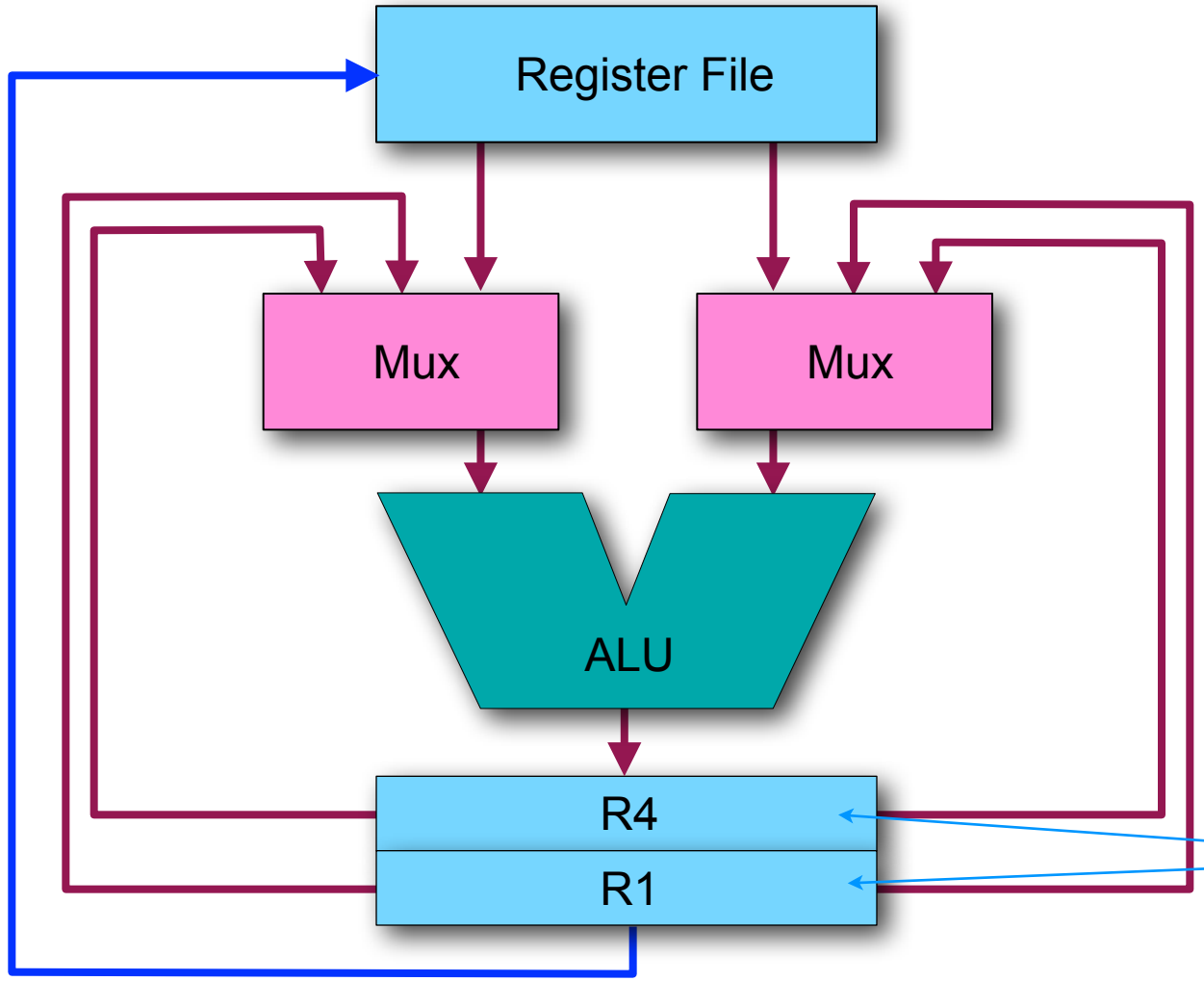
- Solution is possible by hardware extension: „Forwarding“ (also called „bypassing“ or „short circuiting“):
  - The recent ALU result is being always „transferred“ to the ALU input latch → supplied in advance (forwarded).
  - If the forwarding HW recognizes, that the just finished ALU operation has to write to a register, that is, for its part, source of the following operation → the control HW selects the now in a temporary register available result as ALU input value, instead of the value from the register file (wherein the operands are available usually).
- Forwarding can reach several instructions → complex circuit structure required.
- This results in several „bypass levels“ → each level requires a latch and a comparator pair, for testing, if the adjacent instructions have the same register for source and for destination.

Instruction Set in the Pipeline where Forwarding is necessary



The value of R1 must be provided in a buffer to the ALU input immediately, because the following instructions need R1 as an operand. R1 is available "officially" by WB of the ADD instruction – not until of the beginning of the XOR instruction, therefore, here is no need for forwarding.

Structure of an ALU with Bypass Unit



3 Types of Data Hazards (1)

- RAW (read after write) (the most general hazard type)
  Instruction j attempts to read a source, before it has been written by instruction i → hence j gets the former value, incorrectly.
- WAR (write after read)
  A data should be processed in an instruction, that has not yet been calculated by the preceded instruction. j tries to write a destination, before it has been read by i, therefore i obtains the new value, incorrectly. This hazard occurs if instructions exist, which write results in the instruction pipeline early and, other instructions are in the pipeline, which read a source after writing (f. e. autoincrement-addressing).

3 Types of Data Hazards (2)

- WAW (write after write)
  j tries to write an operand before it has been written by i →
  the completed write operations are executed in the wrong order, thus leaving the value from i incorrectly, instead of the value written by j in the destination. This is only possible in pipelines, which write in more than one stage, respectively, which allow an instruction to be continued, even when a subsequent instruction had been stopped.
- RAR (read after read)
  no hazard

Problems

- Not all data hazards can be handled without loss of performance.
- Some instructions (f.e. load instructions) have a delay time, respectively latency time, that cannot be eliminated only by forwarding; for this purpose the data access time should be 0.
- Most common solution: additional hardware („pipeline interlock“), that recognizes a hazard and causes the pipeline to wait (stall) until the hazard has been terminated.
- Blocking stops the pipeline – beginning with the instruction, that intends to use the data, until source instruction provides them.
- This delay cycle („wait cycle“, „pipeline stall“, „bubble“) enables data to be loaded to arrive from memory → they can be transferred now by forwarding.
- The CPI value for the waiting instruction increases by the length of wait cycle.

Effects of Wait Cycles on the Pipeline

Instruction		Clock Cycle 1	2	3	4	5	6	7	8	9	10
Arbitrary Instruction		IF	ID	EX	MEM	WB					
LW	**R1**,32 (R6)		IF	ID	EX	**MEM**	WB				
ADD	R4,**R1**,R7			IF	ID	stall	**EX**	MEM	WB		
SUB	R5,**R1**,R8				IF	stall	**ID**	EX	MEM	WB	
AND	R6,**R1**,R7					stall	IF	ID	EX	MEM	WB

All instructions will be delayed, beginning with the instruction, that has a dependency. Thereby the value arriving at MEM can be transferred to the EX of the next instruction, furthermore R1 is standing by in SUB and AND.

Static Scheduling

- So far: pipeline fetches instruction and transfers it unless there is a data dependency with an instruction already in the pipeline.

  If data dependencies exist ➡ they will cause wait cycles.

- The software (compiler) is responsible for the scheduling of instructions to reduce this waiting time, so called Static Scheduling.

Dynamic Scheduling

Dynamic Scheduling:

Hardware reduces wait cycles by reordering of instruction execution.

Advantages of dynamic scheduling:

- Treatment of cases where the dependencies are unknown at compile time.
- allows effective execution of code on another pipeline as for which it was compiled for.

Disadvantage:

- Significant increase of hardware complexity!

Dynamic Scheduling

Problem:

Transfer of instructions in normal order.

If an instruction stops the pipeline no other instruction can continue and Functional Units (FUs) remain unused.

Code−Segment:

```
DIVF F0, F2, F4
ADDF F10, F0, F8
SUBF F6, F6, F14
```

SUBF cannot be executed because ADDF depends on DIVF (wait cycles).

But SUBF is independent of these two instructions!

This limitation can be avoided by Out-of-Order-Execution of instructions.

Out-of-Order Execution

Problem: WAW−Hazards, WAR−Hazards

New Code−Segment:

```
DIVF F0, F2, F4
ADDF F10, F10, F8
SUBF F10, F6, F14
```

SUBF cannot be executed because ADDF depends on DIVF (wait cycles).

But SUBF is independent of these two instructions!

WAR-Hazard: ADDF must read F10 before SUBF may write!

WAW-Hazard: SUBF must write result to F10 after ADDF!

Hazards need to be treated by control logic.

Out-of-Order Execution

Problem: WAW−Hazards, WAR−Hazards

Two methods:

- Scoreboarding: Centralized authority
- Tomasulo−Algorithm: Decentralized logic

Scoreboard

Scoreboarding − Method to allow an out-of-order execution if enough ressources are available and no data dependencies occur.

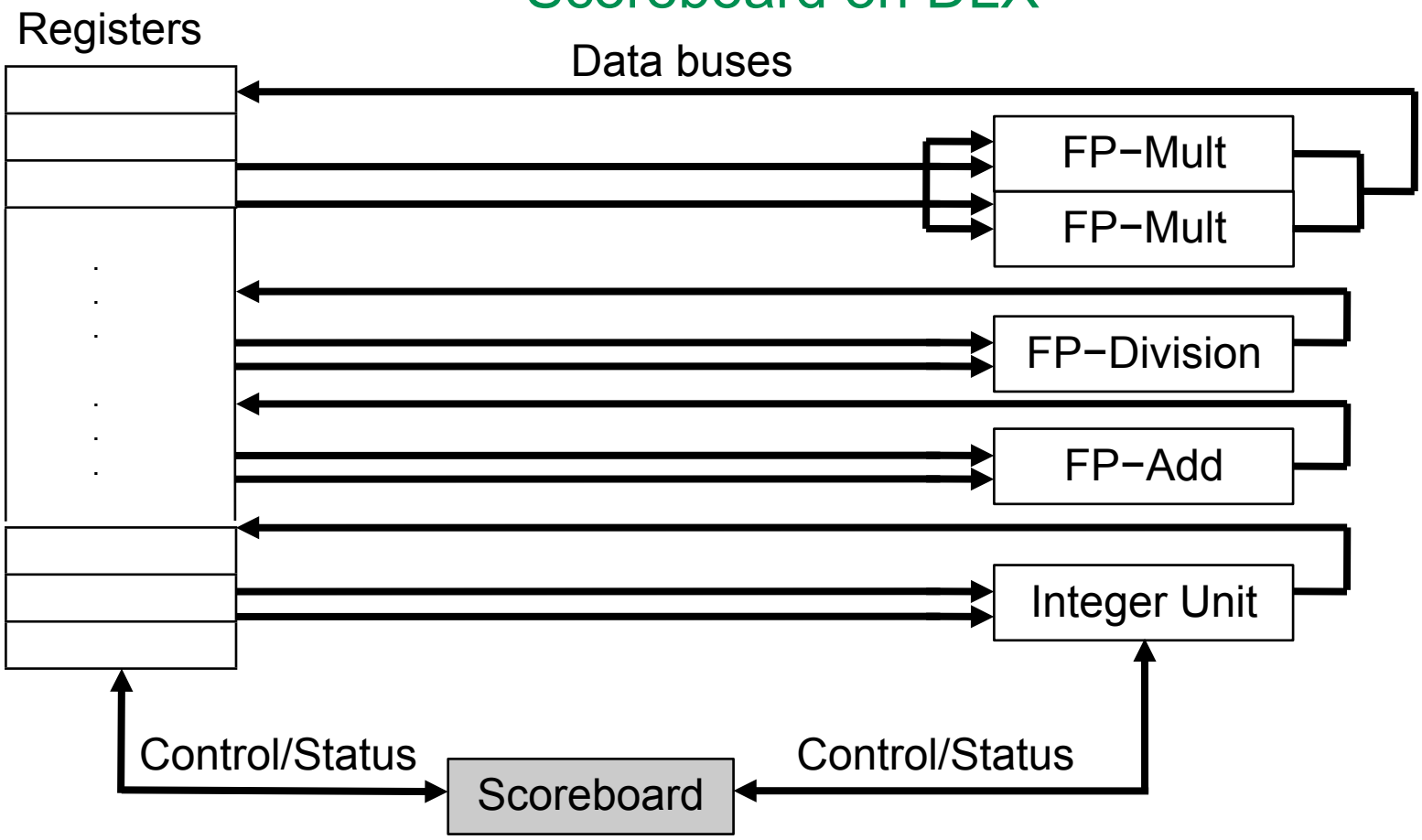
Avoid WAR-Hazards! Two rules:

1. Read registers only in stage Register Read!
2. Store operations and operands in a queue!

WAW-Hazards:

Must be testet! If necessary – introduction of wait cycles.

Scoreboard on DLX



Multiple instructions may be in EX−stage in parallel!

→ Multiple functional units or pipelined FUs necessary.

