Professur Schaltkreis- und Systementwurf

Components and Architectures

Exercise

TECHNISCHE UNIVERSITÄT CHEMNITZ

0. Organization

Organizational

Exercise:

- Gaining knowledge on the theoretical processor DLX
- Development of pipeline-concepts = ”Architectures”
- Arithmetic and logic calculation units = “Components”
- **Ask questions!**

Written exam (IS/IC: 120 minutes, Es: 90 minutes)

On lecture, exercise and practice!

No auxiliaries (pocket calculators, books, writings) allowed in the exam!

Organizational

Practical lessons:

- Winter semester: Simulate DLX pipeline (4 lessons)
  - Preparation needed for every lesson
  - Short introduction question for every lab (question list available in OPAL)
  - Oral examination at the end of lab 4
- Summer semester (IC, IS only):
  - ALU as VHDL component (for VHDL beginners) **OR**
  - Integration and extension of an 8051-Softcore into a Spartan3-FPGA
  - Oral examination on VHDL design

Organizational

Requirements for exam admission:

- passing all 4 lessons in winter semester (all master courses)
- submission of design in summer semester (IS, IC only)
  (students from SS14 may submit the solution up to January 4th,
  but no assistance by advisor possible in winter semester)
- successful students from former semesters keep their admission, but exam questions will rely on last lab cycle
  (but there are no major changes to 2013 lab cycle)

Literature

John L. Hennessy, David A. Patterson:
“Computer Architecture - A quantitative approach.”
Academic Press, 2006; ISBN 978-0123704900
(German version available)
also suitable: “Computer Organization and Design”

William Stallings: “Computer organization and architecture; designing for performance.” Pearson Education, 2013, ISBN 978-0273769194

Literature

Summer semester only:
Peter J. Ashenden:
“The Designer’s Guide to VHDL”, 2nd Edition;
Morgan Kaufmann Publishers, 2002; ISBN 1−55860−674−2

U. Heinkel et al.: http://www.vhdl-online.de/ after the book
“The VHDL Reference”; ISBN: 0-471-89972-0

Course material

is available on OPAL:

https://bildungsportal.sachsen.de/opal/dmz/

- Please register there (URZ login)
- If necessary change language to English (“Einstellungen”)
- Subscribe to the C&A exercise:
  Lehr- und Lernangebote - Fakultät für Elektrotechnik/
  Informationstechnik - Components and Architectures
- or follow link on course homepage

Subscription to practice

is made in OPAL:

https://bildungsportal.sachsen.de/opal/dmz/

- Is possible after subscription to exercise only!
- Several groups available - please check the times of the lessons!
- Check the exact starting time of the labs!
- Download the according manual and solve the preparation tasks!

Contact to advisor

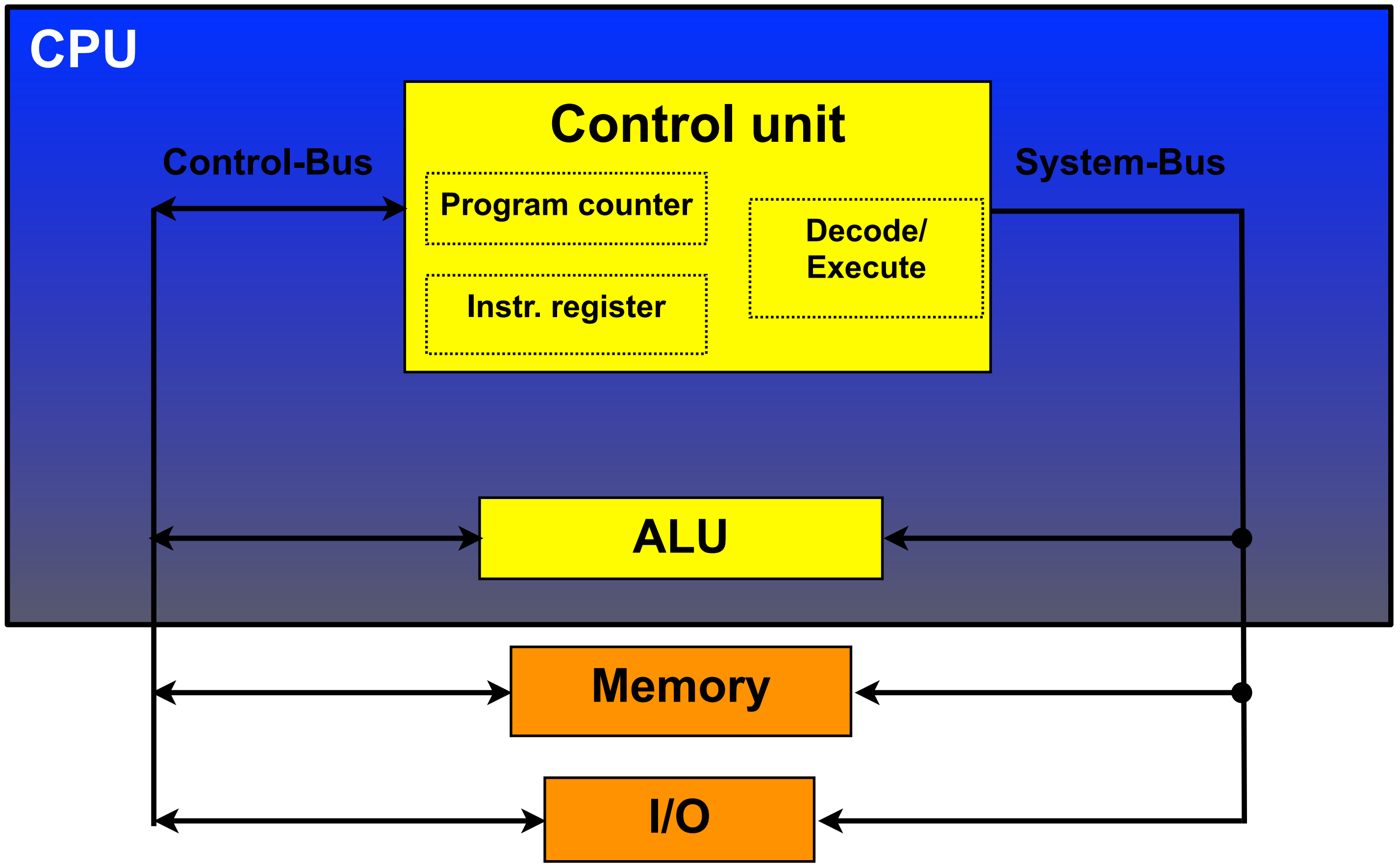
Consultation hour:
every Thursday 10:45-11:30 in 2/W430

Other dates:
E-Mail erik.markert@etit.tu-chemnitz.de
or use E-Mail function in OPAL

All communication from advisors will be done by e-mail using OPAL:
- Check the according mailbox (usually TUC account)
- Ensure that mailbox is not over quota

1a. Introduction to Processors

Von-Neumann-Machine



Processor core

- Two variants: Harvard & Princeton architecture
- Main components are architecture-independent
  - Program Counter
  - Instruction Register
  - Decoder/Control Signal Generator
  - ALU

Processor: Datapath (vN-ALU)

- Usually uses about 50% of the die
- Determines the costs of the chip
- Determines the clock cycle time
- Depends on hardware−technology
- Simple to design (regular structures)

Processor: Datapath (vN-ALU)

- Collection of execution units, e.g. arithmetic−logic units, shifter, registers and their interconnections.
- From programmer's view the processor state needed to be saved at suspend time and to be reloaded at continuation.
- The state contains the general purpose registers, program counter, interrupt−addressregister and program state register (sum of flags).

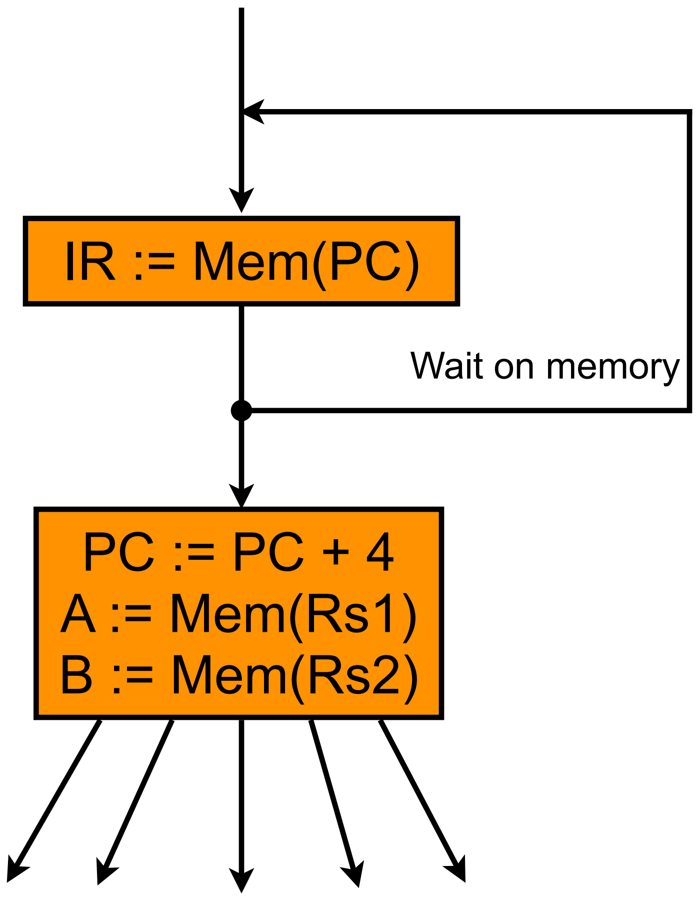
Will be discussed in detail later

Processor: Control Unit

- Hard to design
- Simplification by reduction of instruction set (RISC vs. CISC)
- Control unit generates signals for datapath for every cycle during instruction execution.
- Typically determined by a state diagram/ FSM

Main topic of the next exercises

Processor: Control Unit



Fetch instruction from memory

Increment program counter
Fetch operands from memory
Generate control signals

Techniques of control unit implemenration

Hardwired Control

Transformation of state diagram into hardware
(= via Finite State Machine)

Example DLX:

DLX state diagram with 50 states	6 Bit
control inputs (6 bit OPC, 6 bit Arithmetic-OPC)	12 Bit
condition examination for jumps	3 Bit
control outputs (estimation)	40 Bit
Memory demand at hardwiring:	**10 MByte**

Techniques of control unit implemenration

Microprogram control

Control unit is a ‘miniature computer’ with micro code instructions

Micro instructions on control level steer datapath

Micro instructions specificate all control signals of the datapath, extended by the ability of conditional decisions which micro instruction is the next to execute.

Techniques of control unit implemenration

Microprogram control

Advantages

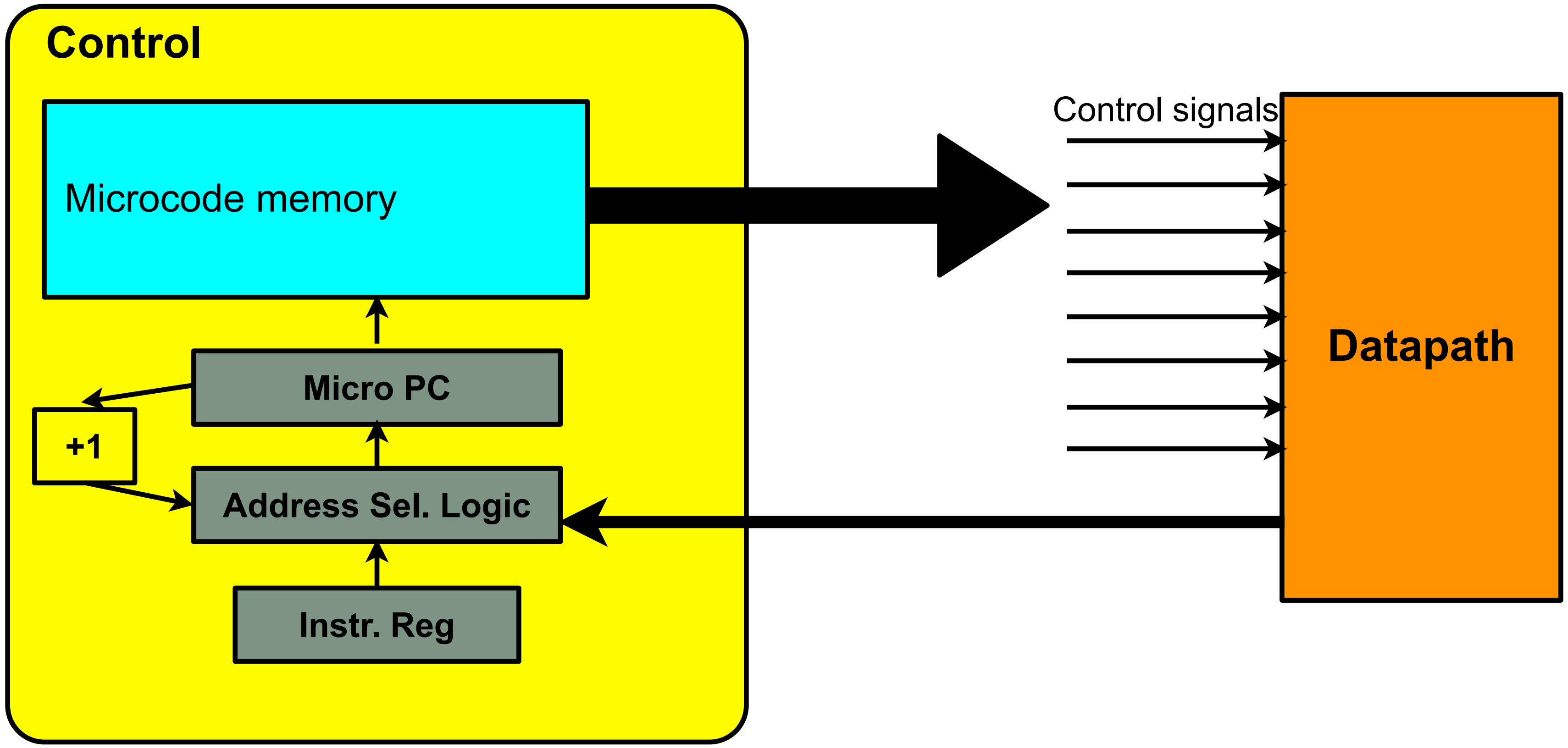
- Change of instruction set by modification of control memory without touching hardware
- Complex instruction set possible

Disadvantages

- Rising design complexity
- Extended instruction execution time

Techniques of control unit implemenration

Microprogramm control



Difficulties in control unit

Problem: Interrupts

Examples for interrupts:

- I/O−Device demand
- Call of an OS-service by user program
- Breakpoint (interrupt set by programmer)
- Arithmetic over− or −underflow
- Page fault (not in main memory)
- unaligned memory access if alignment is necessary
- Memory protection failure
- Undefined instruction
- Trouble in hardware
- Problem with power supply

Difficulties in control unit

Frequency of interrupts at a VAX 8800 (1986) with multi-user workload (12 MIPS):

- I/O−Interrupt 2,7 ms
- Timer−Interrupt 10,0 ms
- Every Hardware−Interrupt 2,1 ms
- Software−Interrupt 1,5 ms
- **Every Interrupt 0,9 ms (every 10.000th instruction!)**

Measurements for Computation power

CPI

Cycles per Instruction - Average number of cycles per instruction

$$\text{CPI} = \frac{\text{Number of required clock cycles}}{\text{Number of instructions}}$$

or

$$\text{CPI} = \sum(\text{CPI of one instruction} * \text{relative frequency in program})$$

Measurements for Computation power

Instruction	Cycles	Frequency	
Load	8	0,21	1,68
Store	7	0,12	0,84
ALU	6	0,37	2,22
Compare	7	0,06	0,42
Jump	4	0,02	0,08
Subprogram call	6	0,00	0,00
Branch taken	5	0,12	0,60
Branch not taken	4	0,11	0,44
CPI			**6,28**

Measurements for Computation power

Instruction	**Cycles**	**Count**		**Count**	
Load	8	21	168	21	168
Store	7	12	84	12	84
ALU (halved)	6	**38**	228	**19**	114
Compare	7	6	42	6	42
Jump	4	2	8	2	8
Branch	5	12	60	12	60
CPI			**6,48**		**6,61**

Measurements for Computation power

Clock cycle length

- Determined by the slowest circuit active during clock cycle (typically datapath)
- Depends on implementation
- Dominant element for pipelining

Measurements for Computation power

MIPS

“Million Instructions Per Second”

$$\text{MIPS} = \frac{\text{Number of instructions}}{\text{Execution time} * 10^6}$$

$$\text{MIPS} = \frac{\text{Clock frequency}}{\text{CPI} * 10^6}$$

Problem: depends on instruction set

(Example: floating point hardware vs. software)

Measurements for Computation power

MFLOPS

“Million FLoating point Operations Per Second”

$$\text{MFLOPS} = \frac{\text{Number of FP-instructions}}{\text{Execution time} * 10^6}$$

Problem: depends on instruction set

(Example: simple fast instructions vs. complex instructions)

Task from exam 2013

Calculate the MIPS and MFLOPS value of a CPU based on the following values. Please also give intermediate results. Calculate the CPI value if the CPU clock is 50 MHz.