Tomasulo Algorithm

Applicated for the first time at IBM 360/91−FP−Unit about 3 years after CDC6600, named after Robert Tomasulo, a member of the design crew at IBM

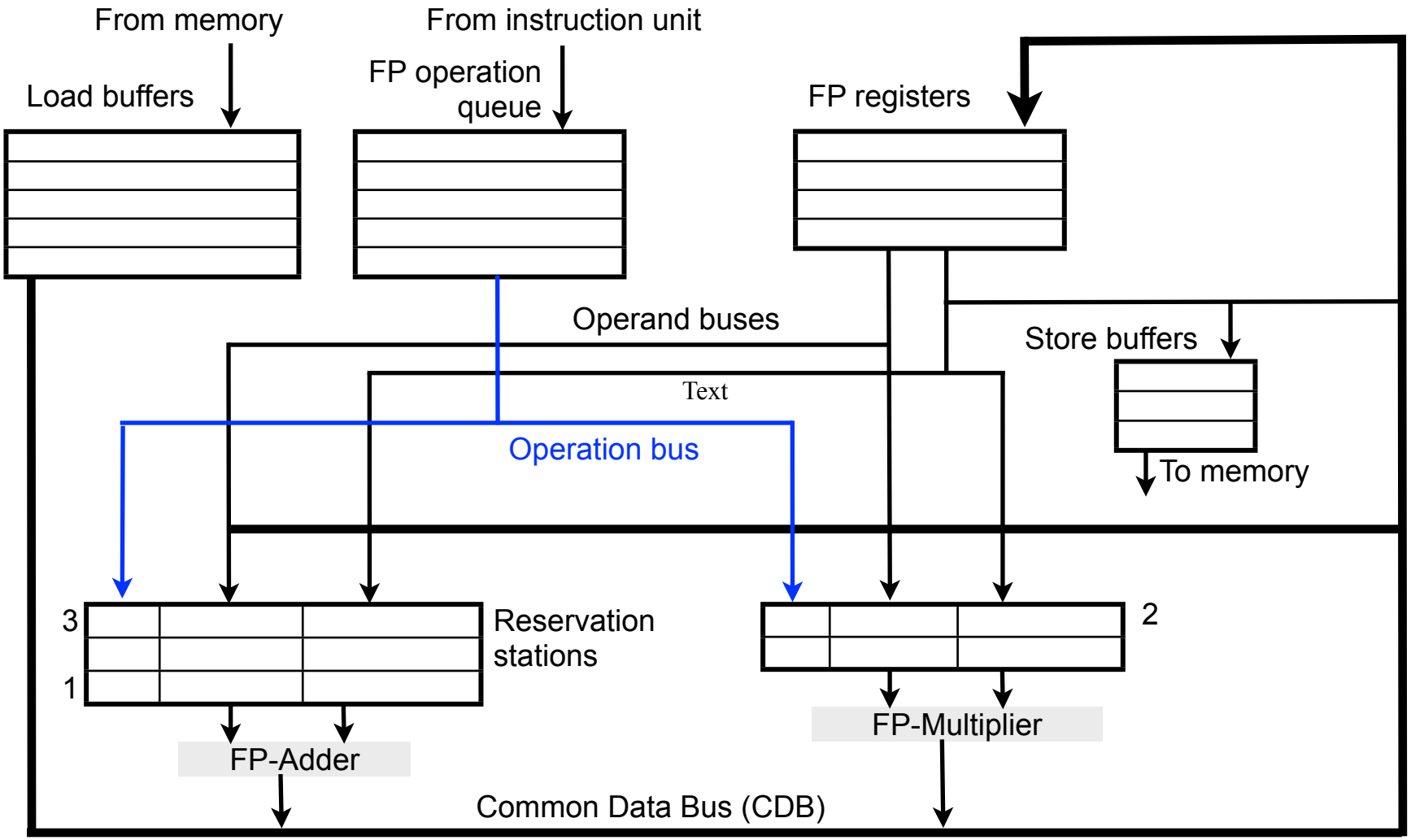
− focusses on floating point unit,

− uses automatic register renaming to avoid data hazards.

Two main differences to Scoreboard

1. Hazard-detection and execution control are distributed − reservation stations (tables) at every FU inform the control unit if an instruction may start its execution at this FU.
2. Results are transmitted directly to FUs using the Common Data Bus (CDB) without passing the register file.

Structure of an FP unit using Tomasulo Algorithm



K+A-3_F 68

Tomasulo Algorithm

Advantages

- Distribution of hazard detection logic.
- Elimination of wait cycles for WAW and WAR hazards by register renaming.

Disadvantages

- Large hardware.
- Single bus (CDB) limits performance gain.
- Load- and Store-instructions may be executed in different order. (It must be ensured that they access different addresses ➡ test in memory buffer.)

Control Hazards: Reasons, Effects

- Can generate a larger loss of performance than data hazards.
- They occur associated with conditional branches, where the condition is satisfied → program flow is broken and has to be continued at another program position corresponding to the branch condition.
- Earliest after analysis of the status register (at end of decode phase) – it is definite, if this case exists (with several processors the status register is being read in the decode phase, but it will be evaluated only later in the executing phase).
- The address of the branch destination will be calculated now (in the executing phase), then, the program execution continues from there.
- This address will be stored in the program counter in the write back phase.
- Unfortunately, the pipeline holds already instructions at this point, (the following to the branch instruction). These are, of course, obsolete now.

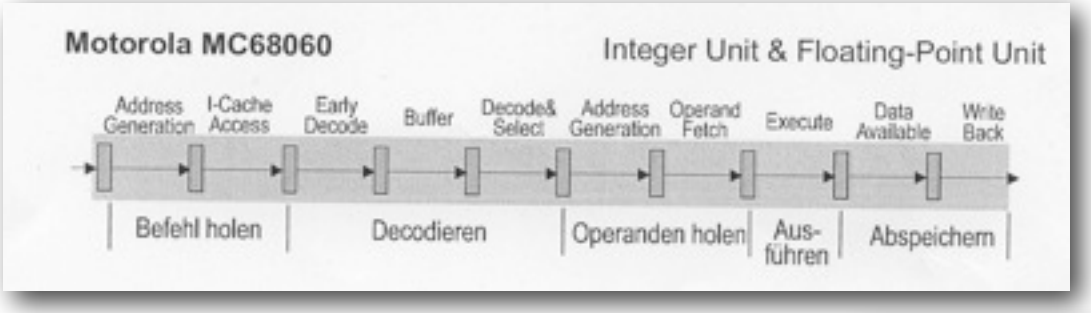
Solutions (1)

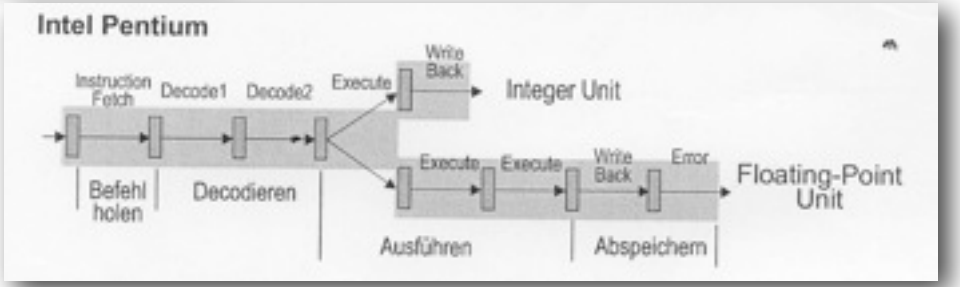
- Simplest possibility: Unnecessarily loaded instructions have to be removed („*pipeline flushing*“) → loss of performance, because the already executed sub-process of removed instructions has been wasted.
- The latter could be moderated thereby, that the compiler places after each branch instruction an adequate number of NOP instructions (with many RISC processors the pipeline is organized in such a way, that one needs only 1 NOP after the branch instruction) → this kind of realization of conditional branches is named „delayed branch“.
- Optimizing compilers try to replace the NOPs with instructions, which have no effects on the branching decision and which have to be executed from that in any case.
- Appearing delays in the execution processing which result of these conditional branches are called „*branch penalties*“.

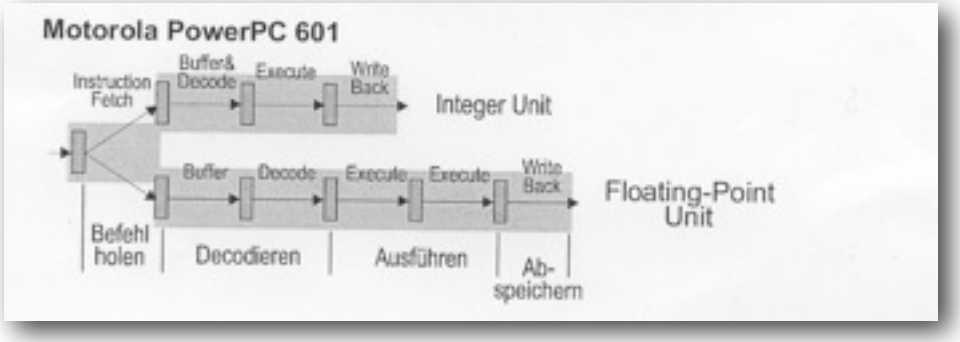
Solutions (2)

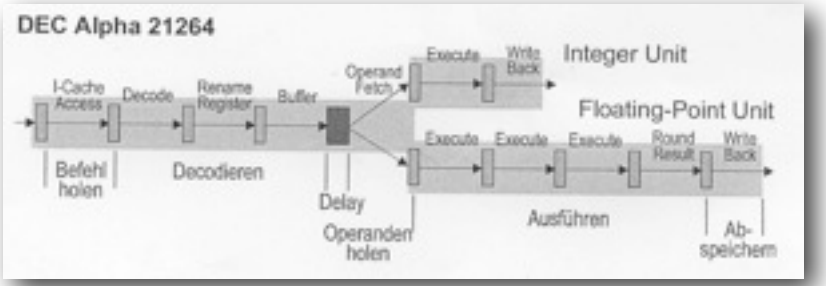
- Another solution: After the branch instruction → read and preventative decode of both possible following instruction sets → then even faster switching.
- With some processors (f. e. some DSPs) the programmer can define, if in case of a branch *flushing* or *delayed branching* should be used.

Pipeline Examples







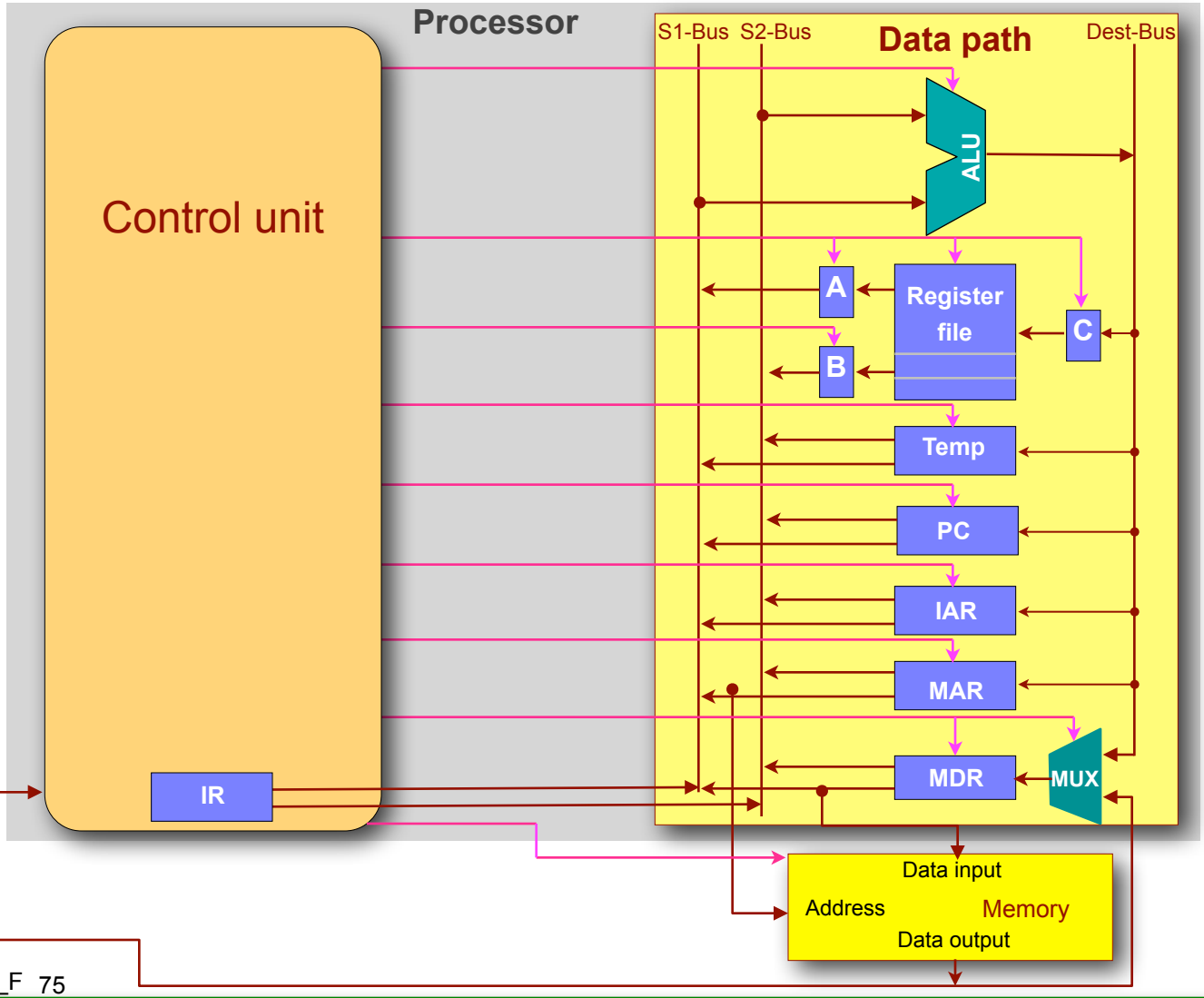


K+A-3_F 73

General

- Simple processors often have only one ALU, which executes all arithmetical and logical operations required from CU → arithmetic logic unit (ALU).
- More complex processors often have multiple, independently operating ALUs for special tasks, f. e. FPU, MMX, frequently used ALUs are several times present, occasionally.
- ALU examples:
  - ALU for integer and floating point numbers (Integer Unit, Floating-Point Unit
  - fast parallel multiplier and divider
  - shifting and rotating units (shifter, rotator)
  - ALU for single bit and bit array processing
  - ALU for graphic and multimedia operations (MMX (MultiMedia Extension), ALU on graphic cards)
- ALU is also called „data path“, because data are combined arithmetically/ logically, this includes additional modules (e. g. registers, …).