Instruction	Number of Executions	Total execution time for all executions (seconds)
ADD/SUB integer	4.000.000	2
Load/Store	1.500.000	3
ADD/SUB float	900.000	5
JMP	500.000	10
MUL/DIV float	100.000	15

1b. DLX processor

DLX

MIPS-like CPU from J. L. Hennessy and D. A. Patterson
auxiliary means for teaching the principles of computer architecture

DLX − Roman numerals (560d)
560 −> average from 13 machine−numbers similar to the DLX−philosophy:

AMD 29k, DECstation 3100, HP 850, IBM 801,Intel i860, MIPS M/120A, MIPS M/1000, Motorola MC88k, RISC I, SGI 4D/60, SPARCstation−1, Sun−4/110, Sun−4/260

Many of the following slides are taken from the books "Computer organization and design" and "Computer architecture - a Quantitative Approach" written by Hennessy and Patterson

DLX

Philosophy:

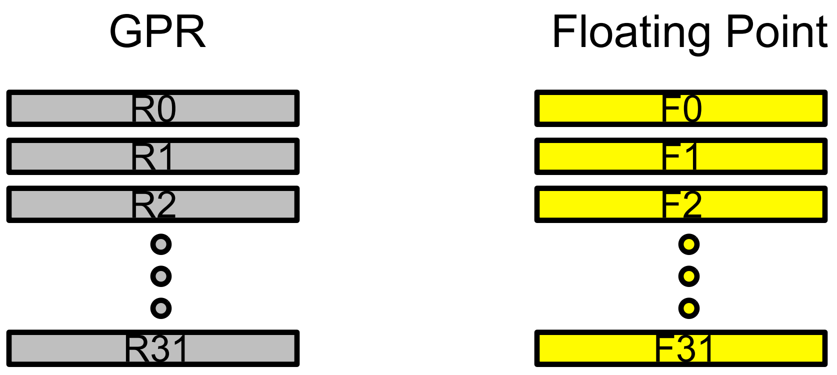
- ✦ Instruction sets reduced on essentials
- ✦ machine instruction is simple and regular, fixed instruction lengths
  - all instructions decodeable with same scheme
- ✦ few and simple addressing modes but many registers

✓ simple and efficient instruction execution

✓ high clock rates possible

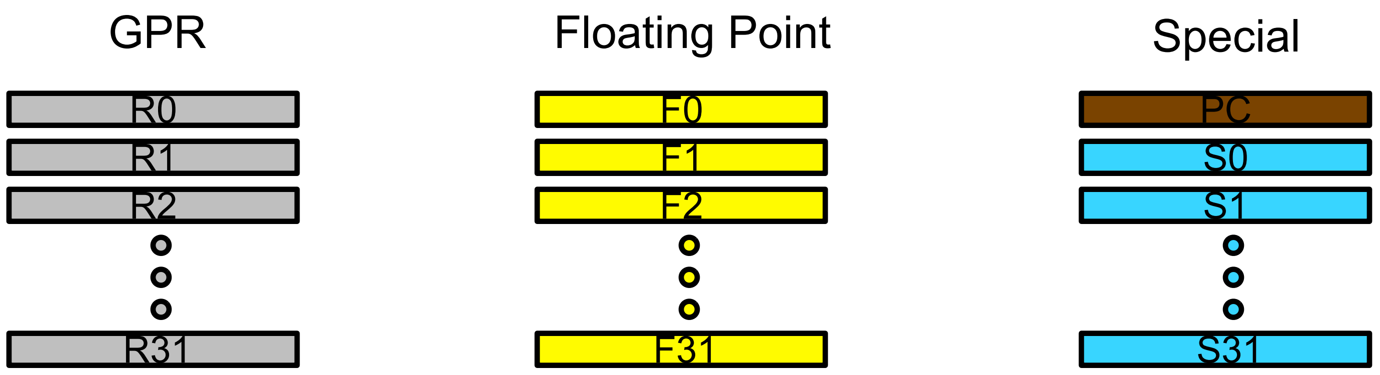
Disadvantage: Code mostly longer than on CISC−machines

DLX register file



- ▢ Relative large number of General-Purpose−Registern (GPR) R0...R31, 32 Bit, e.g. for integer−values, R0 is always zero!
- 🟨 F0...F31 are floating-point register, for single precision IEEE−Format FP−numbers or Double Precision as register−pairs (64 Bit) F0 & F1, F2 & F3, ..., F30 & F31

DLX register file



- 🟫 Programmcounter (PC), stores memory address of next instruction (must be always a multiple of 4 as every DLX instruction is 32Bit long but the memory is byte-addressable)
- 🟦 S0...S31 are special registers: not used for operand storage but other functions like processor state word, Exception Control Register, ...

DLX instruction set

Contains instructions for

- data transfer from and to memory
  - Load/Store
- arithmetical and logical operations
  - Addition/Subtraction
  - Multiplication/Division
  - AND/OR/Shift
- program control
  - Jump/Branch

DLX instruction format

Every DLX instruction is a 32 Bit word. There are 3 groups:

I−Type

Opcode	Rs1	Rs2	Immed−16

Generally for arithmetical and logical instructions with immediate−operand and for branch instructions. The immediate−operand or rather the displacement is encoded in the lower 16 Bit.

DLX instruction format

R−Type

Opcode	Rs1	Rs2	Rd	Unused	Func

For arithmetical and logical instructions operating only on registers.

J−Type

Opcode	Immed−26

For unconditioned jumps. Immediate may be 26 Bit.

DLX instruction format

- Operands are registers only or register and immediate.

```
ADD         R1, R2, R3      (R1 <- R2 + R3; R-Type)
ADDI        R1, R2, #3      (R1 <- R2 + 3; I-Type)
```

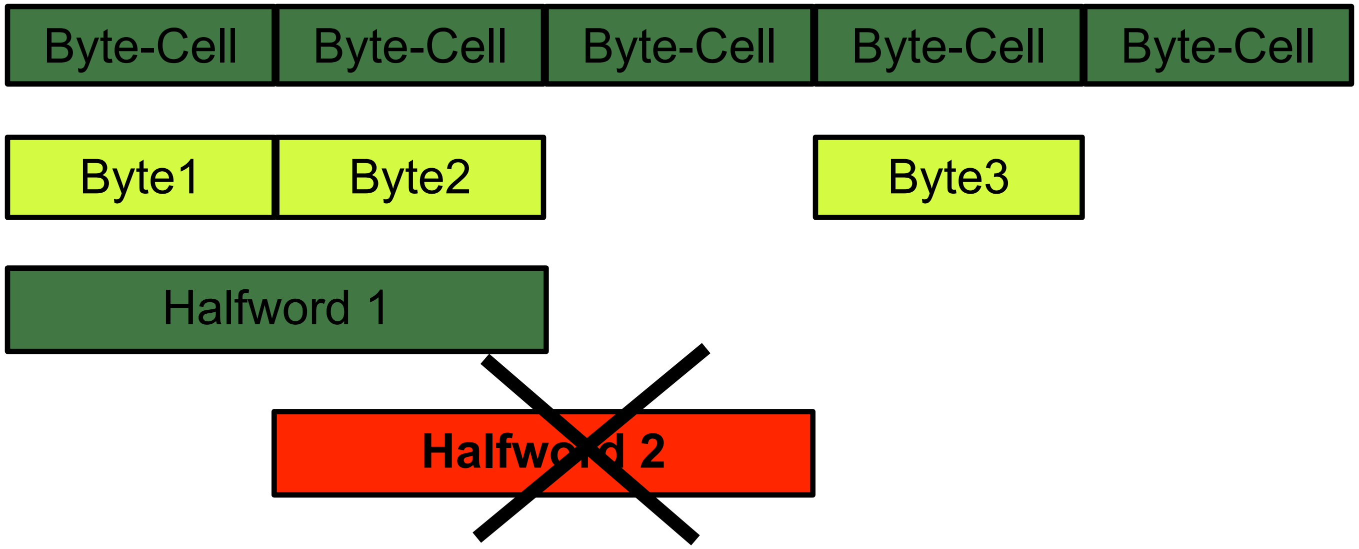
- Differentiation between byte, halfword, word, unsigned and signed.
- For floating point between single precision and double precision.

DLX instruction format

- Memory is byte−addressable

```
LW R1, 32(4*R2)    (R1 <− M[32+4*R2];  I−Type)
```

Memory accesses must be aligned!

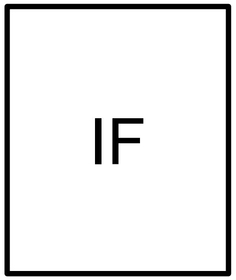


DLX instruction execution

The instruction execution of DLX can be spearated into five basic steps.

1. Instruction fetch

MAR <− PC; IR <− M[MAR]

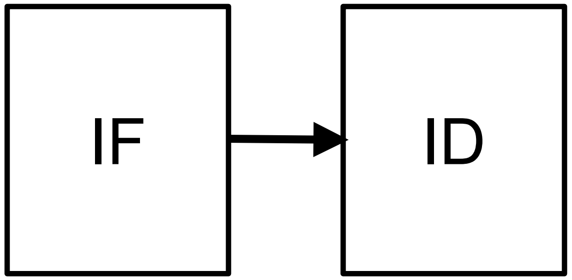


DLX instruction execution

The instruction execution of DLX can be spearated into five basic steps.

2. Instruction decode/register fetch

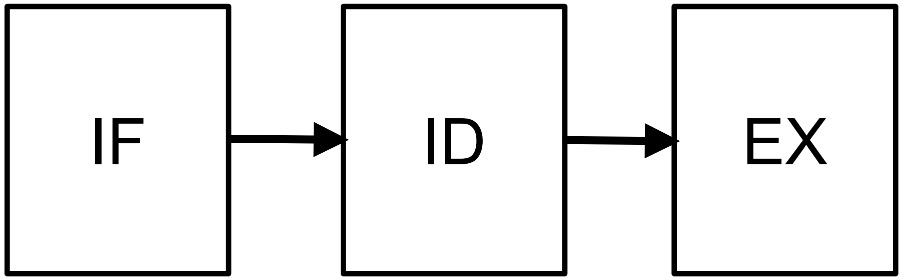
A <– Rs1; B <– Rs2; PC <– PC + 4



DLX instruction execution

The instruction execution of DLX can be spearated into five basic steps.

3. Execution / calculation of effective address



DLX instruction execution

3. Execution / calculation of effective address

Memory access

MAR <− A + (IR 16)16##IR16..31; MDR <− Rd

ALU instruction:

ALUoutput <− A op (B or (IR16)16##IR16..31)

Branch/jump:

ALUoutput <− PC + (IR16)16##IR16..31; cond <− (A op 0)

DLX instruction execution

4. Memory access/Branch completion

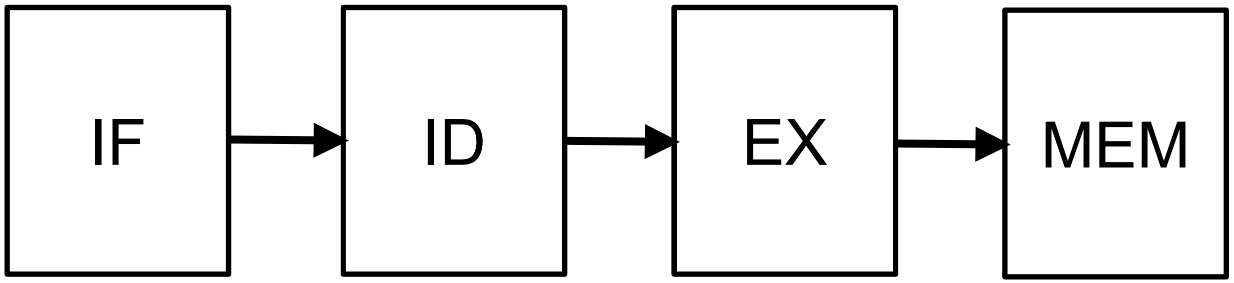
Only load, store, branch and jump instructions are active.

Memory access:

MDR <– M[MAR] or M[MAR] <– MDR

Branch:

If (cond) PC <– ALUoutput (branch)

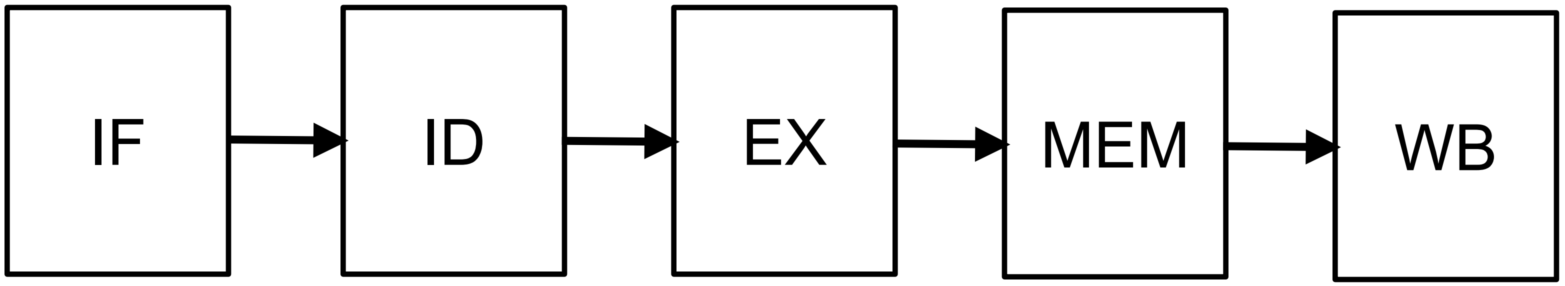


DLX instruction execution

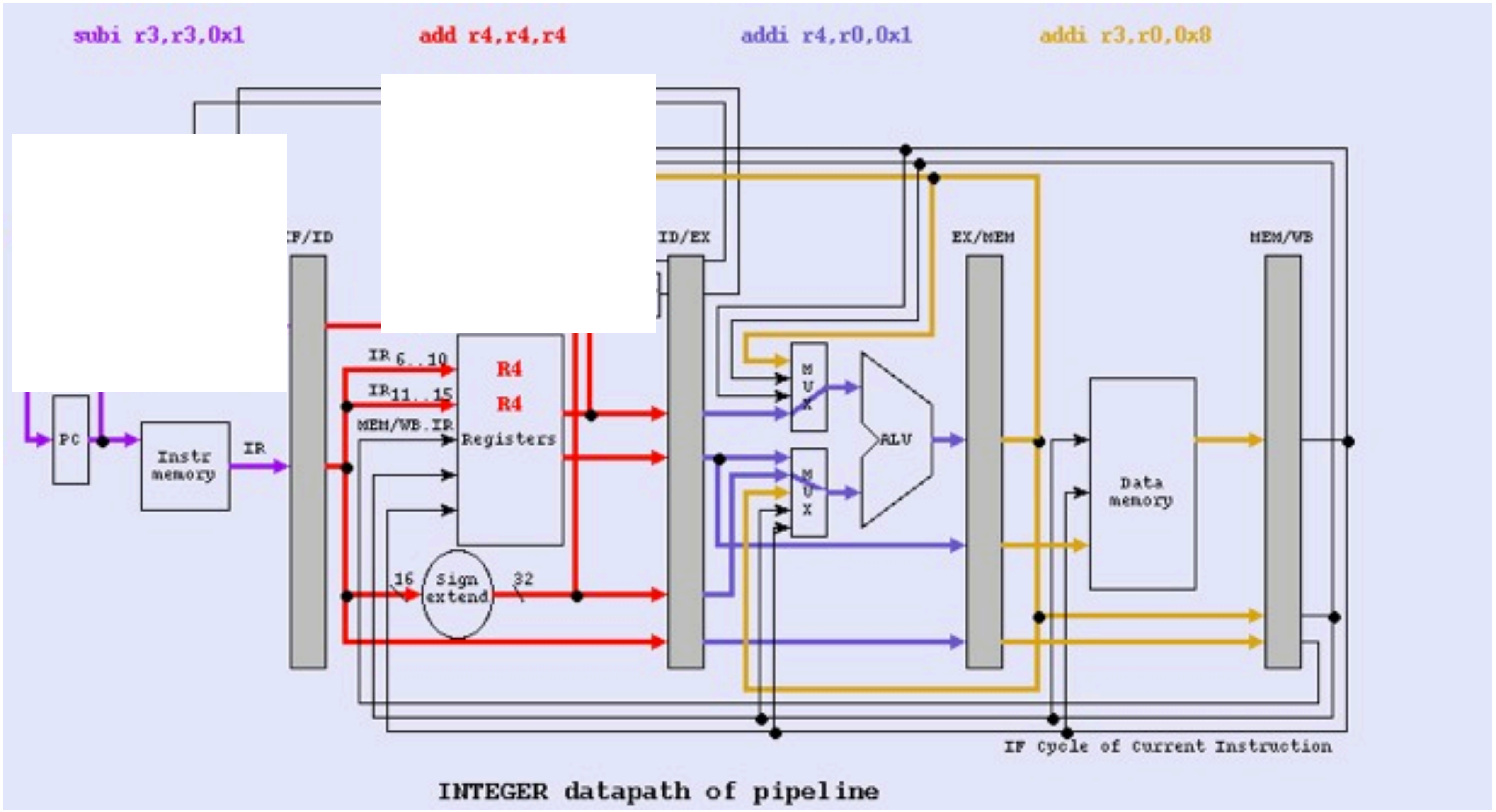
The instruction execution of DLX can be spearated into five basic steps.

5. Write back

Rd <− ALUoutput or MDR



DLX simulator



2. (Re-)Introduction to Assembly language

(Re-) Introduction in Assembler

A processor only understands ‘0’ and ‘1’:

Instruction for x86: **10110000 01100001**

Input in this form is confusing for most people

as Assembler: **mov al, 61h**

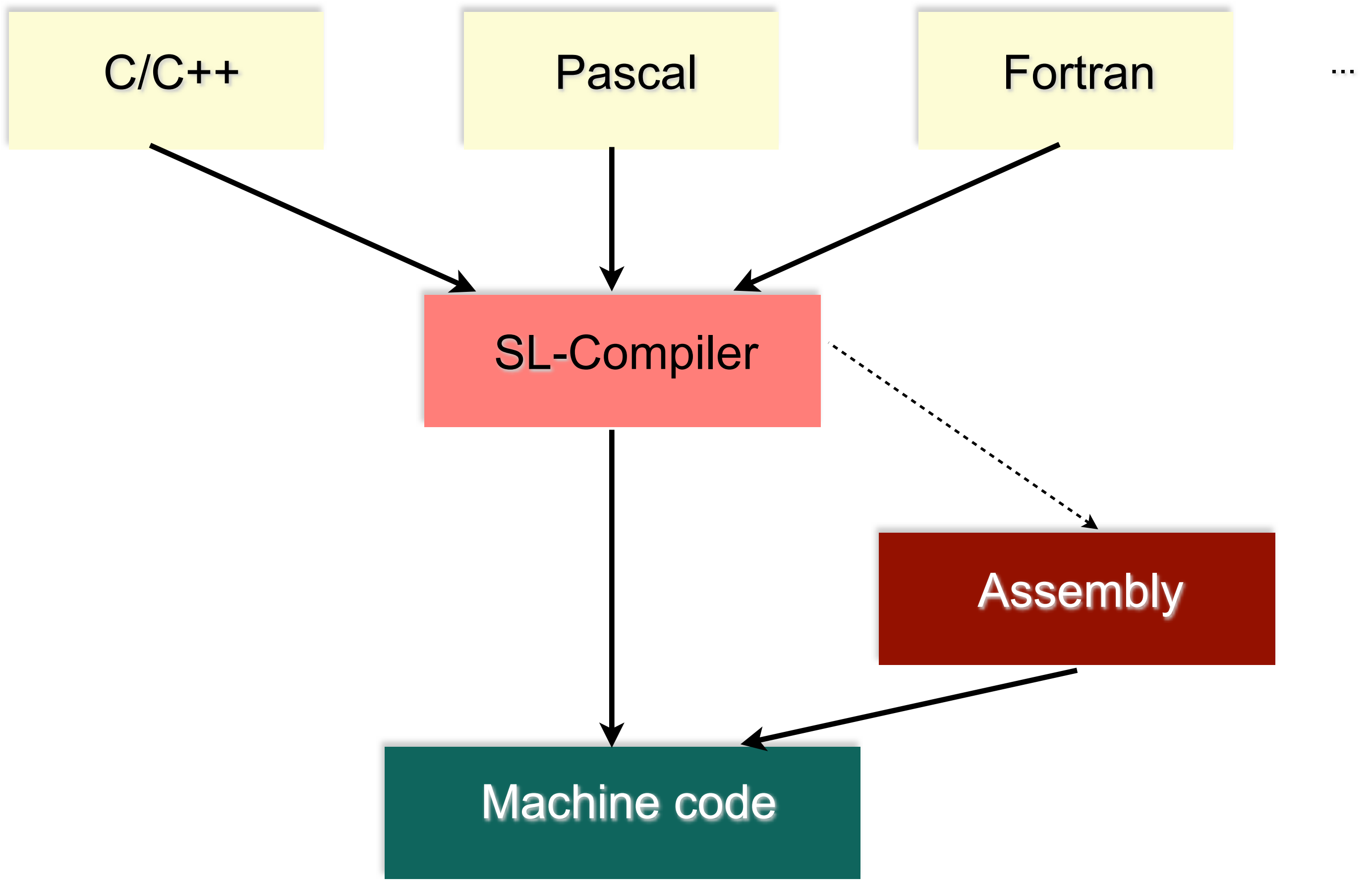
Allows direct manipulation of the instruction on processor

User can write compiler-independent optimized code

(Re-) Introduction in Assembler

- CPU machine code and Assembly instructions have fixed components: Operation code (OPC) and operands (0...n)
- Standard languages (C/C++, Pascal, ...) are problem focussed, not hardware focussed, they abstract the solution algorithm from the machine and compile
- so overhead compared to direct input of machine code/assembly instruction
- To investigate processor behavior we need access to the processes executed in the hardware - only possible using Assembly language
- For hardware-related software Assembly language is still in heavy use today

(Re-) Introduction in Assembler



(Re-) Introduction in Assembler

- Assembly language is always **machine dependent**!
- In many cases more than one Assembly-instruction is necessary to represent a standard-language instruction
- Assembly programs will be assembled, so additionally to the direct machine code conversion also labels and variable names are converted to memory addresses
- Why Assembly language?
  - Direct manipulation/readout of machine state
  - Efficient program structure possible

(Re-) Introduction in Assembler

Basic syntax rules:

;	comment start sign (one line)
:	label sign (e.g. for symbolic jumps)
.	for keywords
Tab	as separator
#	optional sign for numbers (immediate)

linewise interpretation of code (max. 255 characters per line)

A tab is necessary at the beginning of every instruction (exception: label)

(Re-) Introduction in Assembler

Two global segments:

.text	Code area (starts on DLX at address 100h)
.data	Data area
.align	Forces memory alignment (e.g. for labels)
.global	Global declaration of names (multiple files)
.space	Shift of memory pointer (creates empty cells)

(Re-) Introduction in Assembler

Datatypes on DLX:

.ascii	Character/String without end sign
.asciiz	Zero-terminated string
.byte	Byte (8 bit)
.word	Word (32 bit)
.float	Floating point number, single precision (32 bit)
.double	Floating point number, double precision (64 bit)

(Re-) Introduction in Assembler

Addressing modes at DLX:

Addressing Mode	Example
direct operand	addi r3, r0, #10
registers	add r4, r3, r3
direct addressing	lw r7, #1000
indirect addressing	lw r8, 0(r3)
indexed addressing	lw r9, 10(r4)

DLX-Instruction (Overview)

Data transport

Load (from memory): LB, LW, LF, ...

Store (to memory): SB, SW, SF, ...

Moving between registers: MOV

Arithmetic/Logic

ADD, SUB, MULT, AND, OR

LHI, Shift, conditional set of registers

Control

conditioned/unconditioned jumps, OS-calls (traps)

Floating Point

ADDD, SUBD, MULTD, DIVD, Conversion functions, Comparison

(Re-) Introduction in Assembler

C-Functions at DLX:

using TRAP-command

trap #0	Program termination (exit)
trap #1	open()
trap #2	close()
trap #3	read()
trap #4	write()
trap #5	printf()

handover parameter: starting at address in register r14 (indirect addressing)

return value: in register r1 (no FP numbers possible)

DLX-Instruction (Overview)

Register

- 32 GP-register R0...R31
- 32 FP-register F0...F31 (usable also as 16 x double precision)
- Program counter
- further special registers

R0 has always the value 0!

Tasks to solve

Develop an assembly program step-by-step with the following functionality

- Calculate 5+6-7
- Generalize this (y = a+b-c) for the content of the memory cells with address 0 (a), 0x200 (b), 0x300 (c), 0x400 (y)
- Extend this to do the matrix operation |Y|= |A| + |B| - |C| with a fixed matrix dimension of 1x20 integer elements

Tasks to solve

Give Assembly code snippets for the following problems:

- if (r1>r2) then r3 := 5;
- if (r1 >= r2 and r1 < r3) then r4 := 5;
- Fill the memory cells 200-399 with 0!

3. Pipelining

Pipelining

Implementation technique in which multiple instructions are overlapped in execution.

- Today a key implementation technique to create fast processors.

Comparable to conveyor belt:

- ❖ The work done within an instruction is divided into small pieces.
- ❖ Every part of the pipeline (pipeline stage) completes a part of the instruction.
- ❖ Different pipeline−stages are active in parallel.
- ❖ Instructions enter the pipeline at the beginning, are executed in the stages and write there results at the end.

Pipelining

Throughput of a pipeline: How often finish instructions their execution?

The time necessary to transport an execution by one stage is called machine cycle. The length of such a cycle is determined by the slowest pipeline stage as all stages must run synchronously.

Pipelining

Designer's goal:

Convenient balance of the length of pipeline stages

Time between instructions on the pipelined processor (assuming ideal conditions) =

$$\frac{\text{Time between instructions on unpipelined machine}}{\text{Number of pipeline stages}}$$

So under ideal conditions the speed-up from pipelining is approximately equal to the number of pipeline stages

-> use many short pipeline stages?

Pipelining

Yes, usually smaller stages allow an increasing clock frequency

but:

! Minimum stage length determined by minimum segmentation of execution steps and by pipeline overhead (in substance latches – Earle Latches – and clock shift between stages).

Pipelining reduces the average execution time per instruction using parallelism between instructions in a sequential instruction flow. Pipelining is not visible for the user.

DLX basic pipeline

Execution of DLX instructions in five basic stages is already known:

1. IF – Instruction Fetch
2. ID – Instruction Decode and register fetch
3. EX – EXecution and effective address calculation
4. MEM– MEMory access
5. WB – Write Back

DLX basic pipeline

Realization of the pipeline
−> "simply" fetch a new instruction in every clock cycle.

clock number

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

DLX basic pipeline

Characteristics:

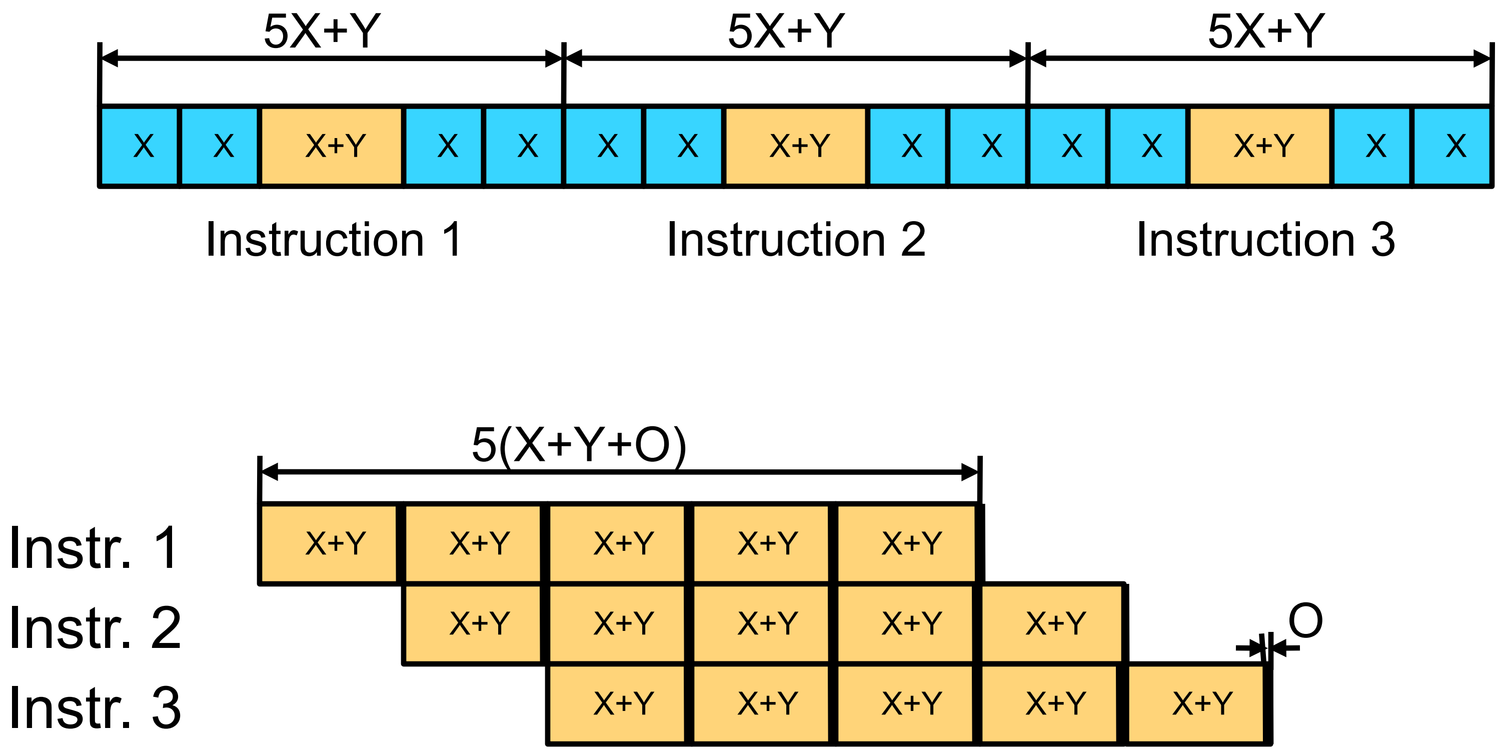
Every instruction still needs 5 clock cycles. The hardware executes in every clock cycle a part of 5 different instructions.