Example: Processor (DLX) – 2 Blocks: CU and ALU (Data path)



IAR – Interrupt Address Register
MAR – Memory Address Register
MDR – Memory Data Register
IR – Instruction Register
PC – Program Counter

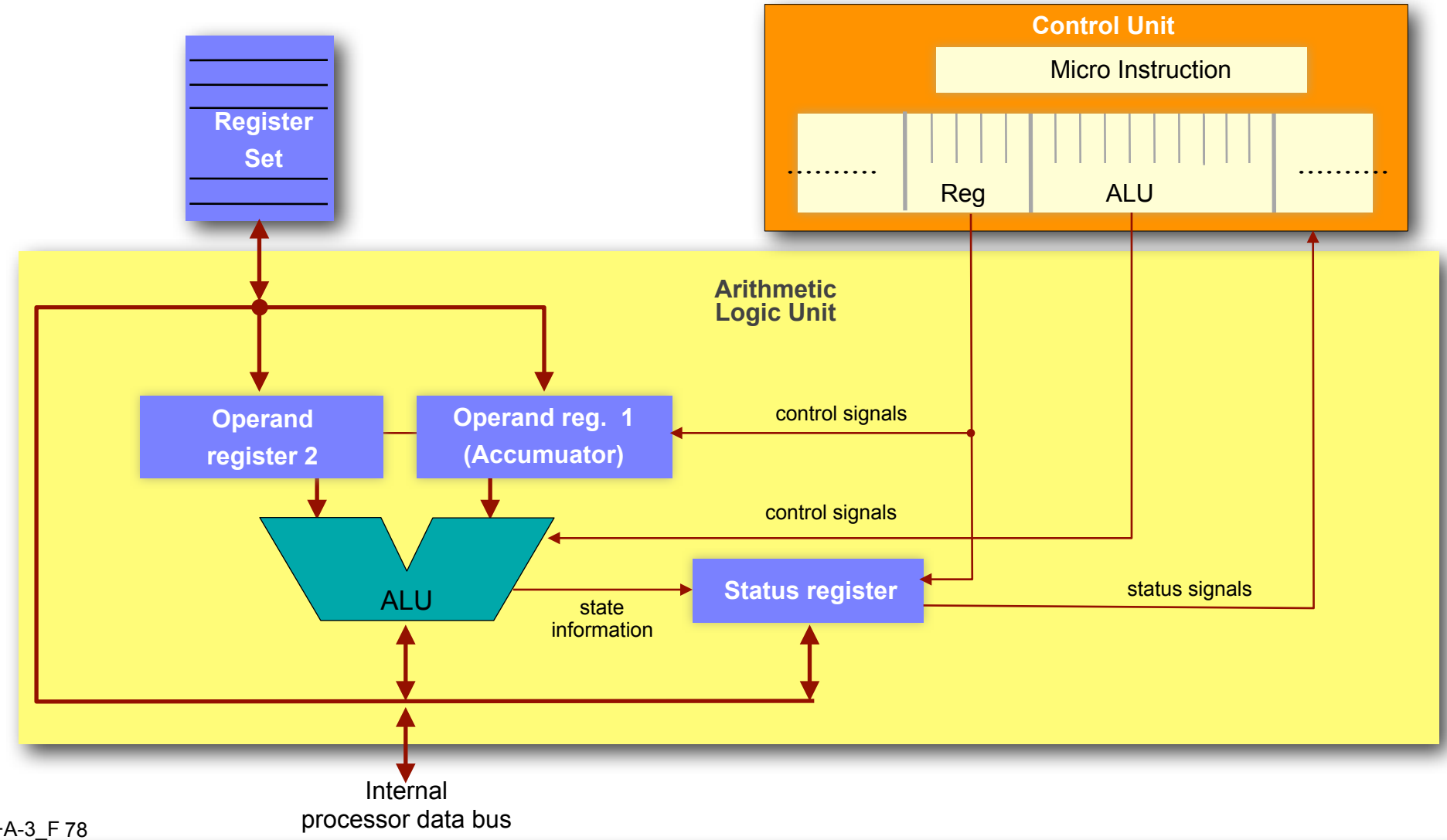
Parallel Arithmetic Logic Units

- There are parallel operations using bit formats with different width as bytes, double bytes, ...
- All concerned bit positions are handled simultaneously/concurrently.
- This provides a higher data throughput per time compared with serial operating arithmetic logic units.
- Disadvantage: more hardware needed.

Serial Arithmetic Logic Units

- Each single operand bit is processed separately.
- Occurring carries must be buffered at least about one clock period.
- Clocked circuitry is required.
- Operating time of serial arithmetic logic unit is larger, especially with larger operand formats.
- Operating time sometimes can be dependent from the structure of the operands with some serial ALUs (distribution of 0s and 1s).
- Advantage: (mostly) lower implementation effort.

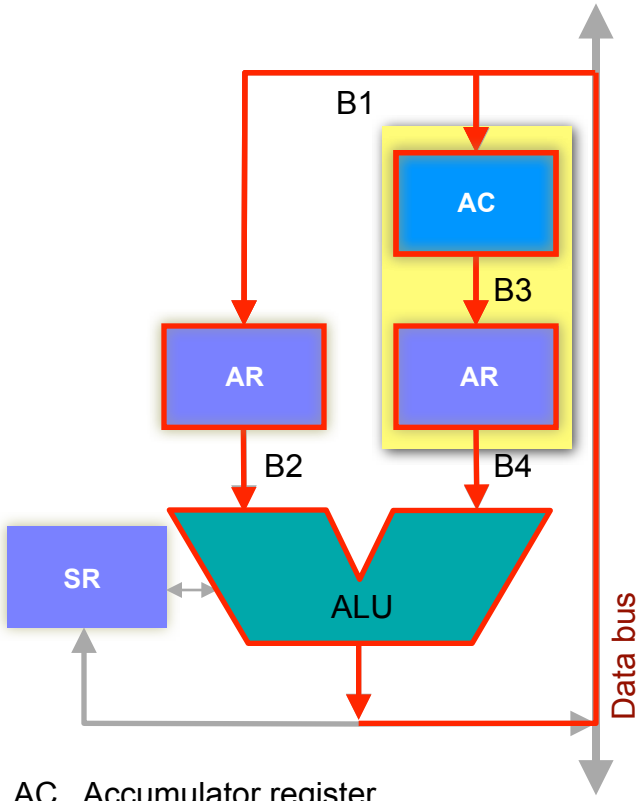
Integer Arithmetic Logic Unit: Structure



Integer ALU: Principle

- Consists of the ALU itself, some auxiliary registers and the status register.
- The ALU combines the two operands, located in 2 auxiliary registers (as intermediate memory) and gives back the result, mostly into the accumulator (one of the two auxiliary registers).
- If there is no result at ALU output recently, the output is set in high-resistance state (tristate), so that the internal bus can be used from other CPU components.
- ALU itself has no memory cells → pure combinational logic.
- Configuration of the required combinations to be realized in accordance with the instruction occurs over corresponding wires by signals, issued from the CU → ”*operation stock*”.
- Information about the state of the final result is transferred through the status register (summary of separate flags) to the CU.

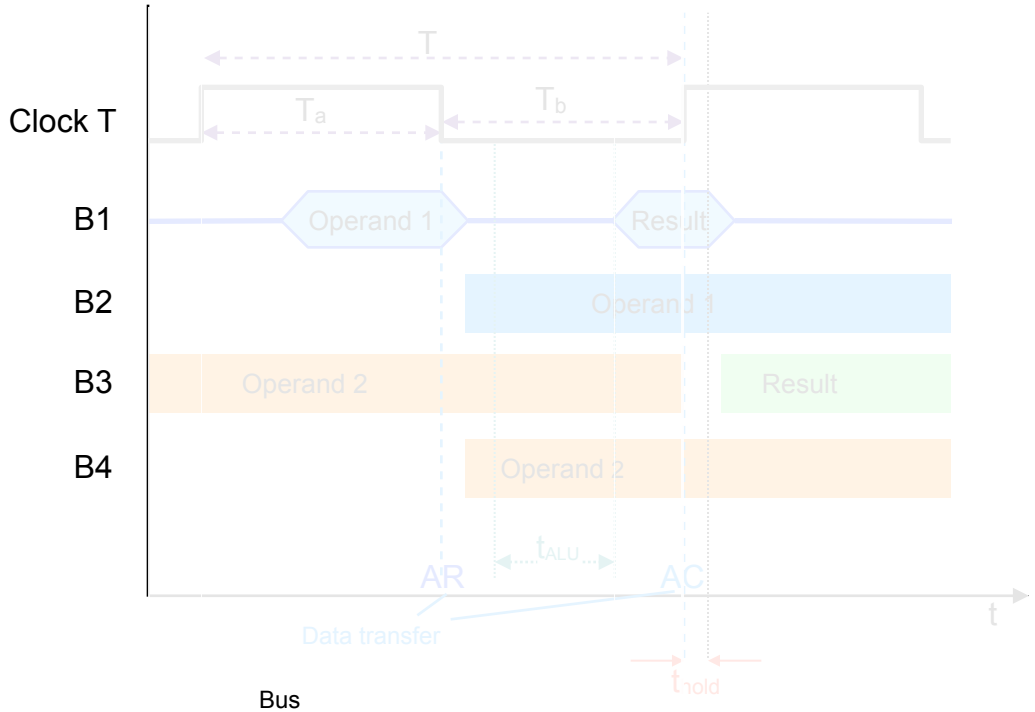
Timing Behavior of Arithmetic Logic Unit and Accumulator (1)

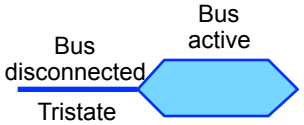


AC Accumulator register

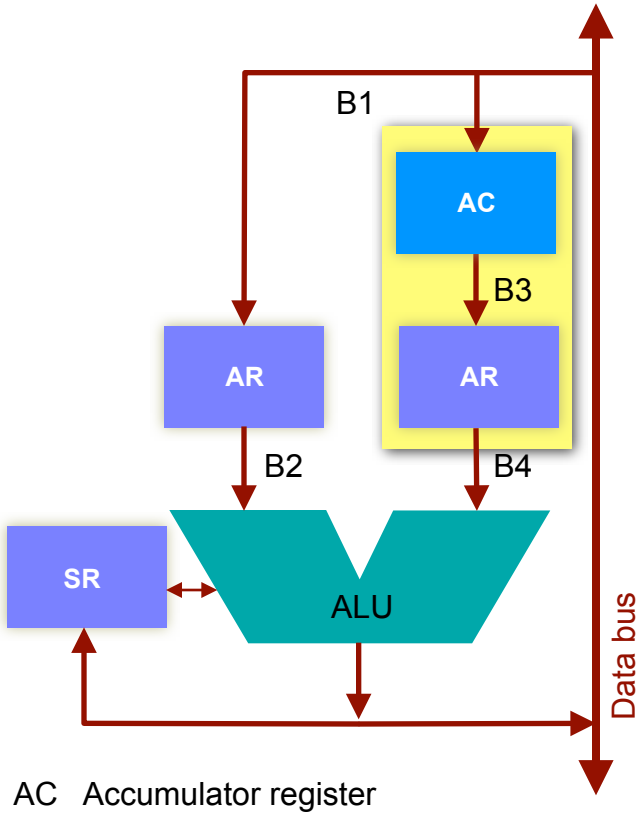
AR Auxiliary register

SR Status Register





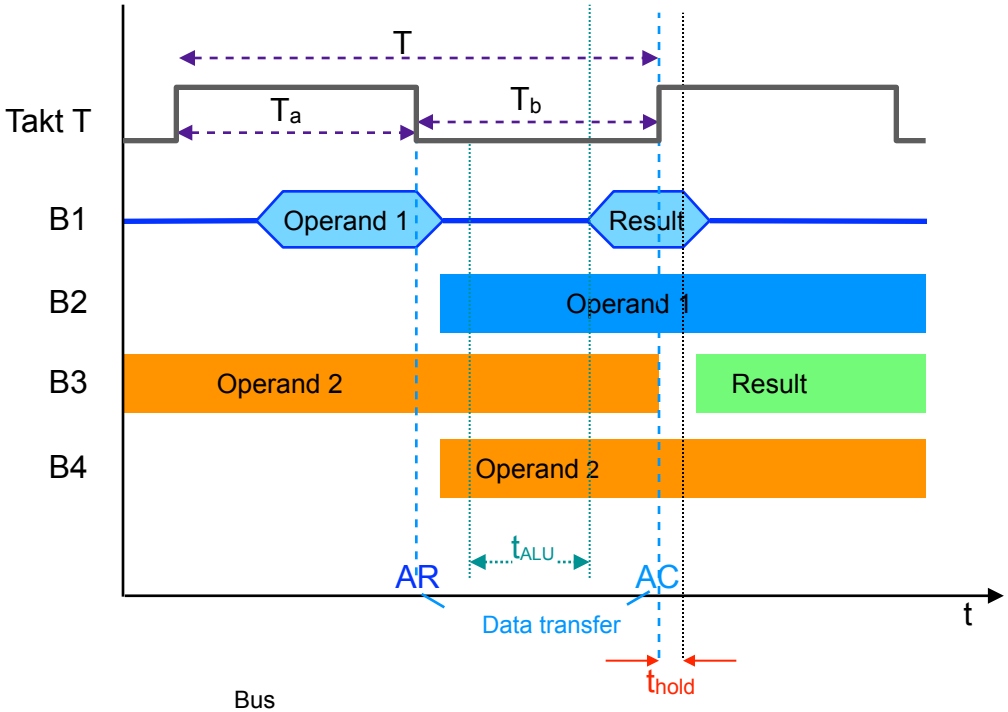
Timing Behavior of Arithmetic Logic Unit and Accumulator (2)