Pipelining increases the throughput of processed instruction - the number of the finished instructions per time unit. It does not reduce the execution time of a single instruction (“latency”)

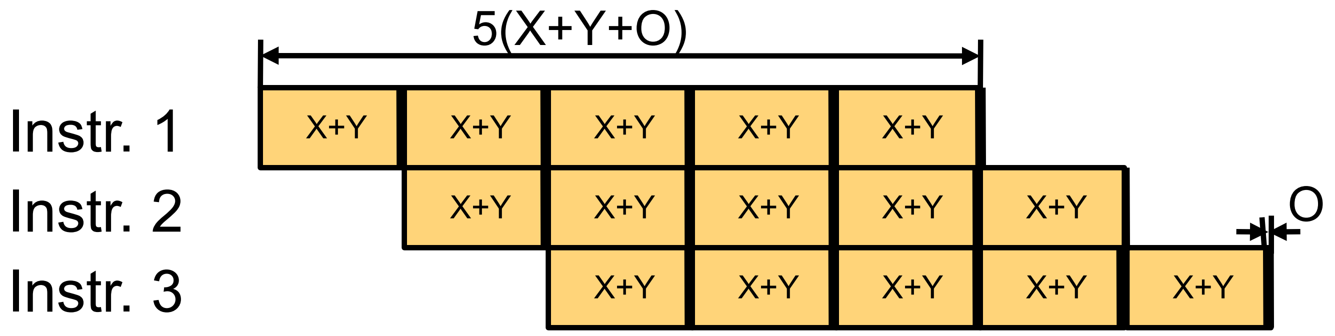
Limits of DLX−Pipeline:

- Execution time of single instruction remains the same (in practice even enlarged by latches and clock delays) = latency
- small pipeline depth

DLX basic pipeline vs. sequential



DLX basic pipeline vs. sequential



Example: X = 20 ns; Y = 5 ns; O = 3 ns;

Sequential execution:

Execution with pipeline:

Task

The instruction execution of machine X can be separated into 10 steps. 9 of those 10 steps need 50 ns, the 10. consumes 70 ns execution time.

A designer should implement machine X as pipeline. Every pipeline stage causes an additional delay of 10 ns.

Alternatively the designer can choose machine Y: It contains 9 execution steps. 8 of those steps need 55 ns, the 9. consumes 65 ns execution time.

Compare the instruction throughput of a sequential and a pipelined implementation without hazards and calculate the acceleration. How does the pipeline depth affect the hazard handling? Which of the two ways of implementation would the designer prefer?

Pipeline problems

The instruction overlap must not overcharge the ressources!

Example:

Simple ALU cannot compute an effective address and subtract operands in parallel

As every stage is active in every clock cycle all operations of a stage must be finished within this cycle and every combination of instructions must be executable in parallel.

Pipeline problems

1. PC must be incremented in every cycle, instead of ID now in IF; ALU is always in use - extra incrementer necessary
2. new instruction fetch in every cycle − in IF stage
3. in every cycle new data might be needed − in MEM
4. separate MDR for load and store necessary (LMDR and SMDR), as both instructions may overlap in time
5. three additional latches necessary for buffering instruction, PC and ALU result

Stage	Program Counter	Memory	Data path
IF	PC <− PC+4;	IR <− Mem[PC];	
ID	PC1 <− PC;	IR1 <− IR;	A <− Rs1; B <− Rs2;

EX			DMAR <− A+$(IR_{16})^{16}$ ##$IR1_{16...31}$; SMDR <− B; **or**
			ALUoutput <− A op (B or $(IR1_{16})^{16}$ ##$IR1_{16...31}$); **or**
			ALUoutput <−PC1 + $(IR1_{16})^{16}$ ##$IR1_{16...31}$);
			cond <− (A op 0);

MEM	If (cond) PC <− ALUoutput;	LMDR <− Mem[DMAR]; or Mem[DMAR] <− SMDR;	ALUoutput1 <− ALUoutput;
WB			Rd <− ALUoutput1; or Rd <− LMDR;

Pipeline problems

The ressource MEMORY probably has the biggest influence on pipelining

The memory access time is equal!

But:

The peak memory bandwidth must be 5 times larger than for a machine without pipeline (two mem accesses per cycle vs. two accesses within 5 cycles in sequential case).

So most computers use separated instruction and data caches (Harvard− Architecture ...).

Pipeline Hazards

The pipeline would work pretty if all instructions are independent.

Instructions may depend on its predecessors!

Hazards are situations, which forbid the execution of the next instruction in the associated clock cycle.

→ **reduce the performance improvement of the pipeline under ideal conditions**

Pipeline Hazards

Structural hazard:	results from a ressource conflict
Data hazard:	caused by an instruction which depends on the result of a preceding one in a matter that would fail on overlapped execution.
Control hazard:	caused by pipelining of instructions which change the PC (jumps, branches)

The control unit must detect hazard–situations!

Hazards can lead to stalls!

Structural Hazards

An instruction sequence cannot be executed caused by ressoruce conflicts (e.g. a functional unit is not fully pipelined or not enough FUs)

Example: only one memory interface for program and data (Princeton architecture)

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

Data Hazards

The order of operand access is changed by the pipeline opposite to the normal order at sequential execution.

e.g.:

```
ADD R1, R2, R3

SUB R4, R1, R5
```

ADD writes R1 in WB, SUB reads R1 in ID −> if dependency is not recognized SUB will read a wrong (old) value.

	1	2	3	4	5	6
ADD	IF	ID	EX	MEM	WB-Write	
SUB		IF	ID-Read	EX	MEM	WB

Solution: Instruction−Scheduling (static by SW or dynamic by HW) and

Data Hazards

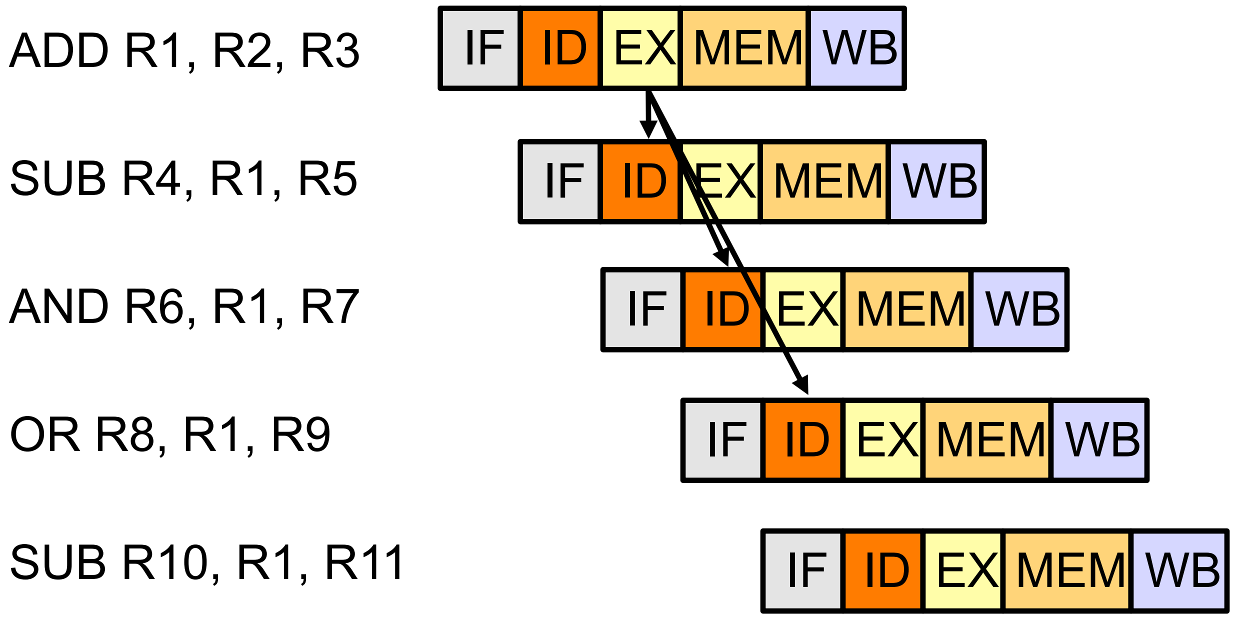
Forwarding

The ALU result is always forwarded to the ALU input latch (generally forwarded to EUs) and so available ahead. The control-hardware must recognize this dependency and choose the right ALU input value.

To consider: The following instruction also needs forwarding as the register write is only finished at the end of WB.

Data Hazards

Forwarding



Remark: Bypass for the third result is not necessary of register file is accessible twice in a clock cycle (first half: WB; second half: ID)

Types of Data Hazards

- RAW (read after write)
- WAR (write after read)
- WAW (write after write)

Control Hazards

Can cause a bigger performance loss than data hazards in a pipeline!

Branch instr.	IF	ID	EX	MEM	WB					
Instr. i + 1		IF	**S**	**S**	IF	ID	EX	MEM	WB	
i + 2			**S**	**S**	**S**	IF	ID	EX	MEM	WB
i + 3				**S**	**S**	**S**	IF	ID	EX	MEM
i + 4					**S**	**S**	**S**	IF	ID	EX
i + 5						**S**	**S**	**S**	IF	ID
i + 6							**S**	**S**	**S**	IF

→ **3 wait cycles!**

Reducing the number of wait cycles:

1. Early recognition of taken or not taken branch case in pipeline.
2. Early calculation of target address for PC if branch is taken

Control Hazards

1. Branch condition in DLX (BEQZ and BNEZ) is simply test on zero.
   - ✦ Decision can be made by a special logic until the end of ID cycle

2. PC must be calculated for both cases (branch taken/not taken).
   - ✦ Separate adder at ID phase and MUX necessary

Leads to only one wait cycle at branches (delayed slot).

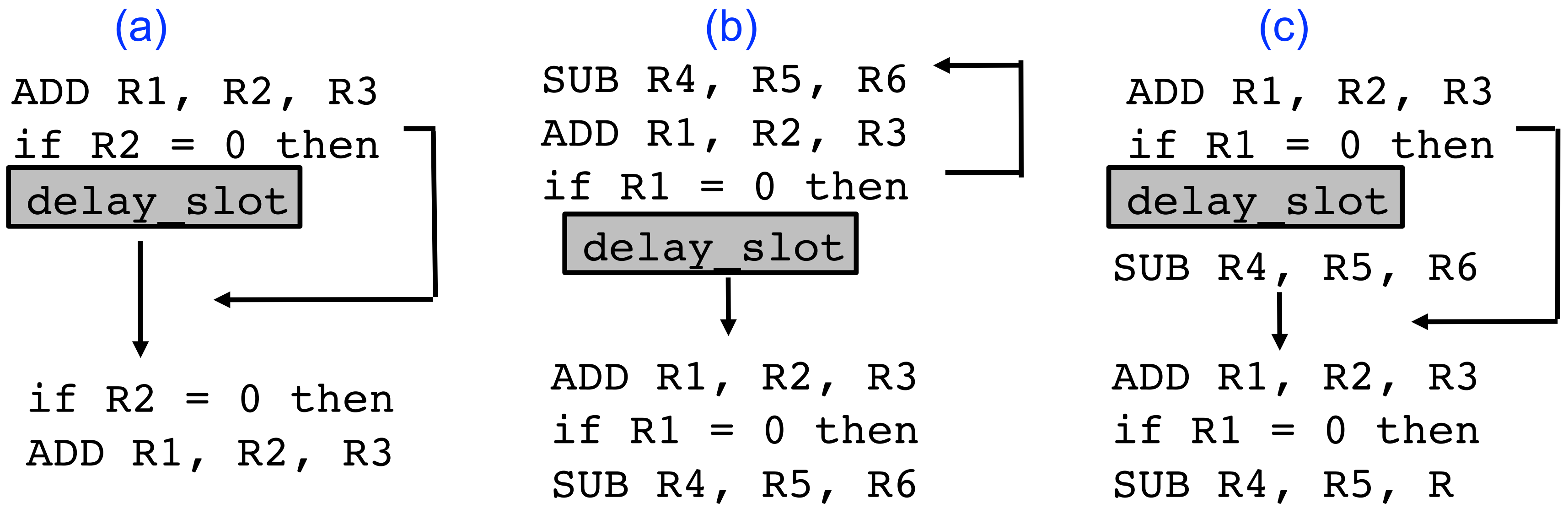
Control Hazards

Leads to only one wait cycle at branches (delayed slot).

Pipeline stage	Branch instruction
IF	IR <− Mem[PC]; PC <− PC + 4
ID	A <− Rs1; B <− Rs2; PC1 <− PC; IR1 <− IR; **BTA <− PC + ((IR16)16## IR16...31)** **If (Rs1 op 0) PC <− BTA**
EX	
MEM	
WB	

Control Hazards

Scheduling−Strategies



(a) Rearrangement of an instruction before branch (best strategy)

(b) Replacement with branch target (usually needs copying)

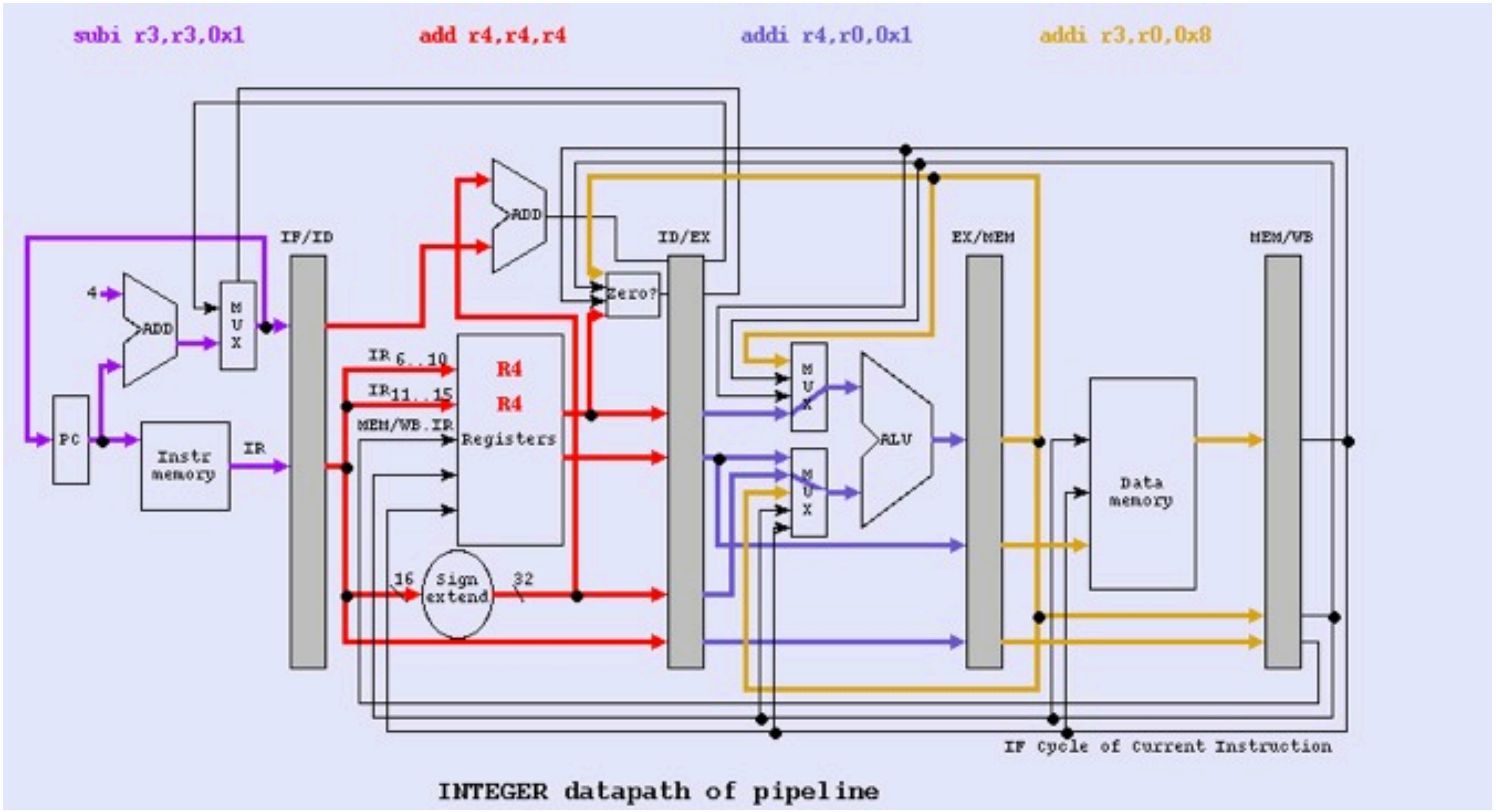
(c) Rearrangement of an instruction after branch