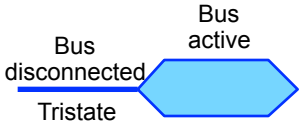


AC Accumulator register

AR Auxiliary register

SR Status Register





Ohne Animation - für Ausdruck

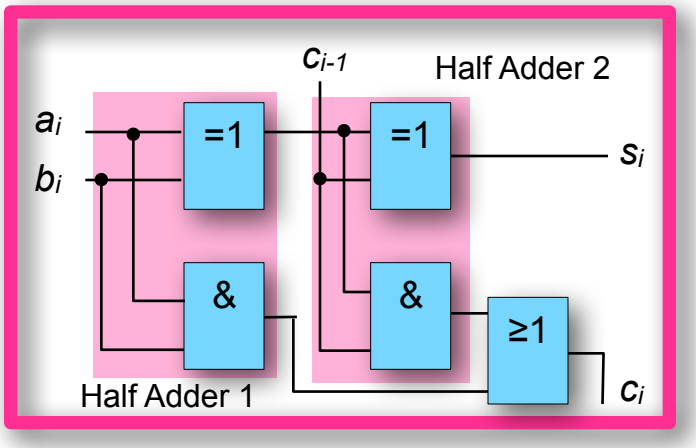
Timing Behavior of Arithmetic Logic Unit and Accumulator (2)

- During the high-level of clock T the two operands are available on the busses B1 and B3.
- They will be transferred into the auxiliary registers AR with the falling edge (high → low) of the clock signal.
- They are available statically for processing - over B2 and B4 - during low-level of clock at the inputs of the ALU.
- After a certain execution time t_{ALU} (due to the gate propagation delay in the ALU) the ALU supplies the result of the operation.
- The result will be taken over via B1 into the real accumulator (AC) with the following rising edge (low → high) of the clock signal.

Timing Behavior of Arithmetic Logic Unit and Accumulator (3)

- This prevents signal races (critical races) - which arise in the ALU possibly (pure combinatorial circuit!) - from falsifying the result, because these signals may possibly be present again at an operand input.
- From the same reason mostly a time t_{hold} is prescribed, wherein the input signals still must be valid after the data-transferring low-high edge of the clock signal.

Ripple-Carry-Adder for n-digit Operands



$$s_i = (a_i \neq b_i) \neq c_{i-1}$$

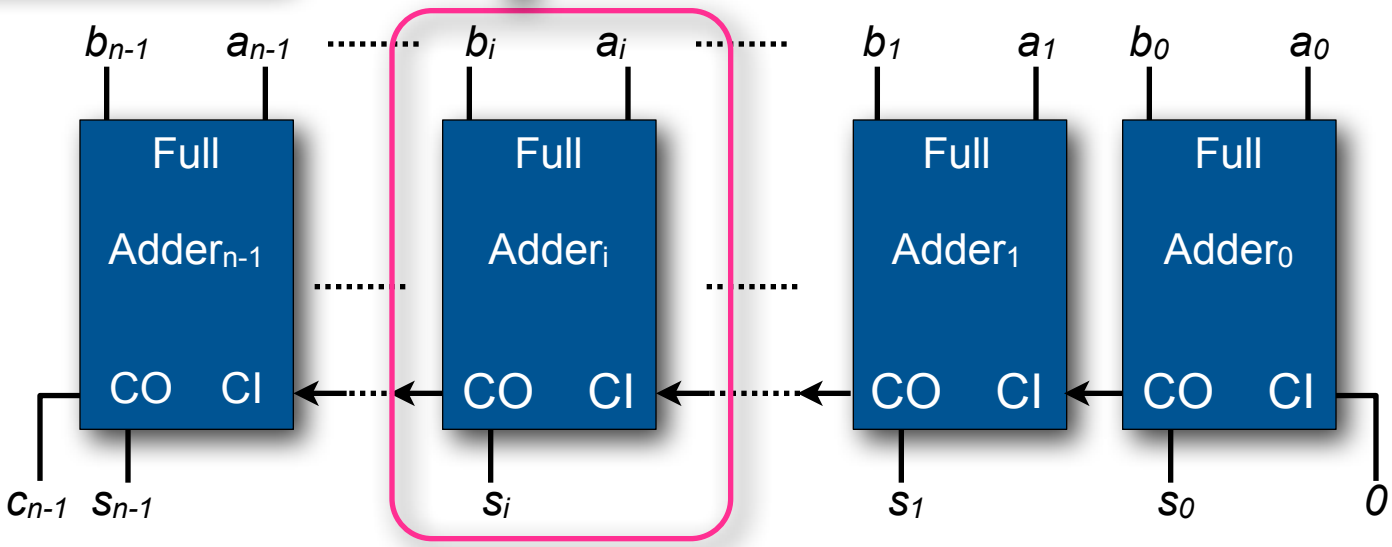
$$c_i = (a_i \neq b_i) \cdot c_{i-1} + a_i \cdot b_i$$

OR AND

≠, ⊗ … XOR, "anticoincidence"

c_i … carry

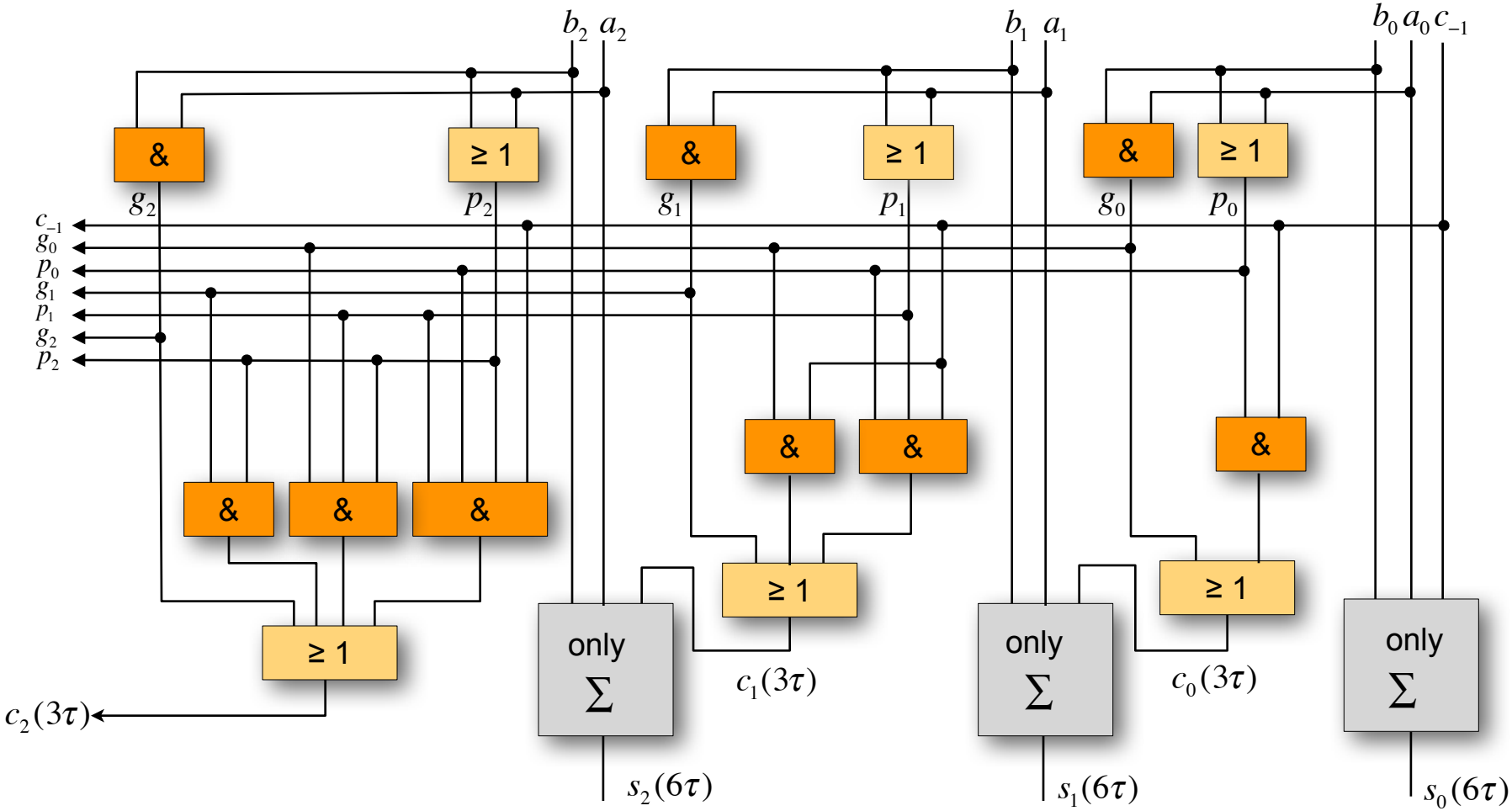
s_i … sum



Ripple-Carry-Adder (RCA): Principle

- An RCA is the simplest parallel working adder for two n bit wide operands A and B with the bit components $a_{n-1} \ldots a_0$ and $b_{n-1} \ldots b_0$.
- It ist characterized by the fact, that – after applying both operands A and B to the RCA – the sum is valid only then, after the carry bit (propagated through every adder) has arrived to the most significant bit.
- The completed sum is available only after the time $t = (2n+1) \cdot \tau$ (τ … gate propagation delay).

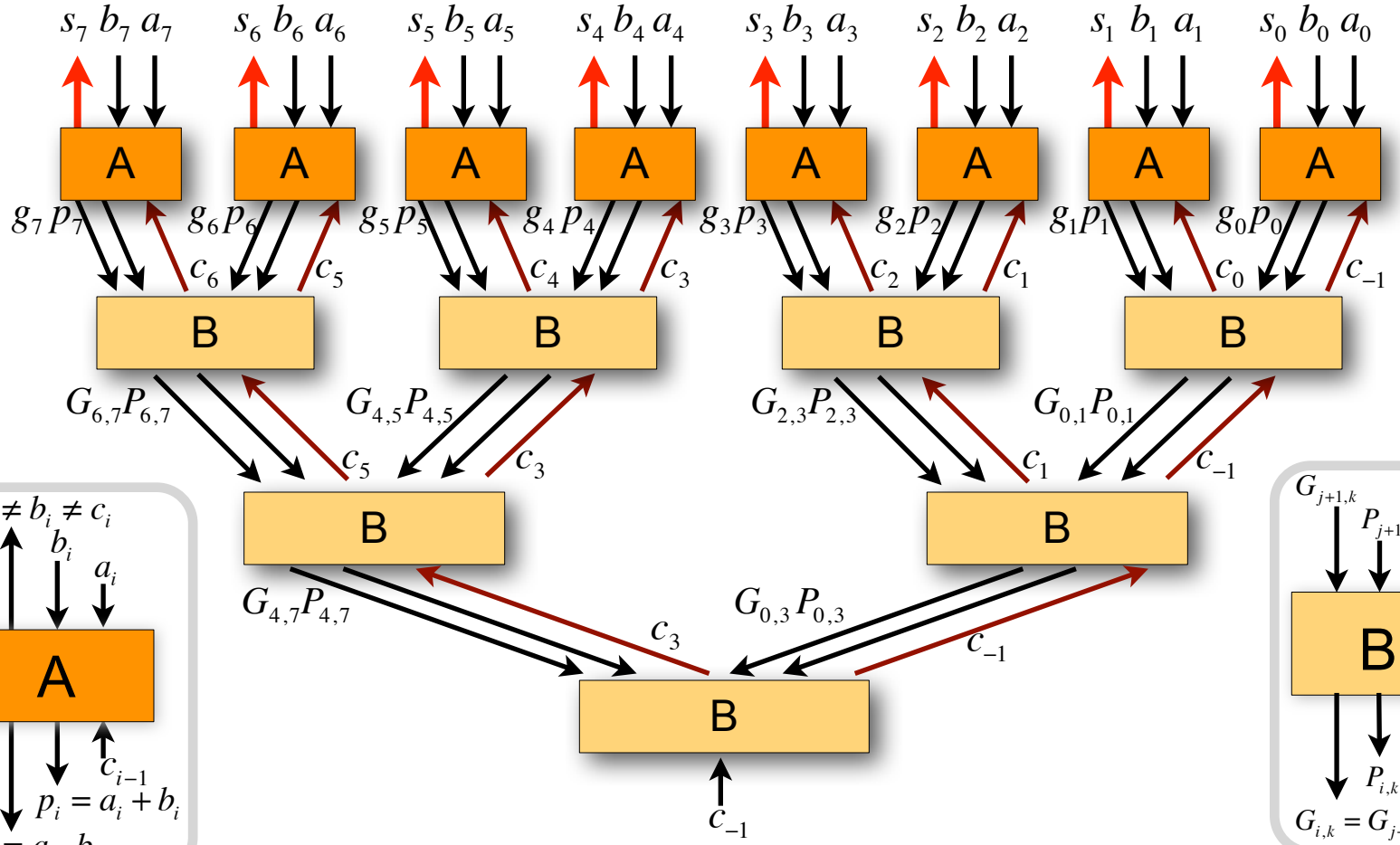
Three-Digit Extensible Carry Look Ahead-Adder