Control Hazards

DLX−Pipeline: Frequency of control instructions 14%
where 65% change the PC

Scheduling−Scheme	Branch−losts	Effective CPI	Acceleration compared to machine without pipeline	Acceleration compared to pipeline with wait cycles
Pipeline with wait cycles	3	1,42	3,52	1
Prediction: taken	1	1,14	4,39	1,25
Prediction: not taken	1	1,09	4,59	1,3
Delayed slot	0,5	1,07	4,67	1,33

DLX simulator



Extension of DLX for FP operations

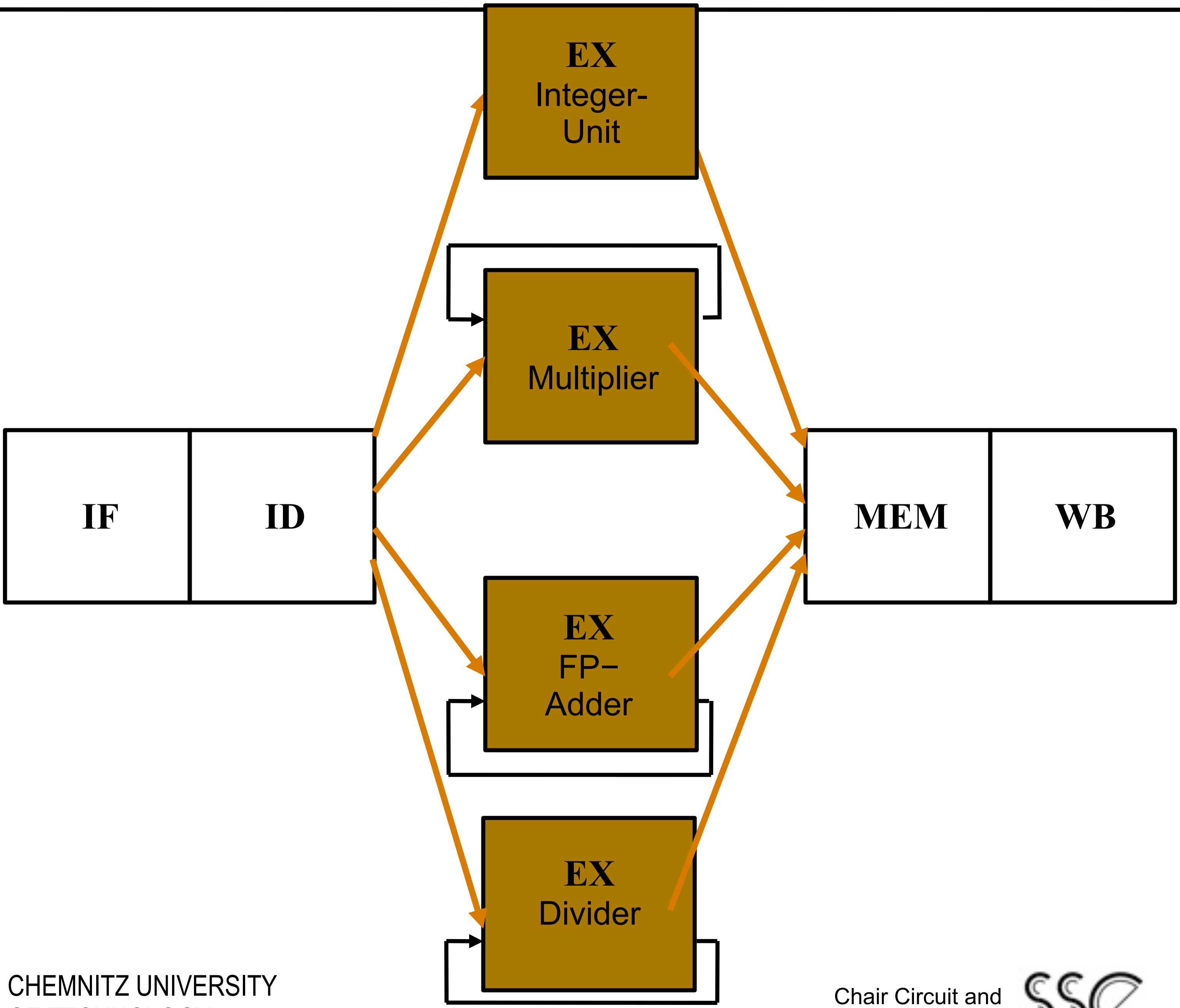
Execution of DLX−FP−operations in one or two clock cycles would either lead to a slow clock frequency or the usage of a huge amount of logic in the FPU!

Changes against integer pipeline:

- ✦ EX−cycle can be repeated as often as needed to finish operation.
- ✦ More than one FP-unit might be possible.

A wait cycle will occur, if the instruction under execution leads to

- ✦ a structural hazard for a functional unit
- ✦ a data hazard.


EX
Integer-
Unit
EX
Multiplier
IF
ID
MEM
WB
EX
FP–
Adder
EX
Divider

Task

We know that a pipeline implementation with 4 stages has the following hazard-frequencies and wait cycles between an instruction I and its successor:

Instruction	**Hazard-frequency**	**Number of wait cycles**
i + 1 (without i + 2)	20%	2
i + 2	5%	1

Assume that the clock frequency is the same as without pipelining. What is the effective advance of the pipeline compared to a sequential machine with and without consideration of hazards?

Task

```
S1:    ADDI          R1, R2, #2
S2:    SUB           R4, R1, R3
S3:    SGE           R7, R4, R0
S4:    BNEZ          R7, S7
S5:    SUB           R3, R5, R6
S6:    J             S8
S7:    ADDI          R3, R3, #2
S8:    ADDI          R4, R4, #1
```

Assume a 5-stage DLX instruction pipeline. If a jump occurs, the program counter is set to the target after the fourth stage. Specify all data and control dependencies in this sequence (even without affecting the flow).

Which dependencies could lead to hazards/stalls if there are no further hardware mechanisms like forwarding? Add the necessary stall cycles to the sequence! How many machine cycles are necessary in the pipeline?

4. Dynamic Scheduling

Dynamic 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 –> 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 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.

Dynamic Scheduling

Necessary: Test on structural hazards and absence of data hazards.

The execution of an instruction can start as soon as the operands are available and no data hazards occur. Tests on structural hazards can still be done during execution phase.

Two methods:

- Scoreboarding
- Tomasulo−Algorithm

} Effort reduction of data dependencies

Out-of-Order execution at DLX

Separation of stages:

1. ID − Instruction decode and test on all hazards and operand fetching
2. EX − Execution

into three stages:

1. Issue − instruction decode and test on structural hazards
2. Read operands − wait until no hazards occur and then operand reading
3. Execution

The issue stage is passed by all instructions in normal order.

In the second stage (read operands) instructions can stop or provide results to each other and so enter an Out-of-Order execution which leads to an Out-of-Order termination

WAW−Hazards, WAR−Hazards

4.a) Scoreboard

Scoreboard

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

(named after 6600−Scoreboard, Control Data Corporation 1964)

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

Function: Control of instruction execution.

- Every register owns Valid−Bit
- Every instruction passes Scoreboard, data dependencies and target registers are logged;
- Logs determine when instruction is allowed to read its operands and start execution
- Future result register will be set to invalid after ID as content changes if instruction finishes

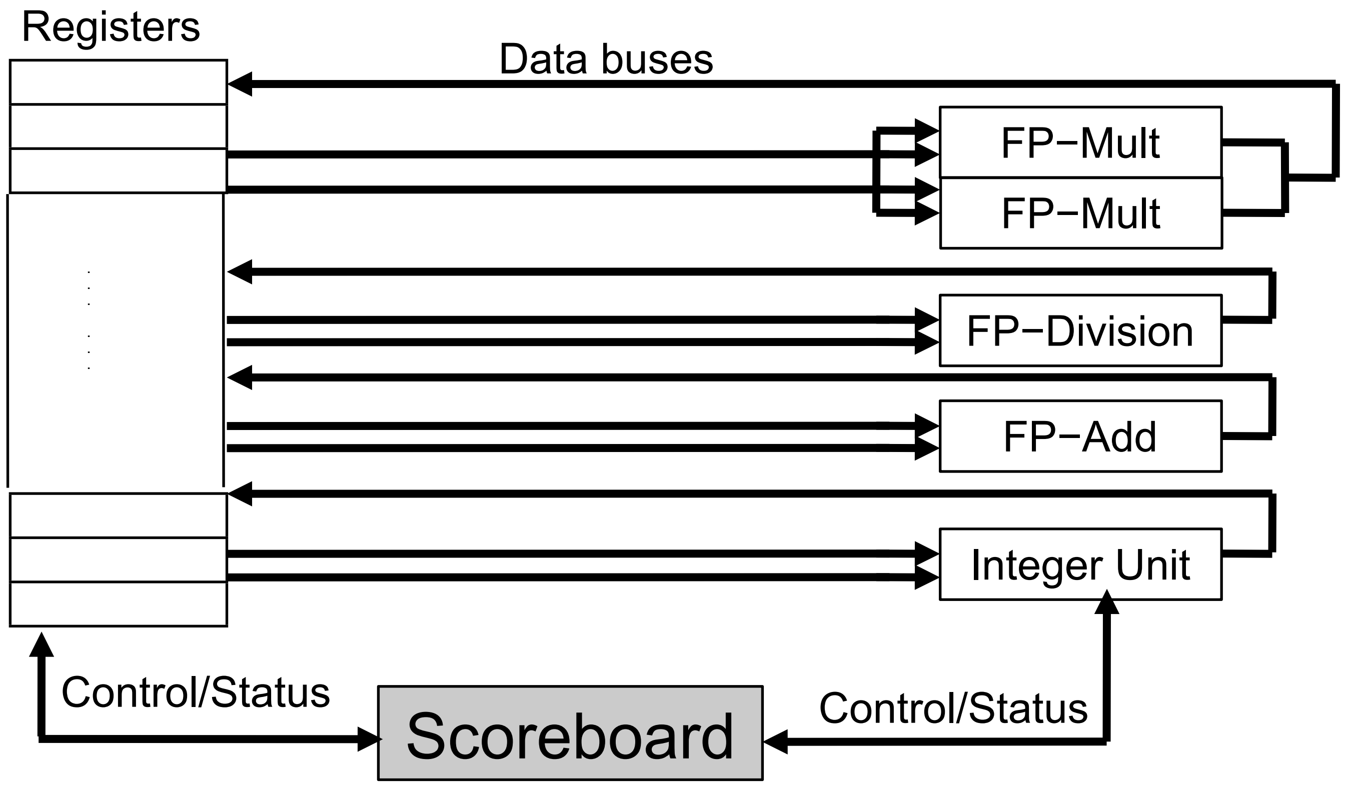
Scoreboard on DLX

Function: Control of instruction execution (continued)

- After WB−phase target register is set to valid
- If Scoreboard decides that instruction will not be executed at the moment (instruction tries to read invalid register or ressources are busy) it supervises every HW change and decides when instruction may start.
- Scoreboard also monitors when the instruction may write the target register.

→ **Centralization of whole hazard detection and resolution in Scoreboard.**

Scoreboard on DLX



Multiple instructions may be in EX−stage in parallel!

→ Multiple functional units or pipelined FUs necessary.

Execution steps of Scoreboard on DLX

1. Issue:

If the functional unit is free and no other active instruction has the same destination register (**WAW**), then issue the instruction to the functional unit and update scoreboard tables; otherwise stall until the **structural hazard** is cleared.

2. Read operands:

Scoreboard monitors the availability of the source operands. When the source operands are available, scoreboard tells the functional unit to read the operands and begin execution. An operand is available if no active instruction is going to write it or being actively written. (dynamic RAW resolution)

Execution steps of Scoreboard on DLX

3. Execution:

The FUs start execution after operands are read. If the result is available the Scoreboard is notified that execution has finished.

4. Write result:

If Scoreboard notices that FU has finished execution it tests on WAR hazards. An instruction is not allowed to write its result if:

- there is an instruction which has not read its operands
- one of the operands is in the same register as the result of the finishing instruction and
- the other operand is a result of a previous instruction

If no WAR hazard occurs or if it is solved, the Scoreboard signals the FUs to store their results in the target register.

Execution steps of Scoreboard on DLX

Aim:

Reach an execution rate of one instruction per clock cycle (if there are no structural hazards) by executing an instruction as soon as possible.

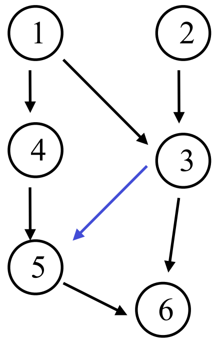
The Scoreboard bears the full responsibility for instruction issue and execution including hazard detection. It controls the instruction execution step-by-step based on its own data structures by communication with the functional units.

Execution steps of Scoreboard on DLX

Problems of Scoreboard:

- All results are written through the register file and never by Forwarding! If an instruction writes its result a dependent instruction must wait for access to the register file. This enlarges latency and limits the possibility to initialize multiple instructions which wait for the result.
- **WAR− and WAW−Hazards are possible!**

Execution steps of Scoreboard on DLX


$$\begin{tikzcd}
1 \arrow[d] \arrow[dr] & 2 \arrow[d] \\
4 \arrow[d] & 3 \arrow[dl] \arrow[d] \\
5 \arrow[r] & 6
\end{tikzcd}$$

```
.data 32
.double 4.06
.data 40
.double 10.47
```

```
.text
1            ld           f2,     32(r0)
2            ld           f4,     40(r0)
3            multd        f6,     f4, f2
4            subd         f8,     f2, f2
5            divd         f4,     f2, f8
6            addd         f10,    f6, f4
```

Execution steps of Scoreboard on DLX

The DLX−Scoreboard contains three parts as tables:

1. Instruction Status: In which stage is the instruction located?
2. Functional Unit Status: 9 fields for every FU:

   Busy.........Is the unit in use?

   Op............Next operation in FU

   Fi..............Target register

   Fj, Fk........Source register addresses

   Qj, Qk.......Number of FU, which provides the register Fj, Fk

   Rj, Rk.......Valid−Bits for Fj and Fk.

   Are cleared when new values are read to notify scoreboard control.
3. Register Result Status: Shows which FU wants to write a result when an active instruction has this target register.