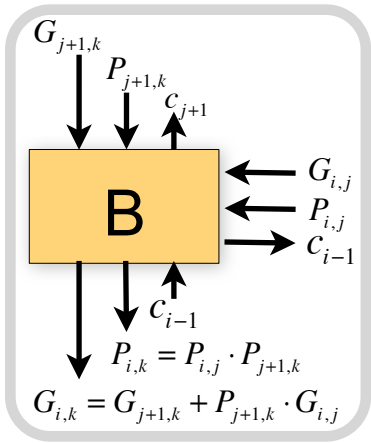


Carry Look Ahead- (CLA-) Adder: Principle

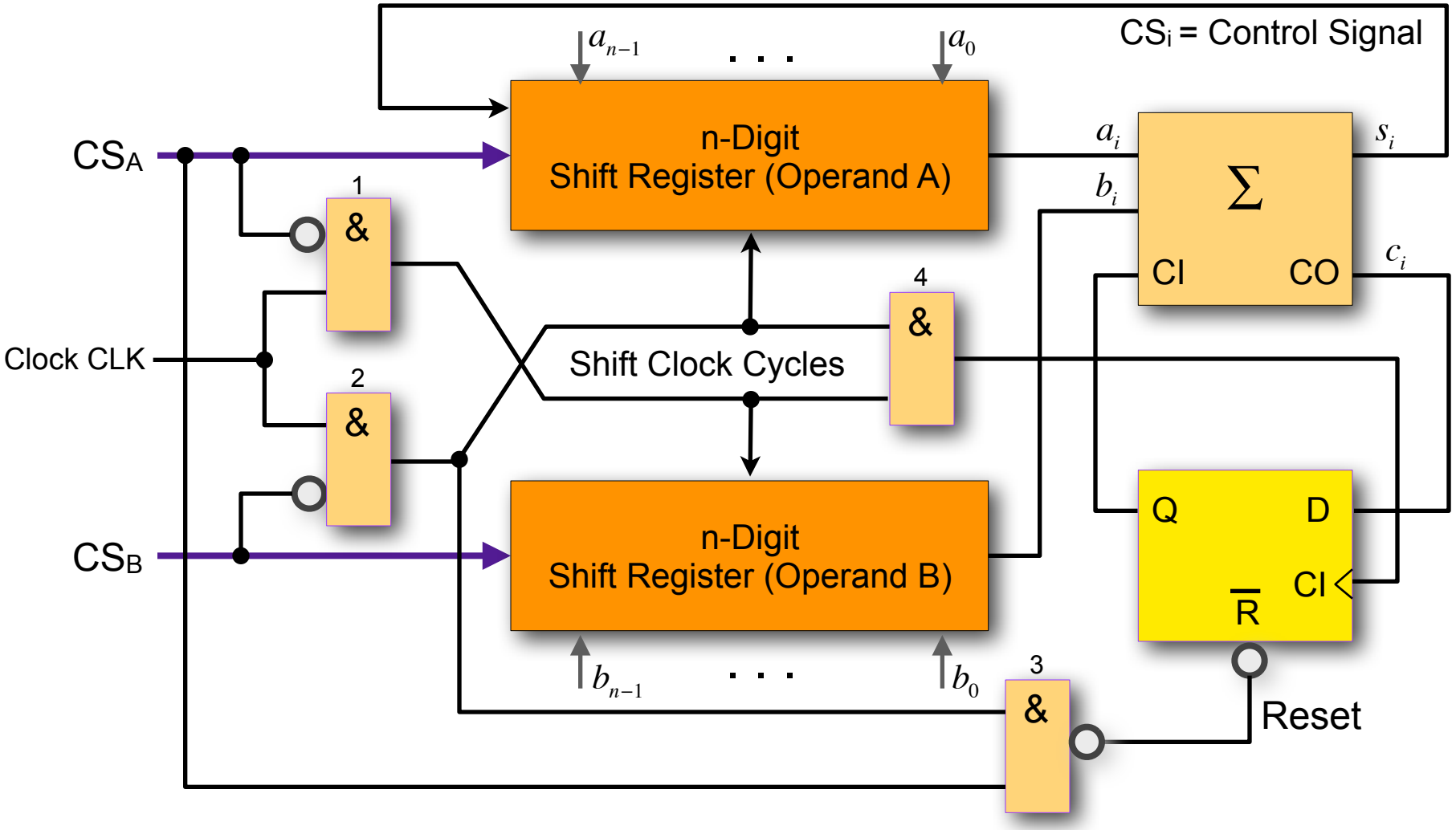
- A CLA-Adder is characterized by the „forward calculation“ (Carry Look Ahead) of carry-bit c_i in a minimum of time – thereby almost independent from n.
- This is also applied to the sum bits s_i.
- The Carry bit c_i can be provided separately – but unitary for each bit position – after a time consumption of $t = 3\tau$.
- The complexity of a CLA-Adder can be considerable larger then that of an RCA designed for same bit formats.
- Computing of sum bits is carried out unitary, too.
- Gate propagation delay in each bit position is $t = 6\tau$.
- The implementation complexity in the higher bit positions will get unacceptable large for higher n → the trade-off between complexity for realizing and achievable computing speed: cascaded CLA-Adder.

Cascaded 8-Bit-CLA-Adder





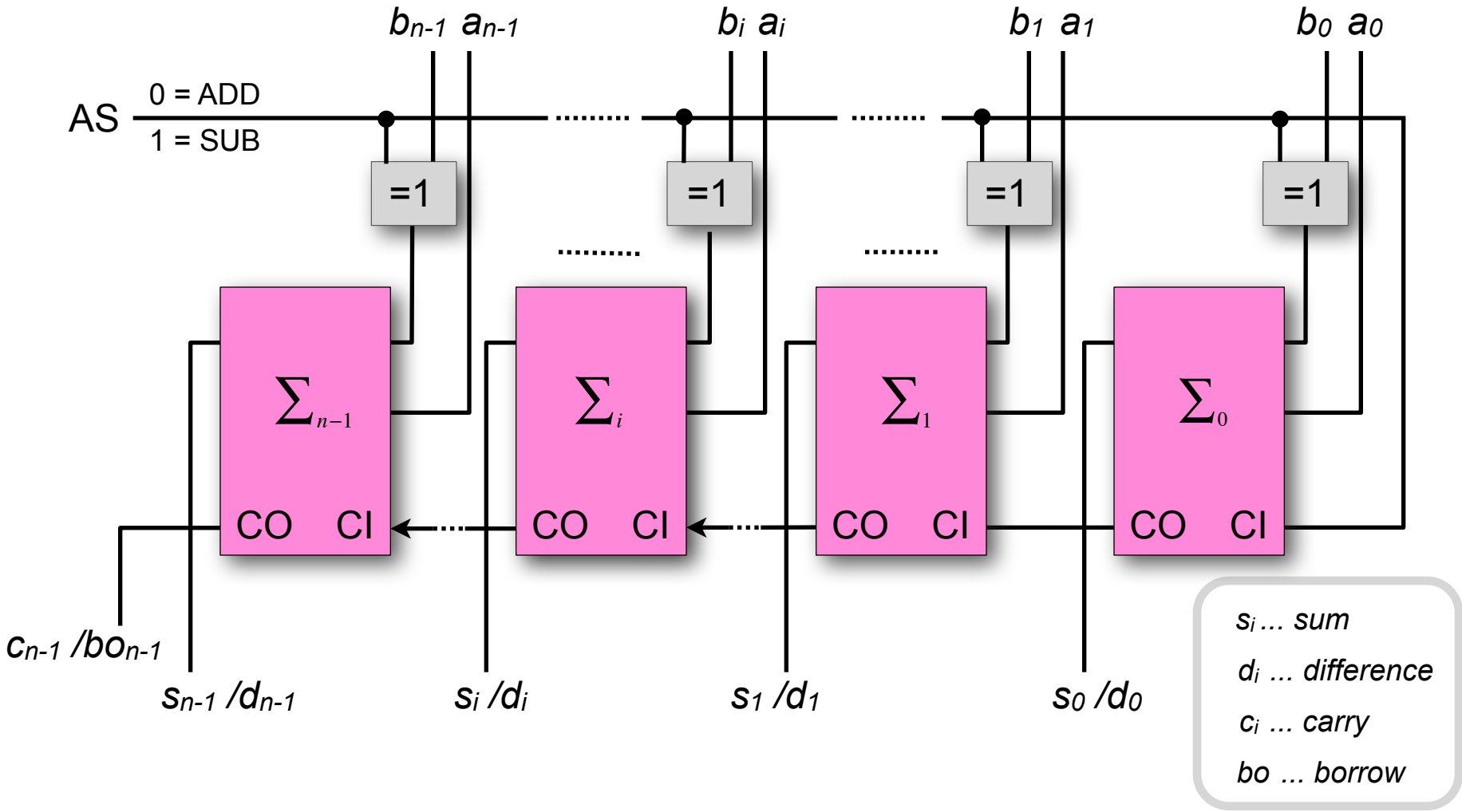
Sequential Logic System as 1-Bit-Adder (1)



Sequential Logic System as 1-Bit-Adder (2)

- Serial adders are clocked sequential logic systems, which by each clock period process exactly one bit of the participated n-bit operands.
- Execution begins with the lowest significant bit.
- The adder needs two preloadable shift registers of n digits, one full-adder and one 1-bit buffer, which undertakes the carry function.
- By serial addition of the operands A and B the sum and the carry bit are available after n+2 clock cycles in the shift register A, as well as in the cache memory.
- The controlling complexity using a serial adder is larger as if using a parallel addition.
- On the other hand – the required circuit effort is with larger n more less then with parallel adder.

Controllable Adder/Subtractor (1)



Controllable Adder/Subtractor (1)

- A parallel adder can execute also a subtraction with the same operands A and B, when with D = A−B the subtrahend B will be interpreted as negative dual number and then will be executed an addition of the kind A+(−B) again.
- Technical realization is simple just by inserting a control information AS in the adder circuit.
- 0/1 configuration of AS decides about the operation to be executed.

Method of Dual Multiplication (1)

- Dual multiplication is also based on similar computing algorithms as the dual addition → multiplication of bit positions can be realized using AND-operations.
- Additional hardware effort is necessary for row-shifting and following required column additions.
- Product P is – using n-bit „multiplicand“ B and „multiplier“ A – always 2n bit wide
- Scheme of a technical realizable process → next foil.

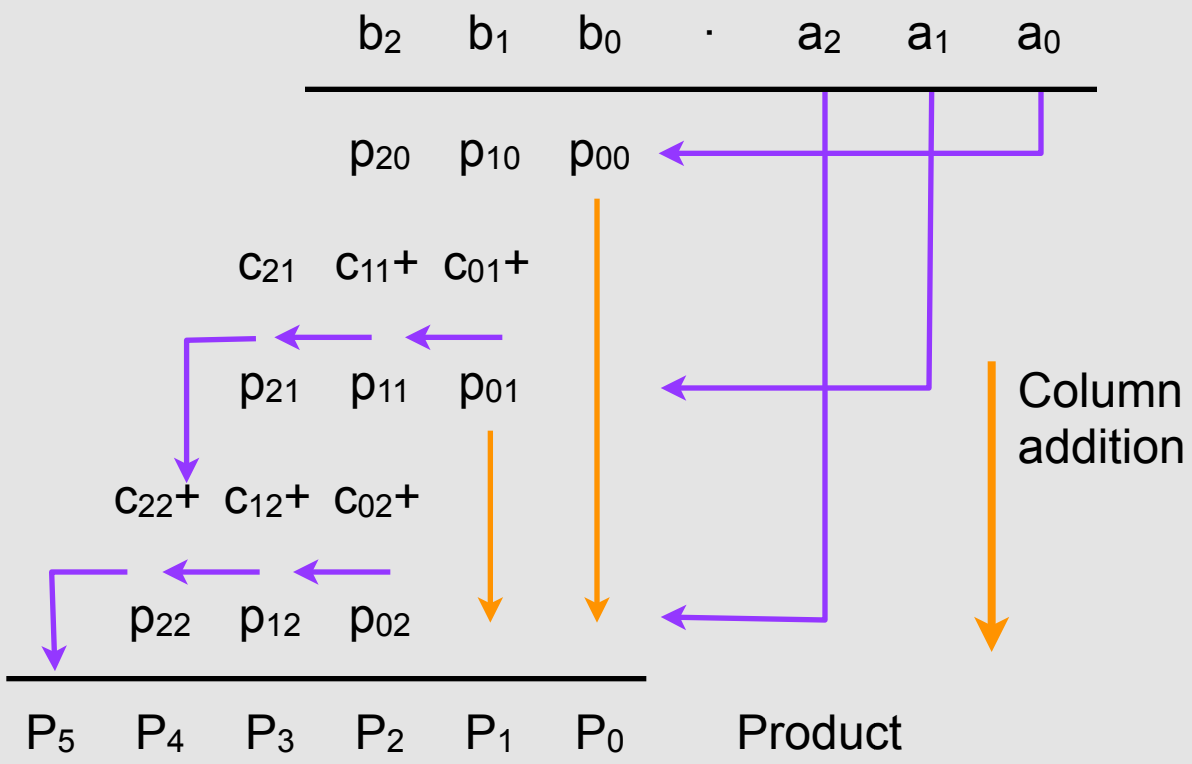
Method of Dual Multiplication (2)

Multiplication rule

b_i · a_j	$p_{i,j}$
0 0	0
0 1	0
1 0	0
1 1	1

Multiplication of three-digit operands with:

$$b_i \cdot a_j = p_{ij}; \quad 0 \le i, j \le 2$$



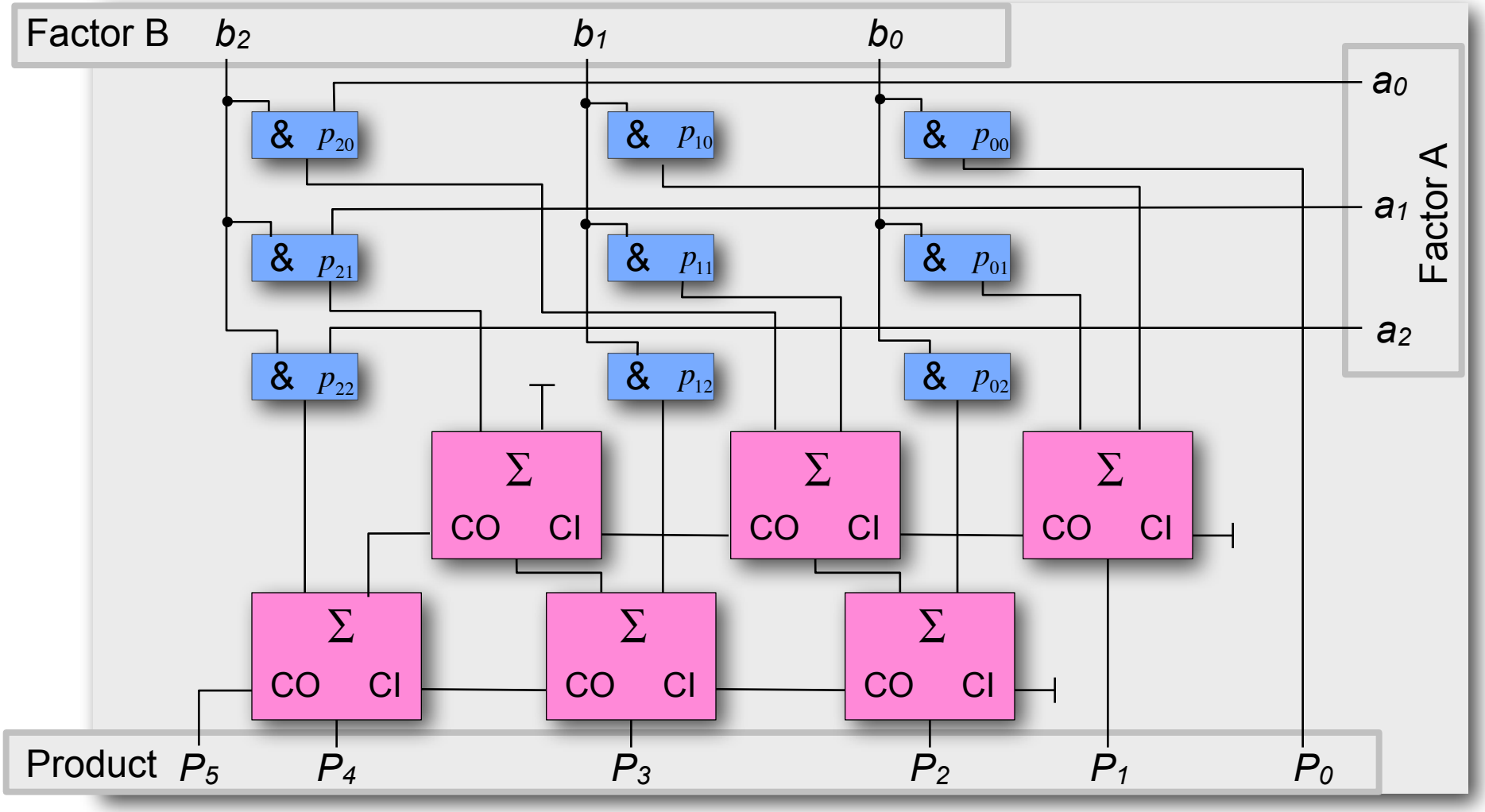
Features

- A practical multiplier requires AND gates and full adders, the term „parallel“ means that the logic combination of all participated bit positions starts „simultaneously“.
- The shown multiplier works relatively fast.
- There are no signals required to control the internal process.
- Larger width of operands leads to higher hardware complexity.
- [Lit.]: 16×16 bit multiplication → ca. 4200 gates,

  32×32 bit multiplication → ca. 253000 gates
  (each one 6 transistors = 1,5 millions!)

  → disproportionally large area requirements on processor chip.
- Example (block circuit) → next foil.

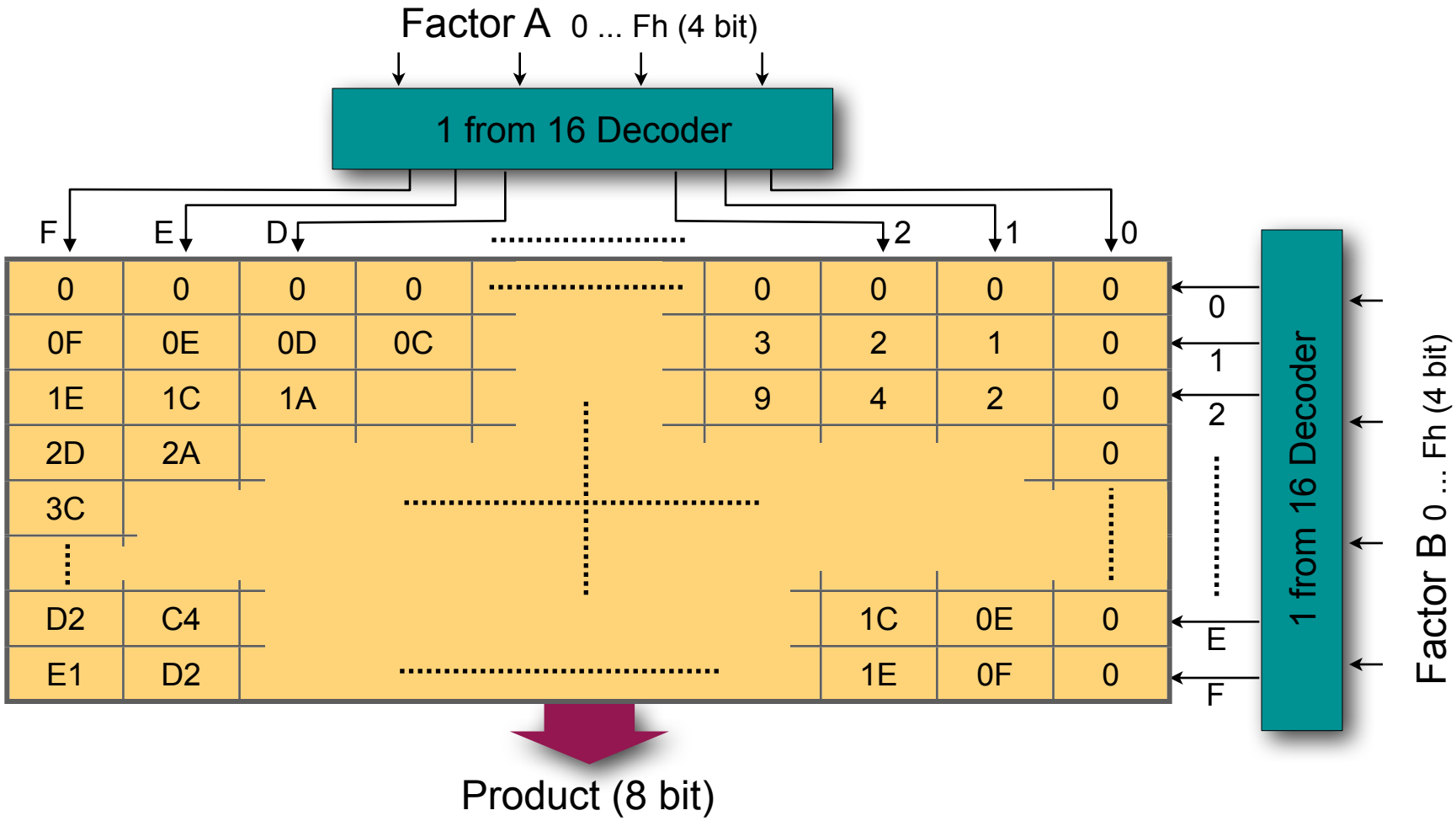
Parallel 3×3 Bit Multiplier



Features (1)

- Trade-off between chip-area/gate requirement and acceptable loss of computing speed.
- Contains a ROM table, whose rows and columns are directly driven by the operands via a decoder.
- The memory cell, arranged in the point of intersection of row and column, contains the product from the current pending operands.

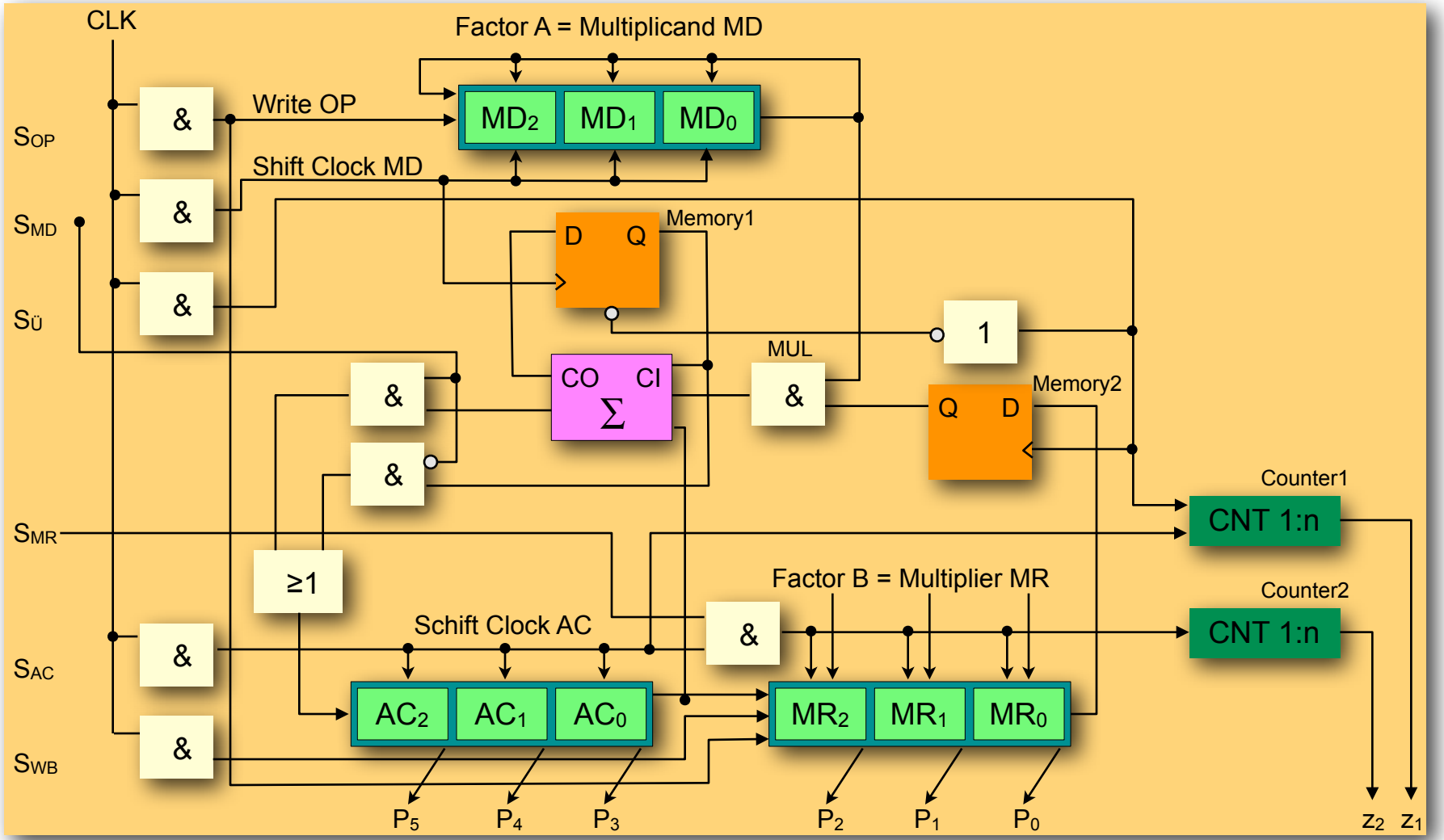
Table Control for a Matrix Multiplier



Features (2)

- The contents of the matrix are products of tetrads (4 bit wide = hexadecimal numbers: 0 … Fh), these occur symmetrically to the main diagonal → even a factor is 0 → then also the product is 0. → This can be used with hardware implementation for reducing HW.
- Such a multiplier is expandible to two byte-operands, indeed it has to cover now already 32.000 products each of 16 bit.
- Otherwise with special multiplication methods there is only required a constant number of clock cycles for generating the product.
- It operates concurrently with parts of operands on the product matrix.
- The required number of clock cycles is independent from the operand structure.
- Therefore it is applicable f.e. for pipelines in RISC processors.