Instruction Status

Instruction	Issue	Read ops	Exec finished	Write result
ld f2, 32(r0)	x	x	x	x
ld f4, 40(r0)	x	x	x	-
multd f6, f4, f2	x	-	-	-
subd f8, f2, f2	x	x	-	-
divd f4, f2, f8	-	-	-	-
addd f10, f6, f4	-	-	-	-

Functional Unit Status

FU	By	Op	Fres	Fj	Fk	Qj	Qk	Rj	Rk
Integer	x	ld	f4	r0	-	-	-	-	-
Mult1	x	multd	f6	f4	f2	Integer	-	ja	-
Mult2	-	-	-	-	-	-	-	-	-
Add	x	subd	f8	f2	f2	-	-	-	-
Div	-	-	-	-	-	-	-	-	-

Register Result Status

f0	f2	f4	f6	f8	f10
-	-	Integer	Mult1	Add	-

Instruction Status

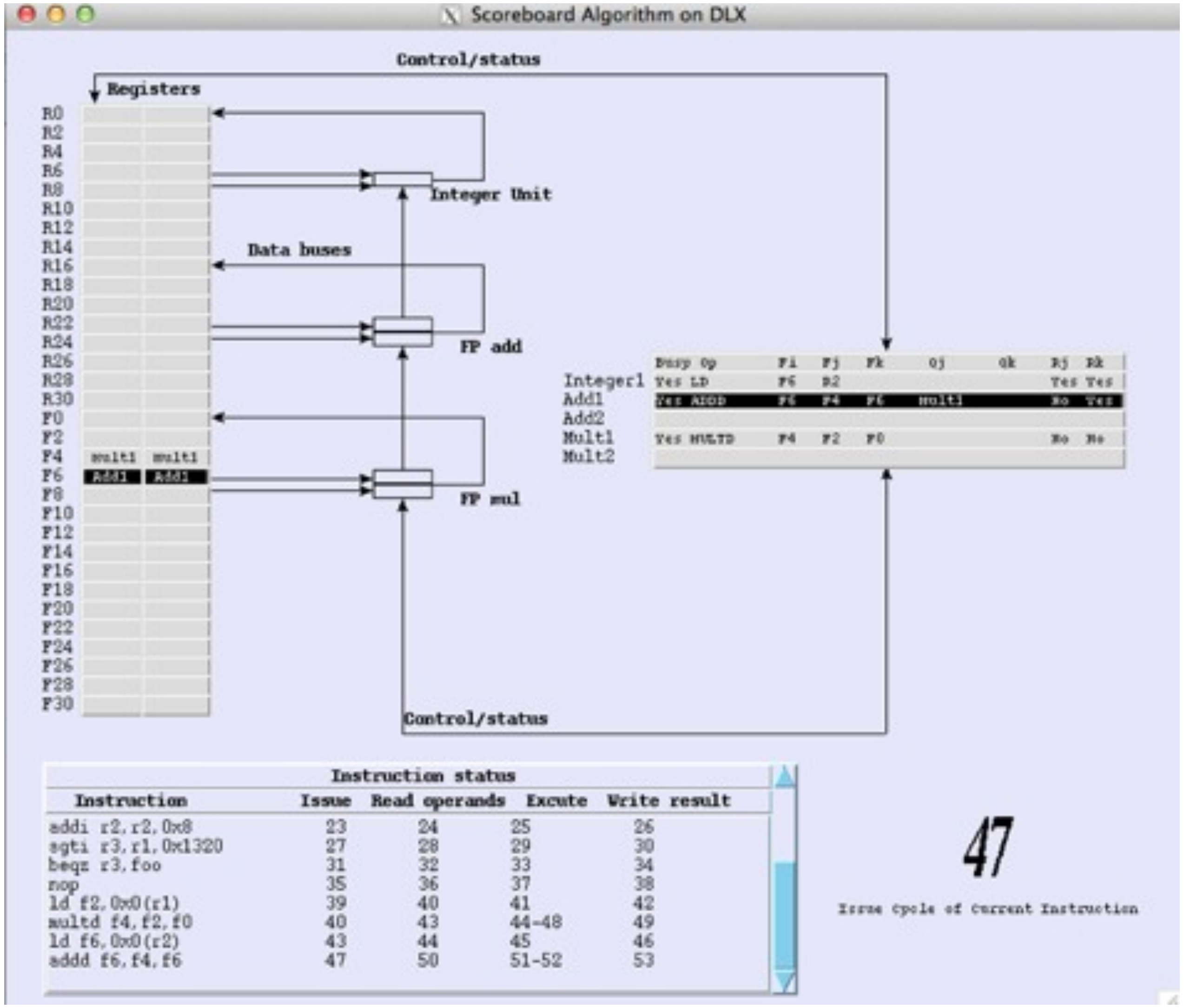
Instruction	Issue	Read ops	Exec finished	Write result
ld f2, 32(r0)	x	x	x	x
ld f4, 40(r0)	x	x	x	x
multd f6, f4, f2	x	x	-	-
subd f8, f2, f2	x	x	-	-
divd f4, f2, f8	x	-	-	-
addd f10, f6, f4	-	-	-	-

Functional Unit Status

FU	By	Op	Fres	Fj	Fk	Qj	Qk	Rj	Rk
Integer	x	ld	f4	r0	-	-	-	-	-
Mult1	x	multd	f6	f4	f2	-	-	-	-
Mult2	-	-	-	-	-	-	-	-	-
Add	x	subd	f8	f2	f2	-	-	-	-
Div	x	divd	f4	f2	f8	-	Add	-	yes

Register Result Status

f0	f2	f4	f6	f8	f10
-	-	Div	Mult1	Add	-



	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer1	Yes	LD	F6	R2				Yes	Yes
Add1	Yes	ADDD	F6	F4	F6	mult1		No	Yes
Add2									
Mult1	Yes	MULTD	F4	F2	F0			No	No
Mult2									

Instruction status

Instruction	Issue	Read operands	Excute	Write result
addi r2, r2, 0x8	23	24	25	26
sgti r3, r1, 0x1320	27	28	29	30
beqz r3, foo	31	32	33	34
nop	35	36	37	38
ld f2, 0x0(r1)	39	40	41	42
multd f4, f2, f0	40	43	44-48	49
ld f6, 0x0(r2)	43	44	45	46
addd f6, f4, f6	47	50	51-52	53

47

Issue Cycle of Current Instruction

4.b) Tomasulo

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

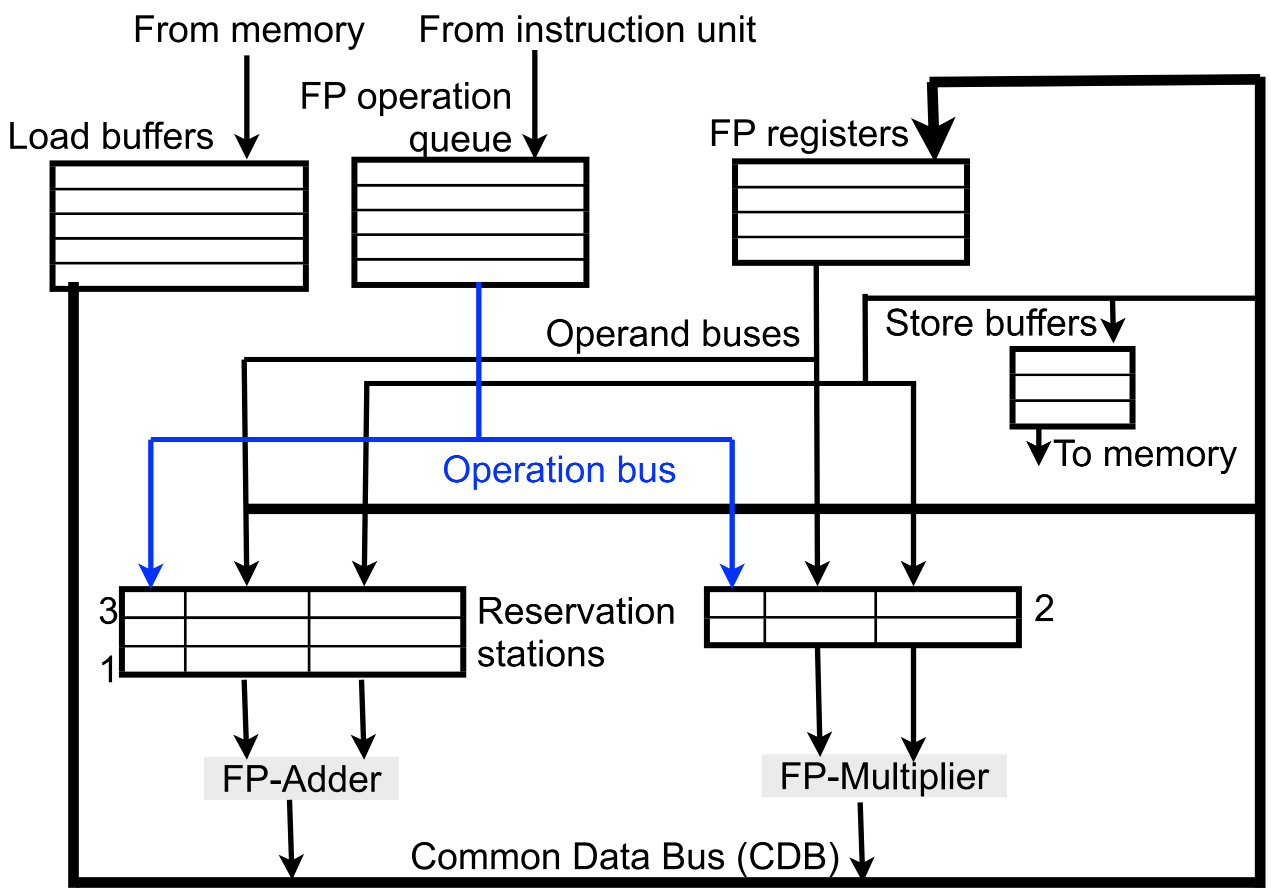
− focusses on floating point unit

− 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 CDB without passing the register file.

Structure of a FP unit using Tomasulo Algorithm



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 (Common Data 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)

Tomasulo Algorithm

Reservation station components at DLX - six cells:

Op	– Operation to be performed on source operands S1 and S2
Qj, Qk	– Reservation stations producing source registers. Zero means that source operand is already available in Vi or Vj or is not necessary.
Vj, Vk	– Values of source operands. Either V or Q is valid for one operand.
Busy	– Indicates reservation station or FU is busy

The register file and the store buffer have a field Qi:

Qi	– Address of FU which generates a value to be written to register or memory. If value is zero no active instruction calculates a result to be sent to register file or store buffer

Tomasulo Algorithm - Functionality

1. Issue:

   An instruction is provided by FP operations queue. The FP instruction can be taken from queue if there is an empty reservation station and the operand values shall be sent to reservation station if they are available.

   If it is a load or store instruction it can be delivered if there is a free buffer entry.

   If there is no empty reservation station or buffer then a structural hazard occurs and the instruction (and so the queue) stops until a station/buffer is available.

Tomasulo Algorithm - Functionality

2. Execution:

When one or more operands are not yet available the reservation station monitors the CDB. This stage tests on RAW hazards. If both operands are available the operation is executed.

3. Write result:

When the result is available it is written to the CDB. The CDB sends it to the target registers and to every FU which waits for this result.

Tomasulo Algorithm - Functionality

There is no test on WAW− and WAR−hazards! They are automatically eliminated by the algorithm.

Load and store units are treated as basic functional units.

The CDB is used for result transmission instead of waiting for registers.

Normal data bus: data + destination (“go to” bus)

Common data bus: data + source (“come from” bus)

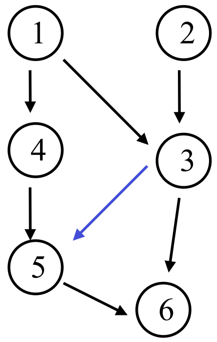
– 64 bits of data + 4 bits of Functional Unit source address

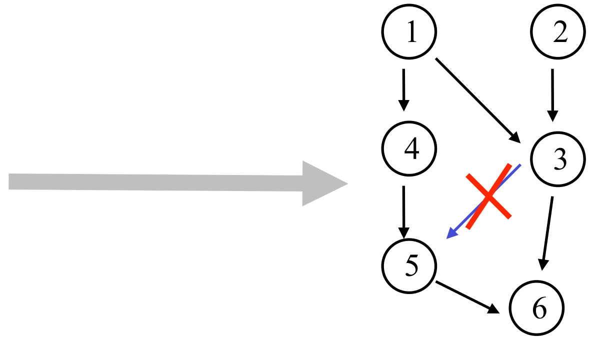
– Write if matches expected Functional Unit (produces result)

– Does the broadcast

Tomasulo Algorithm - Execution stages

```
  .data 32
  .double 4.06
  .data 40
  .double 10.47
```





implicit Register Renaming

(Tomasulo)

```
5 divd f4', f2, f8
6 addd f10, f6, f4'
```

```
.text
1         ld       f2,    32(r0)
2         ld       f4,    40(r0)
3         multd    f6,    f4, f2
4         subd     f8,    f2, f2
5         divd     f4,    f2, f8
6         addd     f10,   f6, f4
```

Reservation stations

Name	By	Op	Vj	Vk	Qj	Qk
Add1	x	sub	(Load1)	(Load1)	-	-
Add2	x	add			Mult1	Mult2
Add3						
Mult1	x	multd	-	(Load1)	Load2	-
Mult2	x	div	(Load1)	-	-	Add1

Register status

	f0	f2	f4	f6	f8	f10
Qi	-	-	Mult2	Mult1	Add1	Add2
Busy	-	-	x	x	x	x

No f4-WAW as Mult1 reads operand directly from CDB so Mult2 can write independently

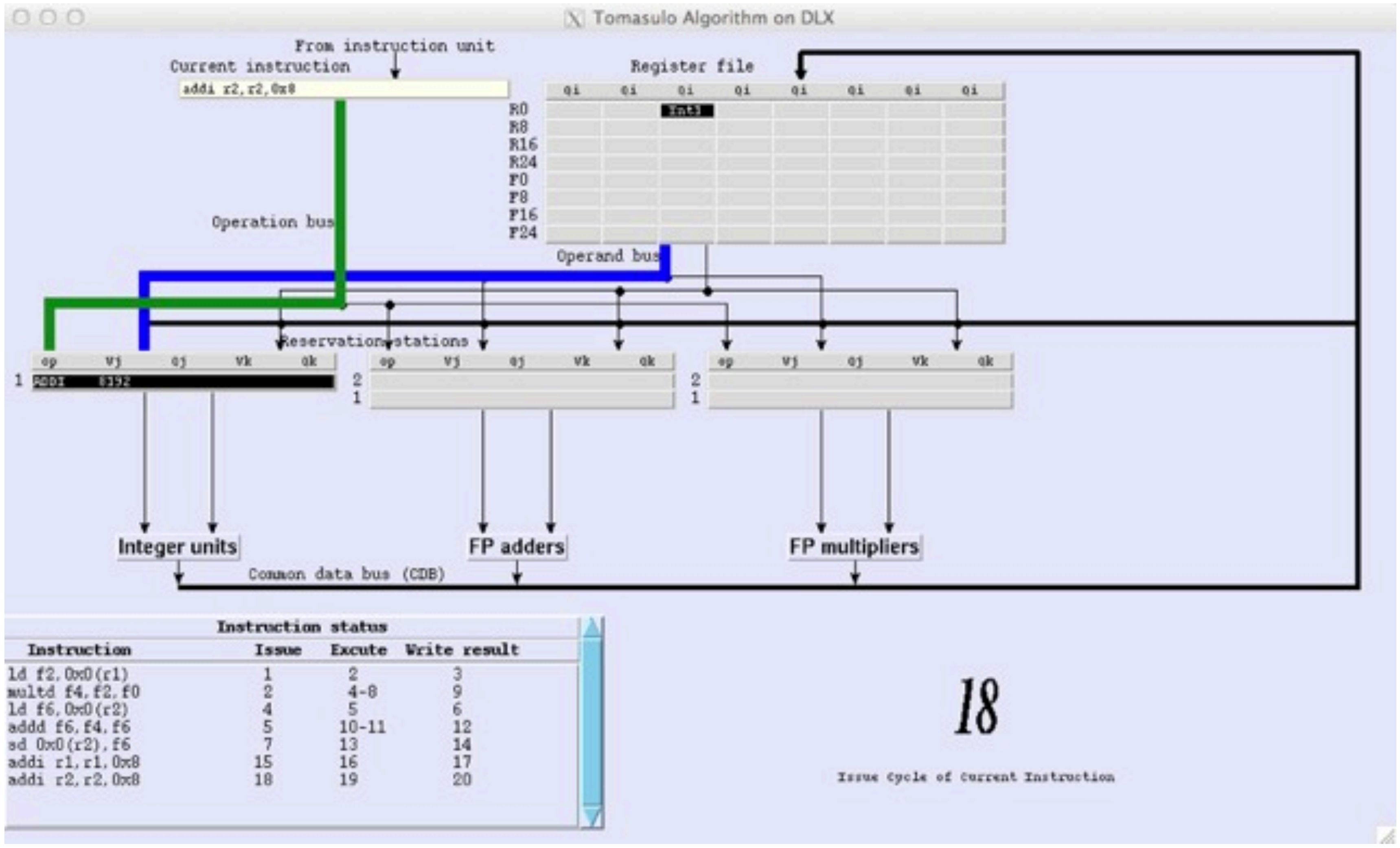
Reservation stations

Name	By	Op	Vj	Vk	Qj	Qk
Add1	x	sub	(Load1)	(Load1)	-	-
Add2	x	add			Mult1	Mult2
Add3						
Mult1	x	multd	(Load2)	(Load1)	-	-
Mult2	x	div	(Load1)	-	-	Add1

Register status

	f0	f2	f4	f6	f8	f10
Qi	-	-	Mult2	Mult1	Add1	Add2
Busy	-	-	x	x	x	x

No f4-WAW as Mult1 reads operand directly from CDB so Mult2 can write independently



Instruction status

Instruction	Issue	Excute	Write result
ld f2,0x0(r1)	1	2	3
multd f4,f2,f0	2	4-8	9
ld f6,0x0(r2)	4	5	6
addd f6,f4,f6	5	10-11	12
sd 0x0(r2),f6	7	13	14
addi r1,r1,0x8	15	16	17
addi r2,r2,0x8	18	19	20

Comparison Scoreboard - Tomasulo

	Scoreboard	Tomasulo
Start execution	If FU available	If reservation station empty
Read operands	from register file	from register file and CDB
Write operands	to register file	to CDB
WAW−, WAR −Hazards	Yes	No
Implicit Register Renaming	No	Yes
Instructions starting execution	1 per clock cycle	no limit
Instructions finishing execution	no limit	1 per CDB
Control Unit	centralized in Scoreboard	Distributed to FUs (reservation stations), instructions plan their execution on their own

Tasks

Develop two code snippets which show on the one hand the advantages of Scoreboarding and on the other hand the disadvantages (= advantages of simple pipelining).