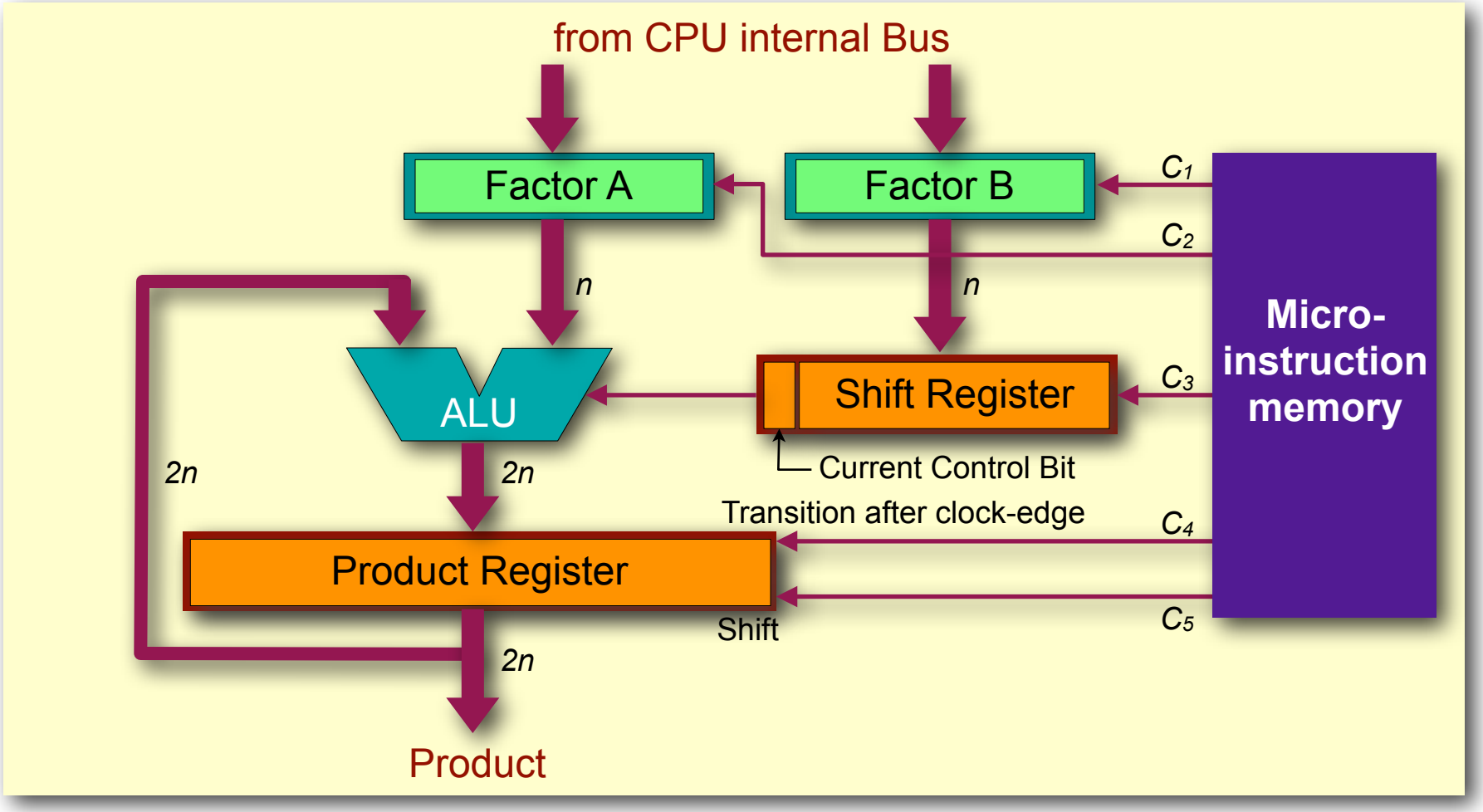
Serial 3x3 Bit Multiplier



Features

- They process like serial adders each bit of operands separately.
- Advantage: smaller hardware complexity → important when using larger operand formats.
- Disadvantage: more complicated clock control required.

Serial-Parallel Addition Multiplier



Features of a Parallel Addition Multiplier

- CISC instructions are complex → often multiple operations in one instruction → circuit combines multiplication and addition.
- The controlling of the internal operating sequences – required for multiplication – is provided by a micro instruction memory, that supplies the control signals C_i.
- The multiplier requires n cycles of different length for executing the multiplication.
- That is, duration of the multiplication is dependent as well as from structure of operands and from n
- Multiplication time increases with the number of „1s“ in the multiplier.
- Because of structural dependencies of the multiplication time this arithmetic logic unit is only suitable for CISC architectures.

Principle

- Division is with algebraic view the inverse of multiplication.
- Generating a product of two n-bit operands → a 2n bit wide result arises.
- In contrast, with parallel division the dividend must have 2n and the divisor n digits.
- There arise two results: Dividend : Divisor = Quotient and Remainder.
- Thereby the rule is always, that the remainder must be greater than or equal to zero, but less than the quotient.

General (1)

- Specific ALUs, developed for die acceleration of multimedia- and communication applications.
- This has been reached with new instruction and data types, which allow a higher performance by utilization of inherent parallelism of various multimedia- and communication algorithms.
- Full compatibility with existing operating systems.
- Most known: MMX technology (Multi Media Extension) from Intel.
- MMX technology has been designed as a set of common integer instructions, that has been customized for the requirements of these wide application field.

Features (1)

- Particular features of these applications:
  - often „short“ data, f.e. two 8 bit (8 bit pixel) or 16 bit operators (audio samples) have to be combined, examples: music synthesis („synthesizer“), voice compression and −recognition, image processing, MPEG video …
  - small, often rerunning loops
  - frequently multiplying and adding up
  - CPU-intensive algorithms
  - highly parallel operations
- Characteristic features of MMX technology:
  - category SIMD
  - 57 new instructions
  - eight 64 bit wide MMX register
  - 4 new data types

Features (2)

- The MMX unit consists of two pipelines (MMX-U and -V) like the normal processing unit, which allow for MMX-instructions once again a separate superscalar processing.
- Both pipelines have each an ALU for addition, subtraction and for logic operations.
- These ALUs are designed with respect to the arithmetic to use instructions of image processing, f. e. with regard to the overflow behavior.
- Therefore exist a special multiplication unit, a combinatorial shifting unit (shifter) and a register set with eight MMU registers, where each is 64 bit wide.

Features (3)

- The multiplication itself is organized as pipeline operation again → there can be executed overlapped several successive loaded multiplication instructions.
- So one multiplication per clock cycle can be executed on average.
- Also a combined addition and multiplication is possible (in total of only three clock periods), especially for audio and video algorithms.
- The MMX unit doesn‘t know an overflow (or underflow) and no special overflow flag bit. If there exists an overflow, the following will be done:
  - Either the overflow bit will be ignored and, a reset of register contents will be performed (wrap-around),
  - or the register contents sticks at highest value (saturation), namely depending from actual instruction.
- If a register value "underflows" then it sticks at the smallest possible value.

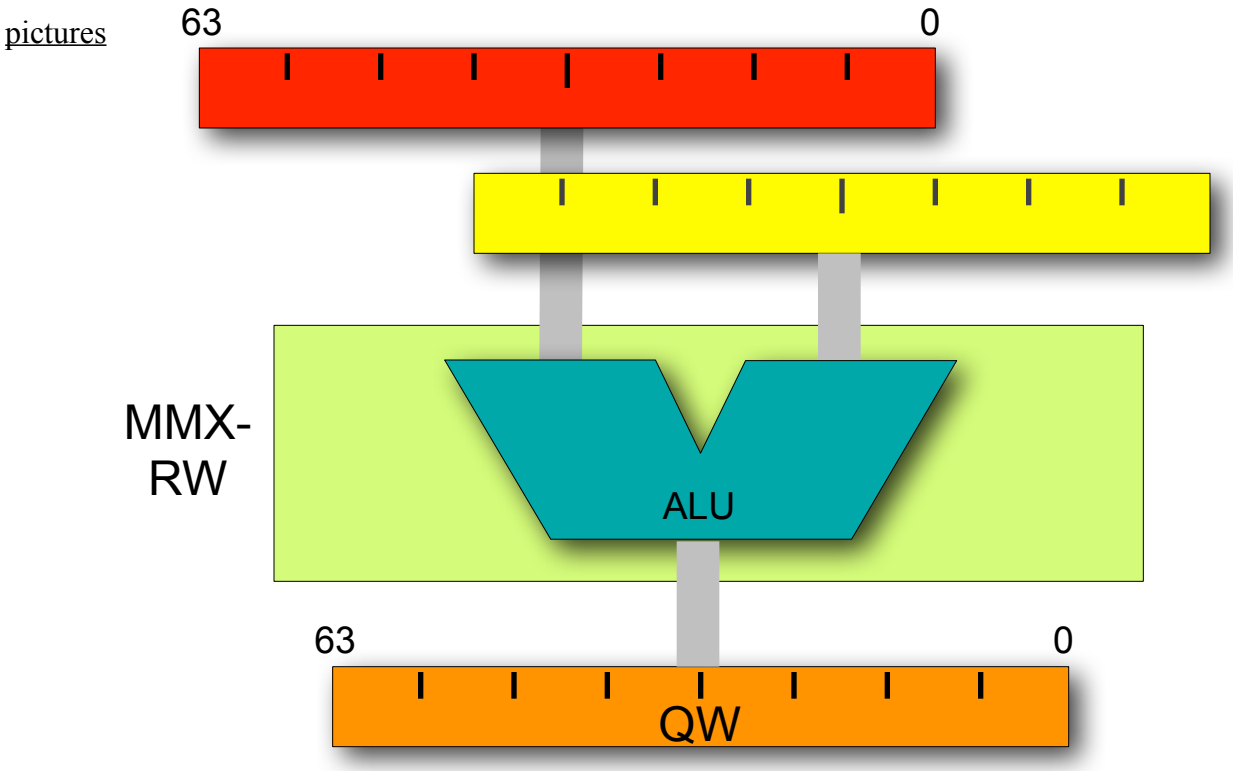
Register

- A variable segmentation of register length exists in the 64 bit registers with mapping by optionally multiple data types:
  - 8 packed bytes,
  - 4 packed words (2 times 16 Bit),
  - 2 packed 32-bit double words,
  - 1 quad word of 64 bit.
- A single instruction therefore can be affected parallel to 8, 4 or 2 data units.
- The 8 MMX registers
  - can be addressed directly with the register names MM0-MM7.
  - are handled with task switching as additional floating point register.

57 new Instructions

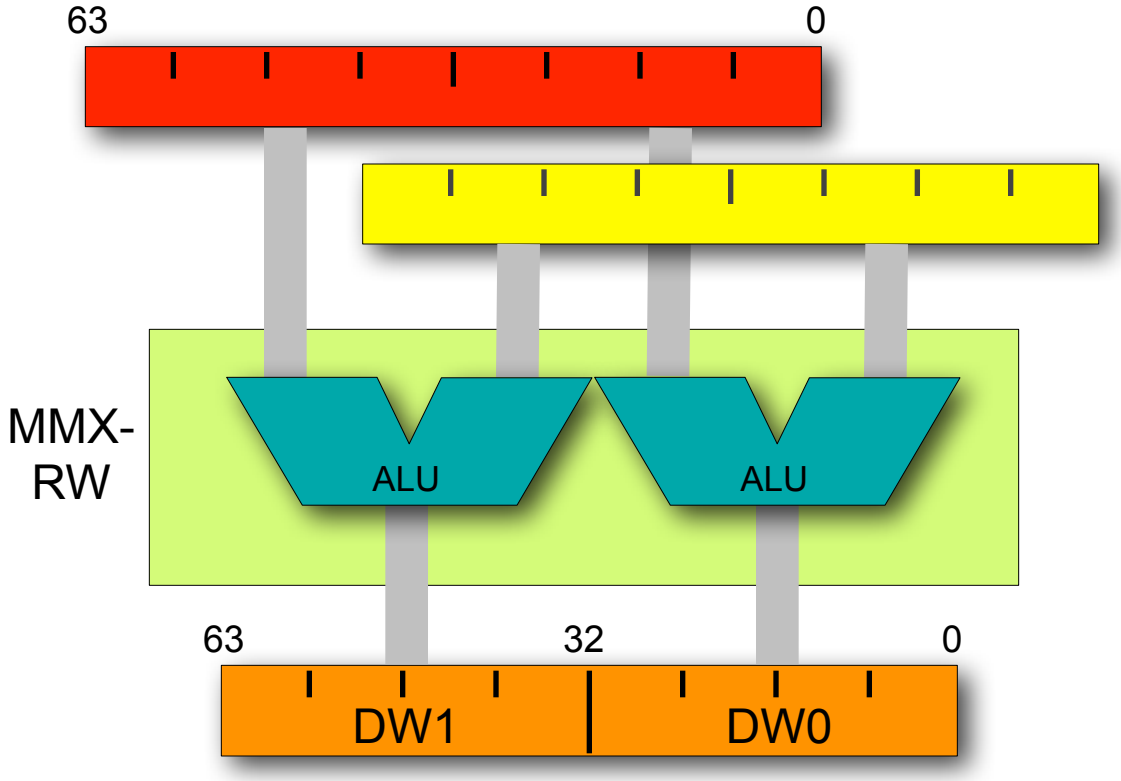
- Data transfer instructions → with 32 bit or 64 bit width, f.e. for exchanging a data between a normal and a MMX register.
- Arithmetic instructions → packed data can be added and subtracted, optionally with wraparound or saturation, multiplication oder combined multiplication/addition.
- Shift instructions → a 64 bit pattern can be shifted single or multiple L/R.
- Compare instructions → 2 packed 64 bit registers can be compared with each other.
- Logic instructions → 2 64 bit registers can be linked together bitwise AND, OR, EXOR.
- Convert instruction → transformation of different packed data types.
- Exit statement for the MMX state.

Examples of Operations (1)



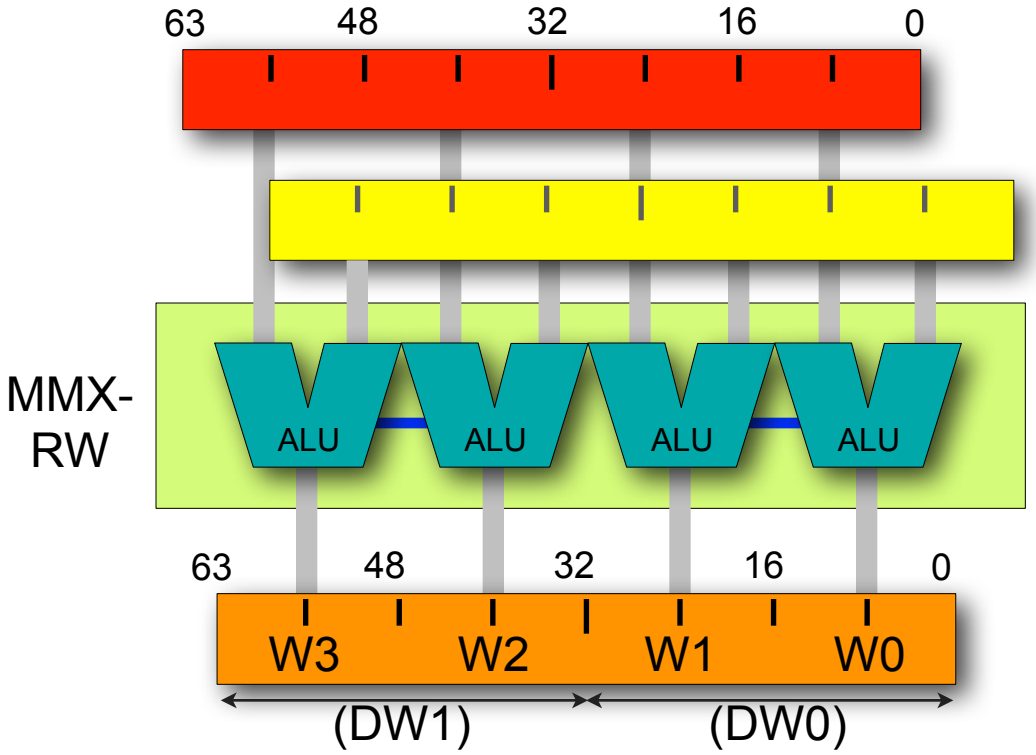
Two 64-bit operands *(also called Quadwords, QW)* are switched together to a 64 bit operand.
Beside the load/store operations logic bitwise combinations are possible as well as shift operations.

Examples of Operations (2)



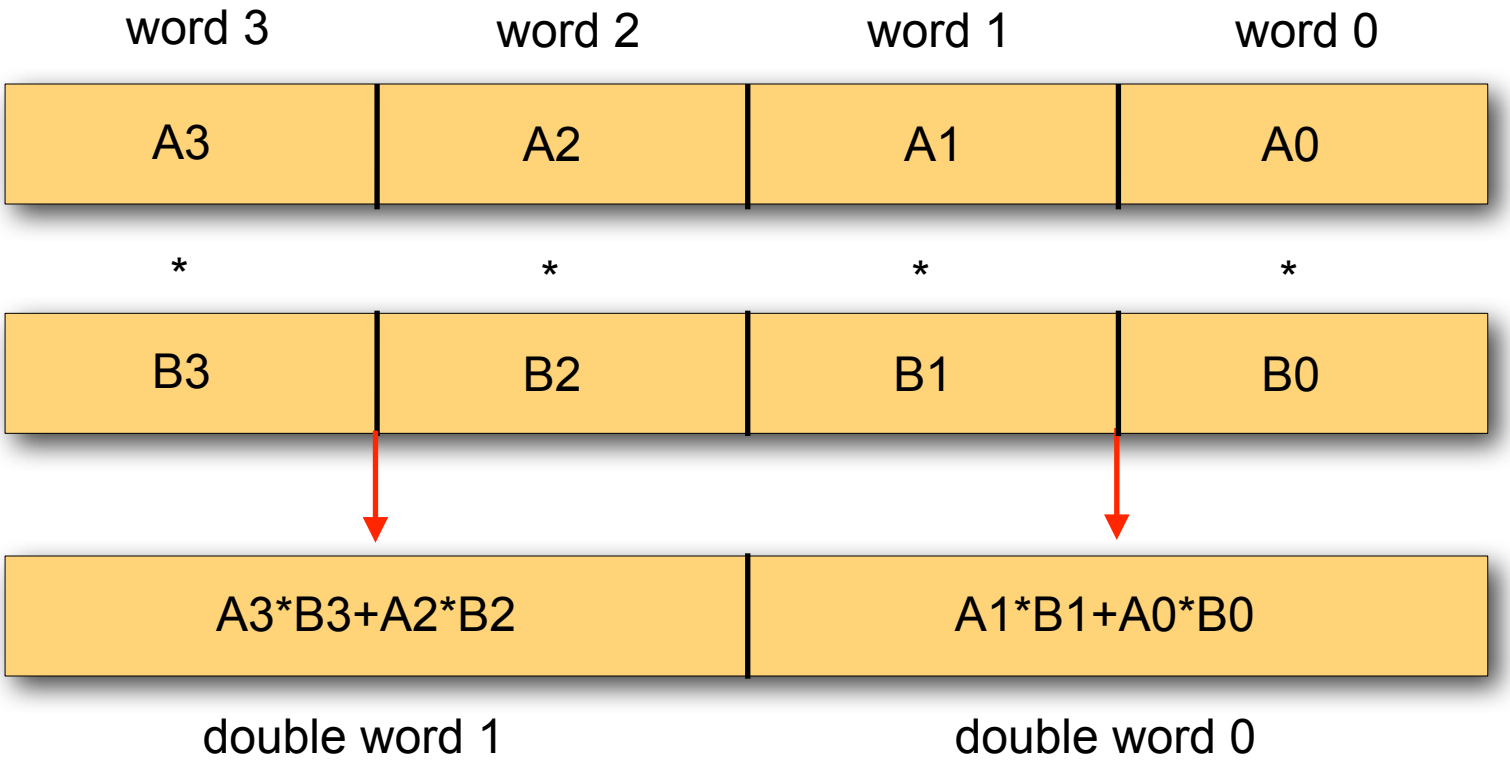
Two pairs of each 32 bit double words *(Double Words, DW)* are combined independently from each other to two 32 bit results. There exists no carry-bit between the two sub-calculations.
Additional to the shift- and transfer operations especially come along the addition and the subtraction as well as the compare operations "*equal*" and "*greater*".

Examples of Operations (3)



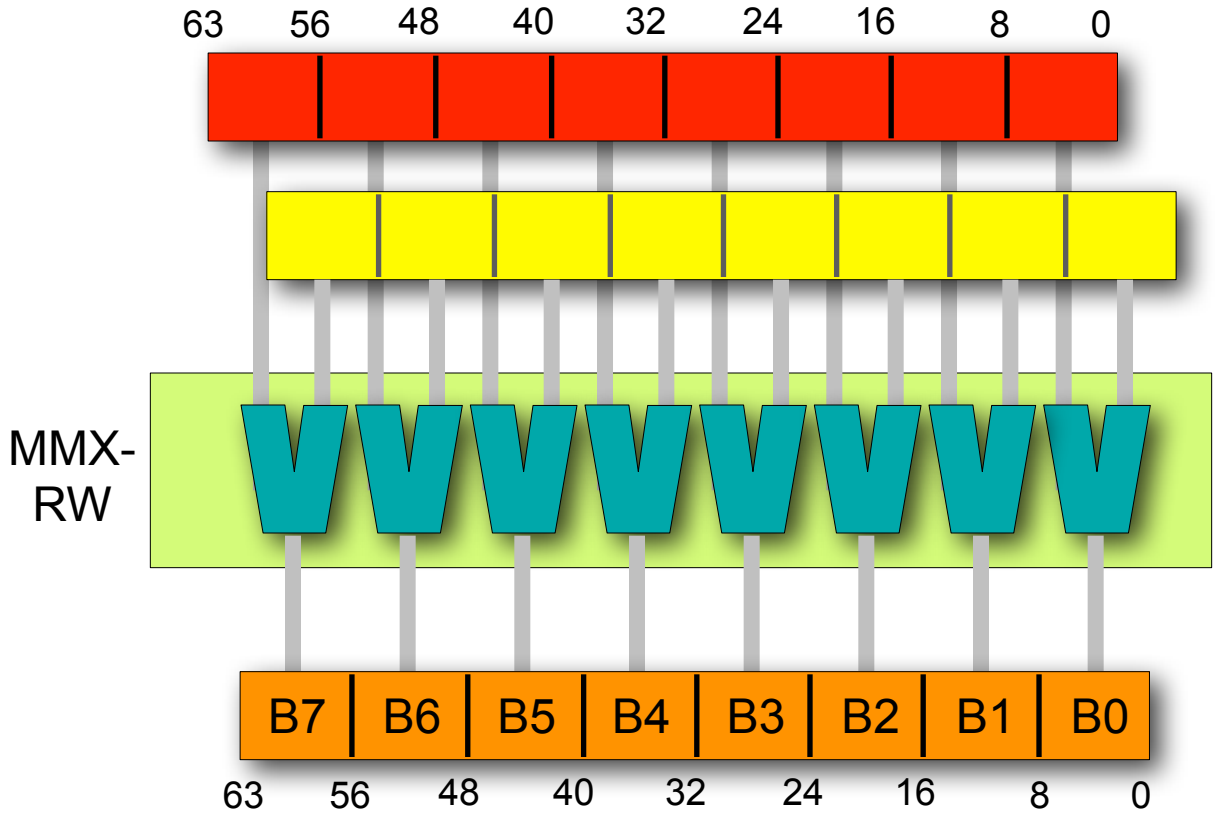
Operation acts on four pairs of 16-bit words *(Words, W).*
Additionally, a multiplication (16 x 16→16 bit) can be executed, whereby the higher or lower 16 bit can be taken from the 32-bit result as the final result, optionally.
The blue marked lines between each two sub-ALUs indicate a complex operation, whereby the concerned sub-ALUs execute each a 16x16 bit multiplication and add their 32-bit products to a 32-bit total result.

Examples of Operations (4)



These operating mode affects on four pairs of 16-bit words:
The participated sub-ALUs execute each a 16x16 bit multiplication and their 32-bit products are added to a 32-bit total result.

Examples of Operations (5)



Eight byte pairs (*Byte, B*) are used as operands and a result from also 8 bytes has been calculated.
Application: addition, subtraction and compare operations.