Which additional components are necessary for Scoreboard?

Develop two code snippets which show on the one hand the advantages of Tomasulo and on the other hand the disadvantages (= advantages of simple pipelining).

Which additional components are necessary for Tomasulo?

Is it possible that Tomasulo is slower than Scoreboarding?

Task Register Renaming

Find the name dependencies (WAR, WAW) in the following code segment. Delete these dependencies by changing the register allocation.

```
loop:          ld          f0, 0(r1)
               addd        f4, f0, f2
               sd          0(r1), f4
               ld          f0, -8(r1)
               addd        f4, f0, f2
               sd          -8(r1), f4
               ld          f0, -16(r1)
               addd        f4, f0, f2
               sd          -16(r1), f4
               ld          f0, -24(r1)
               addd        f4, f0, f2
               sd          -24(r1), f4
               subi        r1, r1, #32
               bnez        r1, loop
               nop
```

5. Dynamic branch (target) prediction

Dynamic branch prediction

Problem: Control−Hazards limit pipeline performance substantially.

Usage of hardware for dynamic prediction of the result of a branch to reduce the branch delay.

Branch taken / not taken? next PC?

The prediction changes if the branch behavior changes during program execution.

Task

Assume the following frequencies of branches (as percent of all instructions):

Instruction	Frequency
Conditioned branches	20% (thereof 60% taken)
Jumps and SP-Calls	5%

We deal with a 4-stage-pipeline, where unconditioned branches are decided after 2 cycles and conditioned branches after 3 cycles. Assume, that the first pipeline stage is always carried out. Neglect all other hazards. How much faster would be a machine without control hazards?

Dynamic branch prediction

Dynamic branch prediction scheme:

Branch prediction buffer

Prediction of the taken/not taken case.

Contains a statistically statement of the taken/not taken case of branches.

No prediction of next PC!

Indexing using lower part of branch instruction address (fast). It is unknown whether prediction is right!

Instruction fetch can start only when PC is calculated! −> only useful if branch delay is bigger than the time for calculating target PC.

Branch target buffer

Prediction of address of the next instruction.

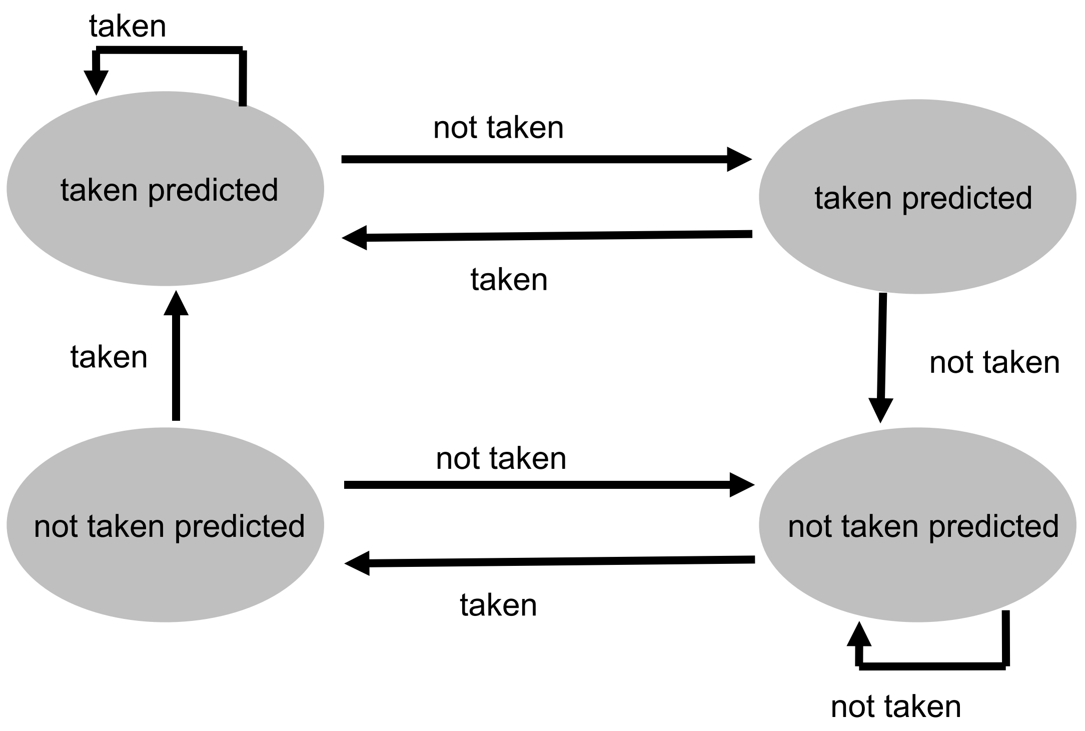
Contains beside the address an information whether the fetched instruction is a taken branch and whether the address in target buffer is a “taken” or “not taken” prediction.

Indexing of instruction with whole PC necessary as the next PC is predicted (must be correct).

Fetching next instruction starts before decoding (if in branch target buffer).

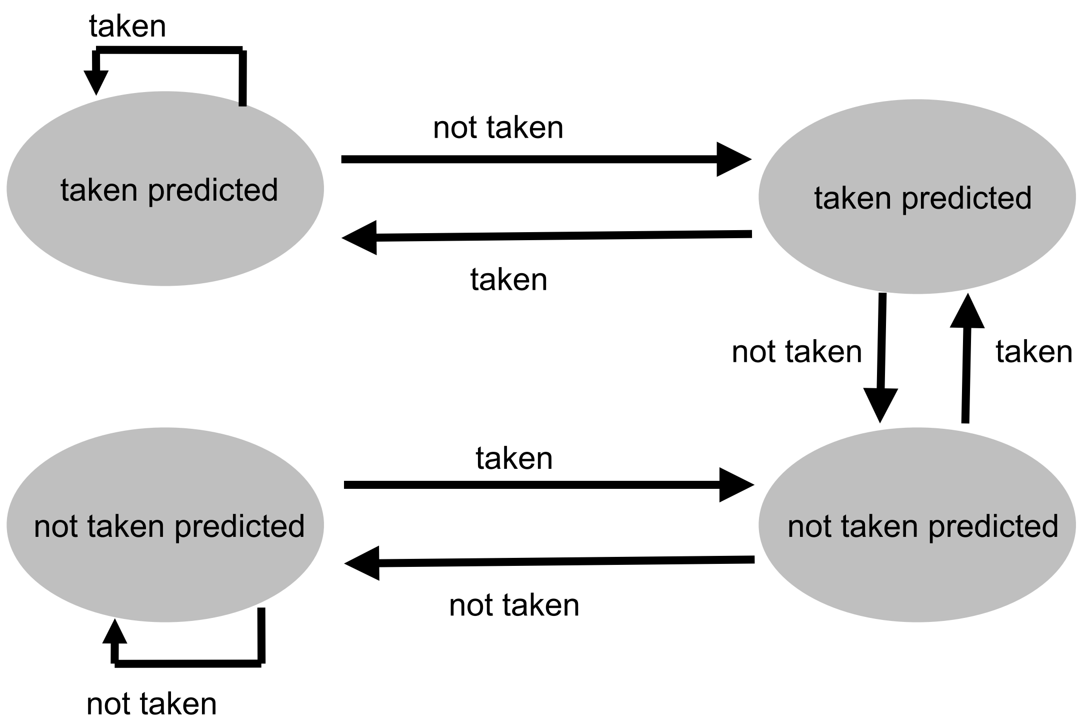
Dynamic branch prediction

Branch prediction scheme (2 Bit-scheme)

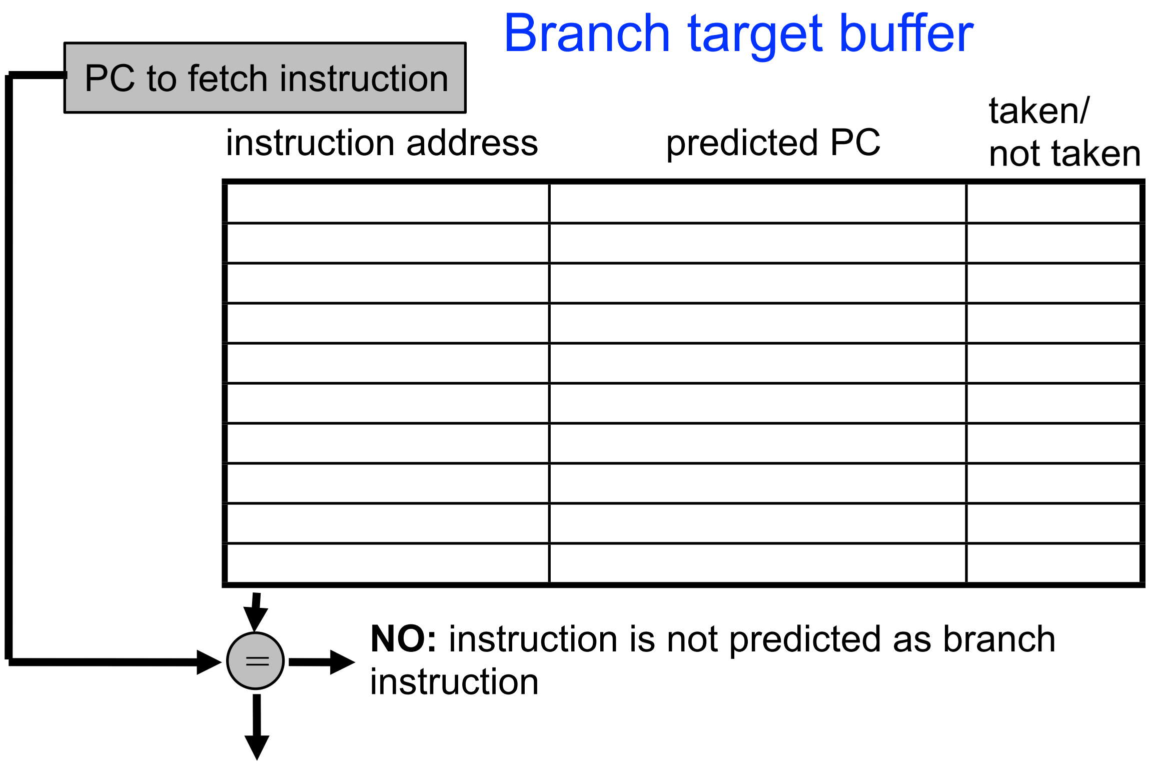


Dynamic branch prediction

Alternative branch prediction scheme (2 Bit-scheme)

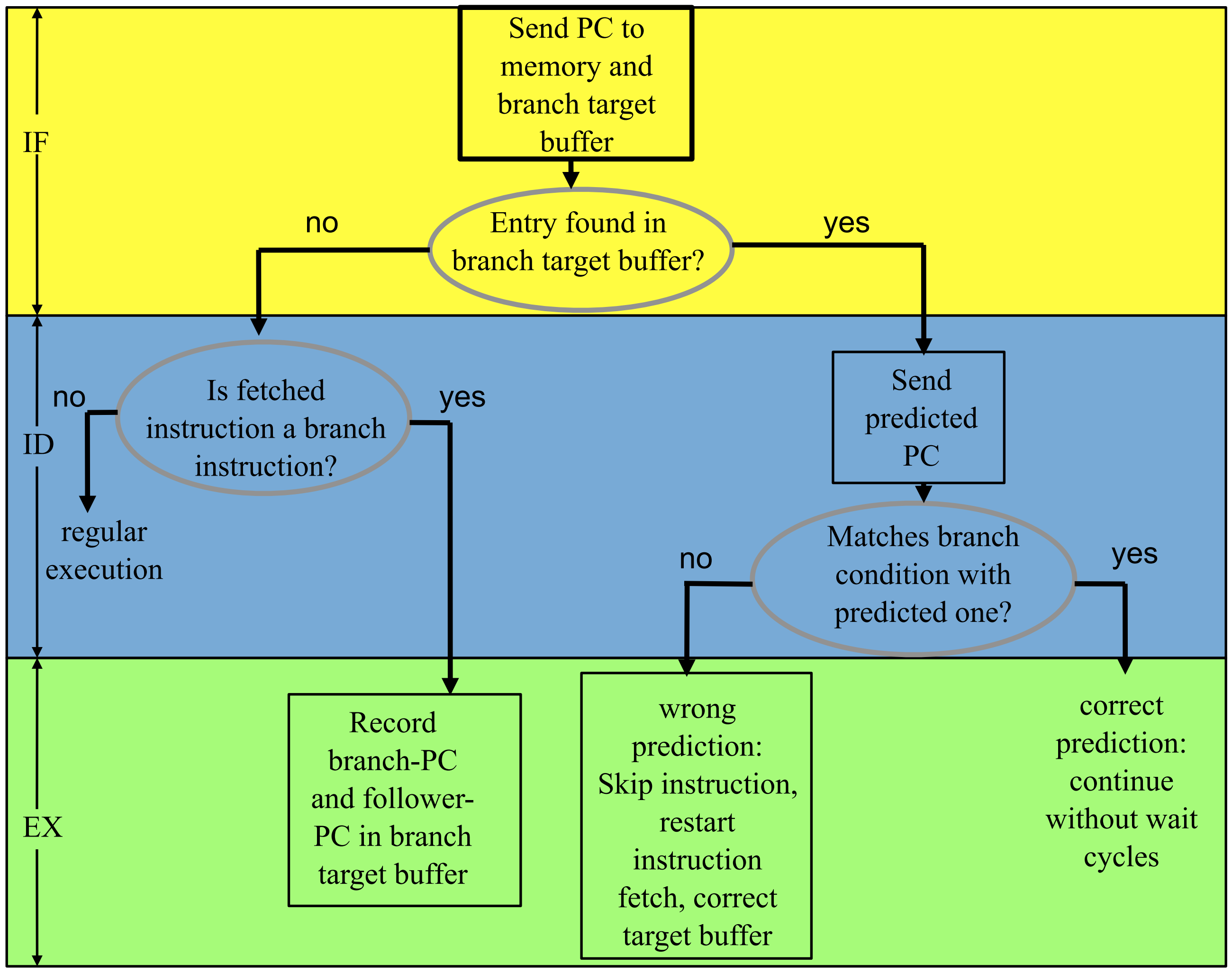


Dynamic branch prediction



NO: instruction is not predicted as branch instruction

YES: instruction is a branch and the predicted PC will be used as next PC


IF
ID
EX
Send PC to memory and branch target buffer
Entry found in branch target buffer?
no
yes
Is fetched instruction a branch instruction?
no
yes
regular execution
Send predicted PC
Matches branch condition with predicted one?
no
yes
Record branch-PC and follower-PC in branch target buffer
wrong prediction: Skip instruction, restart instruction fetch, correct target buffer
correct prediction: continue without wait cycles

Dynamic branch prediction

Interrupts/Exceptions

- Are treated as a special kind of branches
- No prediction possible
- In control handled as mispredicted branches (similar signaling)
- Sometimes all stages need to be flushed - PC must be reset not to last instruction fetched but to some instructions earlier!
- If exception is caused by instruction: PC set to instruction after that

Dynamic branch prediction

Speculation

- Allows compiler or processor to “guess” about the properties of an instruction - so following instructions can start earlier
- Recovery mechanisms needed if speculation was wrong
- HW: processor buffers speculative results in shadow registers
- can lead to exceptions (e.g. load with wrong address)

Task

Assume the following code snippet, the initial value of r3 = r2 + 400:

```
loop:  lw     r1, 0(r2)
       addi   r1, r1, #1
       sw     0(r2), r1
       addi   r2, r2, #4
       sub    r4, r3, r2
       bnez   r4, loop
```

Consider the DLX-Integer-Pipeline for this lesson.

a) Show the timing diagram of the code sequence for the DLX pipeline **without forwarding** but with register write and read within the same clock cycle. **Branches lead to a 3 cycle stall**.

Task

Assume the following code snippet, the initial value of r3 = r2 + 396:

```
loop:  lw     r1, 0(r2)
       addi   r1, r1, #1
       sw     0(r2), r1
       addi   r2, r2, #4
       sub    r4, r3, r2
       bnez   r4, loop
```

b) Show the timing-diagram of the DLX-Pipeline **with Forwarding**-Hardware. Consider the branches **predicted** as “not taken”. How many cycles needs this loop execution?

c) Assume now a DLX-Pipeline with a **delayed slot** (one cycle) **and Forwarding**-Hardware. Reschedule the instructions in the loop and, if necessary, modify operands. Show the timing diagram and calculate the number of cycles for the loop.

6. Loop unrolling

Increased Instruction Level parallelism

Aim: Further performance gain by CPI-value smaller than one.

$\longrightarrow$ multiple instruction fetch in one cycle

Usage in pipeline needs usage of parallelisms between instructions:

- Detection of series of independent instructions

Overlap of multiple instructions and execution in one cycle requires additional instruction level parallelism. There exist some simple compiler techniques helping to create additional parallelism.

Increased Instruction Level parallelism

- Software−Pipelining

  Overlap of different iterations of one loop body in time
- Trace−Scheduling

  Scheduling on multiple branches, optimized on the most likely path
- Loop Unrolling

  Creation of long straightforward instruction series
- Code−Inlining

  replace e.g. subprogram calls by copying the subprogram code directly in the program series
- ...

Loop Unrolling

Loop Unrolling – Increase number of instructions in one loop cycle

That means multiple repeating of loop body, correction of loop body and scheduling of unrolled loop.

e.g.:

```
for (i=1; i<n+1; i++) y[i] = y[i] + a*x[i];
```

is unrolled triply:

```
nrest = mod(n, 4);// Prologue for the first elements
n1 = nrest + 1;
      ...   // Elements 0...(n1−1)
for (i=n1; i<n+1; i+=4){// triply unrolled loop
                        // -> unroll depth 4
      y[i] = y[i] + a*x[i];
      y[i+1] = y[i+1] + a*x[i+1];
      y[i+2] = y[i+2] + a*x[i+2];
      y[i+3] = y[i+3] + a*x[i+3];}
```

Loop Unrolling

Aim:

- Reduce number of calculation steps (fewer iterations −> fewer loop overhead)
- Generation of more instruction level parallelism by longer straightforward series per iteration

Necessary:

- Fixed number of loop iterations (e.g. no break statement)
- Prologue and/or Epilogue needed if unroll depth does not match loop count (loop_count *mod* unroll_depth != 0)

also called “Modulo Scheduling”

Loop Unrolling

Loop:

```
Loop:  ld   f0, 0(r1)       ;Load vector element
       addd  f4, f0, f2     ;Add scalar in f2
       sd   0(r1), f4       ;Store vector element
       sub  r1, r1, #8      ;Decrement pointer by 8 Byte
       bnez  r1, Loop       ;Branch if r1=0
```

-> Needs 9 cock cycles per iteration

Latency times of operations:

instruction generating the result	instruction accessing the result	Latency in clock cycles
FP-ALU-Operation	Further FP-ALU-Operation	3
FP-ALU-Operation	Store double	2
Load DP-FP	FP-ALU-Operation	1
Load DP-FP	Store double	0
Other	Combinations	0

Branches generate a delay slot. Branches are predicted as “not taken”.

Loop Unrolling

Scheduling of the loop:

```
Loop:
        ld      f0, 0(r1)
        addd    f4, f0, f2
        sub     r1, r1, #8
        bnez    r1, Loop        ;delayed branch
        sd      8(r1), f4       ;swapped with sub
```

-> Needs 6 clock cycles per iteration

The loop contains overhead (SUB, BNEZ and delayed slot).

Loop Unrolling

Loop unrolled triply (Assumption: r1 is multiple of 4), no scheduling:

```
Loop: ld     f0, 0(r1)
      addd   f4, f0, f2
      sd     0(r1), f4       ; sub and bnez deleted
      ld     f6, −8(r1)
      addd   f8, f6, f2
      sd     −8(r1), f8      ; sub and bnez deleted
      ld     f10, −16(r1)
      addd   f12, f10, f2
      sd     −16(r1), f12    ; sub and bnez deleted
      ld     f14, −24(r1)
      addd   f14, f2
      sd     −24(r1), f16
      sub    r1, r1, #32
      bnez   r1, Loop
```

–> needs 27 clock cycles for 4 iterations, that means
6,75 clock cycles per iteration

Loop Unrolling

Unrolled loop with scheduling:

```
Loop:  ld     f0, 0(r1)
       ld     f6, −8(r1)
       ld     f10, −16(r1)
       ld     f14, −24(r1)
       addd   f4, f0, f2
       addd   f8, f6, f2
       addd   f12, f10, f2
       addd   f16, f14, f2
       sd     0(r1), f4
       sd     −8(r1), f8
       sd     −16(r1), f12
       sub    r1, r1, #32      ;branch dependence
       bnez   r1,  Loop
       sd     −24(r1), f16 ;8 − 32 = −24
```

−> needs 14 clock cycles for 4 iterations, that means 3,5 clock cycles per iteration

Software Pipelining

Reorganization of loops so that every new iteration contains instructions from successive loop iterations of original code segment.

The scheduler nests instructions of different loop interations without unrolling the loop.

```
Loop: ld     f0, 0(r1)
      addd   f4, f0, f2
      sd     0(r1), f4
      sub    r1, r1, #8
      bnez   r1, Loop
```

Software−Pipelining ↓

Software Pipelining

```
        ld      f0, 0(r1)       ; Load M[n]
        addd    f4, f0, f2      ; Add to M[n]
        sd      0(r1), f4       ; Store M[n]
        subi    r1, r1, #16     ; Set loop−counter
    ; M[n] is the first stored element in the loop
    ; M[3] is the last stored element in the loop

Loop:   sd      16(r1), f4      ; Store M[i]
        addd    f4, f0, f2      ; Add to M[i−1]
        ld      f0, 0(r1)       ; Load M[i−2]
        subi    r1, r1, #8
        bnez    r1, Loop
out_of_loop:
        sd      16(r1), f4      ; Store M[2]
        addd    f4, f0, f2      ; Add to M[1]
        sd      8(r1), f4       ; Store M[1]
```

A superscalar DLX version

Machines, which fetch multiple independent instructions per clock cycle if they are suitable assigned by the compiler, are called superscalar machines.

HW can fetch a small number (2 to 4) independent instructions in a single clock cycle.

Machine checks dependencies: Are instructions in the queue dependent on each other or do not fulfill dedicated criterions only the first instruction in the line will be fetched.

A superscalar DLX version

Example: Superscalar DLX; 2 instructions per cycle − Integer/Load/Store/Branch − and FP−Operations as pair; requires 64 Bit for instruction fetch and decode

Instruction type								
FK−instr.	IF	ID	EX	MEM	WB			
GK−instr.	IF	ID	EX	MEM	WB			
FK		IF	ID	EX	MEM	WB		
GK		IF	ID	EX	MEM	WB		
FK			IF	ID	EX	MEM	WB	
GK			IF	ID	EX	MEM	WB	
FK				IF	ID	EX	MEM	WB
GK				IF	ID	EX	MEM	WB

$\longrightarrow$ Increases in substance the fetch rate of FP instructions. There are either pipeline−FP−units or multiple independent units necessary!

A superscalar DLX version

Compiler-Scheduling-Methods are important for the utilisation of the available effective parallelism in a superscalar pipeline.

Example: execution of the unrolled loop on a superscalar DLX:

To allow a scheduling without any delay five copies of the loop need to be unrolled.

	Int−instruction		FP-instruction	Cycle
Loop:	ld	f0, 0(r1)		1
	ld	f6, −8(r1)		2
	ld	f10, −16(r1)	addd f4, f0, f2	3
	ld	f14, −24(r1)	addd f8, f6, f2	4
	ld	f18, −32(r1)	addd f12, f10, f2	5
	sd	0(r1), f4	addd f16, f14, f2	6
	sd	−8(r1), f8	addd f20, f18, f2	7
	sd	−16(r1), f12		8
	sd	−24(r1), f16		9
	sub	r1, r1, #40		10
	bnez	r1, Loop		11
	sd	8(r1), f20		12

−> 2,4 cock cycles per iteration

Task

Remember the float_test program:

```
      ld     f2,a
      add  r1,r0,xtop
loop: ld   f0,0(r1)
      multd   f4,f0,f2
      sd   0(r1),f4
      sub  r1,r1,#8
      bnez    r1,loop
      nop
      trap  #0
```

Assume a standard DLX pipeline with the following (default) latencies:
FP-load: 1 cycle; FP-Addition/Subtraction: 2 cycles; FP-Multiplication: 5 cycles

Unroll this loop triply and schedule it for the standard DLX pipeline. Try to rearrange the code to get a maximum of execution power (minimized execution time).

7. Very Long Instruction Word

The VLIW Method

Superscalar: 2 instructions per clock cycle are issued, perhaps 3, 4 or 5. Difficulty: Detection, how many instructions and in which order can be issued simultaneously, how they can be loaded and which dependencies exist.

Alternative: LIW− resp. VLIW (Very Long Instruction Word)−Architecture

VLIWs use multiple independent functional units. Instead of trying to issue multiple independent instructions a VLIW architecture combines these operations into a single very long instruction.

Detection and generation of parallelism to use the available FUs must be done by the compiler!

The VLIW Method

Limits/Costs (Why not issue 50 operations per clock cycle?)

- limited parallelism
- limited hardware ressources (memory− and registerfile bandwidth −> many read- and write ports necessary)
- Difficulties in using data caches
- limited code length/instruction word length

The VLIW Method

Allocation of unrolled loop to VLIW−instruction of a VLIW−DLX, which can handle two memory accesses, two FP - and two integer operations in one instruction word.
(To avoid any wait cycle the loop must be unrolled sixfold.)

Mem access 1	Mem access 2	FP−Operation 1	FP−Operation 2	Integer/Branch
ld f0, 0(r1)	ld f6, −8(r1)			
ld f10, −16(r1)	ld f14, −24(r1)			
ld f18, −32(r1)	ld f22, −40(r1)	addd f4, f0, f2	addd f8, f6, f2	
ld f26, −48(r1)		addd f12, f10, f2	addd f16, f14, f2	
		addd f20, f18, f2	addd f24, f22, f2	
sd 0(r1), f4	sd −8(r1), f8	addd f28, f26, f2		
sd −16(r1), f12	sd −24(r1), f16			
sd −32(r1), f20	sd −40(r1), f24			sub r1, r1, #48
sd −0(r1), f28				bnez r1, Loop

−> 1,29 clock cycles per iteration

The VLIW Method

The effectiveness, determined by the number of used operation fields, is here about

60 %

Issue rate: 23 operations in 9 clock cycles (2,5 operations per cycle). To reach this issue rate a much higher number of registers is necessary than DLX would use for this loop in normal case.

Vector processors

Why should we use vector processors?

Pipeline performance is limited by two factors:

- Clock cycle time − can be reduced by long pipelines.
  But increasing pipeline depth increases dependencies which leads to a higher CPI value.

- Instruction fetch and instruction decode rate (Flynn's bottleneck) forbids fetching and issuing of more than a fistful instructions per clock cycle.

Vector processors

Vector machines have operations of higher level working with vectors.

Properties of these operations:

- Computation of every result is independent of previous results and allows high pipeline depth without data hazards
  (compiler/programmer decides where to use vector instructions).
- Single vector instruction specifies a large amount of work - reduces instruction flow (reduce limitation of Flynn's bottleneck)
- Memory accesses of vector instructions have a known access pattern. An access is initiated for the complete vector, that means the memory latency does not carry that much weight.
- Vector instructions replace loops; the behavior is predefined -> no loop branch, no control hazards.

Vector instruction can be handled faster than a series of scalar operations with the same amount of data elements.

Vector processors

Example DLXV:

- 8 Vector registers, each with 64 double words (= 512 Byte)
- at least two read ports and two write ports,
- 5 pipeline-implemented vector-FUs,
- vector-load/store unit
- scalar register file,
- special registers VLR (vector length register)
  and VM (vector mask register)

Vector processors

Problem Y = a*X + Y, assumption vector length = 64

Standard DLX−code:

```
        ld    f0,a
        addi  f4,rx,#512          ; Load from last address
loop:
        ld    f2, 0(rx)           ; Load X(i)
        multd f2, f0, f2          ; a*X(i)
        ld    f4, 0(ry)           ; Load Y(i)
        addd  f4, f2,    f4       ; a*X(i) + Y(i)
        sd    f4, 0(ry)           ; Store to Y(i)
        addi  rx, ry,    #8       ; Increment index for X
        addi  ry, ry,    #8       ; Increment indexes for Y
        sub   r20,r4,    rx       ; Decrement loop index
        bnez  r20,loop            ; Test on last loop iteration
```

Almost 600 instructions to execute!

Vector processors

Problem Y = a*X + Y, assumption vector length = 64

DLXV−code:

```
ld          f0, a       ; Load scalar a
lv          v1, rx      ; Load vector X
multsv      v2, f0, v1  ; Vector−Scalar−Multiply
lv          v3, ry      ; Load Vector Y
addv        v4, v2, v3  ; Vektor add
sv          ry, v4      ; Store result vector
```

Every vector instruction works with all vector elements independently
- no pipeline stalls.
But keep in mind: **startup-time** and **initiation rate**!

Only 6 instructions to execute!

Vector processors

Startup time of a vector operation:

- Time until the first result is available
- determined by the pipeline depth of the executing FU
- corresponds to latency of that pipeline

Initiation rate:

- Time per result if FU-pipeline is running
- must be equal to finishing rate

For the computation of a vector with length n:

Vector running time = Startup time + n * Initiation rate

Vector processors

Example: Vector length n = 64 elements
Startup period 10 cycles
Initiation rate 1 cycle

How many clock cycles per vector element?

$$\frac{\text{Cycles}}{\text{Vector element}} = \frac{\text{Vector running time}}{\text{Vector length}}$$

$$= \frac{\text{startup time} + n * \text{Initiation rate}}{n}$$

$$= (10 + 64*1) / 64 = 1{,}16 \text{ cycles per vector element}$$

Vector processors

Register−Vector operations:

- Startup time depends on pipeline depth
- Additional starting delay if there are dependencies between operands!
- Initiation rate determined how often a unit can fetch new operands. If fully pipelined then one per cycle.

Load/Store−Vector operations:

- Startup time for load instruction to fetch first doubleword from memory to register
- No startup time for store instruction
- If load must wait for store completion than load instruction has an additional startup time
- To reach an initiation rate of one doubleword per cycle for load/store the memory system must contain multiple banks accessible in parallel

Vector processors

Vector length control:

Length of vector register determines the maximum length executable in a vector operation (MVL)
- but does rarely match with the real vector length!

Vector length register (VLR) saves length of actual vector, vector of arbitrary lenght must be segmented before its processing:

Number of equal segments: **j = n div MVL**

Length of rest element: **m = n modulo MVL**

Except the rest segment all segments have the maximum vector length MVL − use the full performance of the vector machine

Vector processors

Vector stride:

Order of vector’s neighbour elements in memory is not always sequential.

Example: A(i,j+1) is colocated in most languages with A(i,j), but A(i+1,j) not with A(i,j).

Distance of separated but in a vector collected elements defined as stride

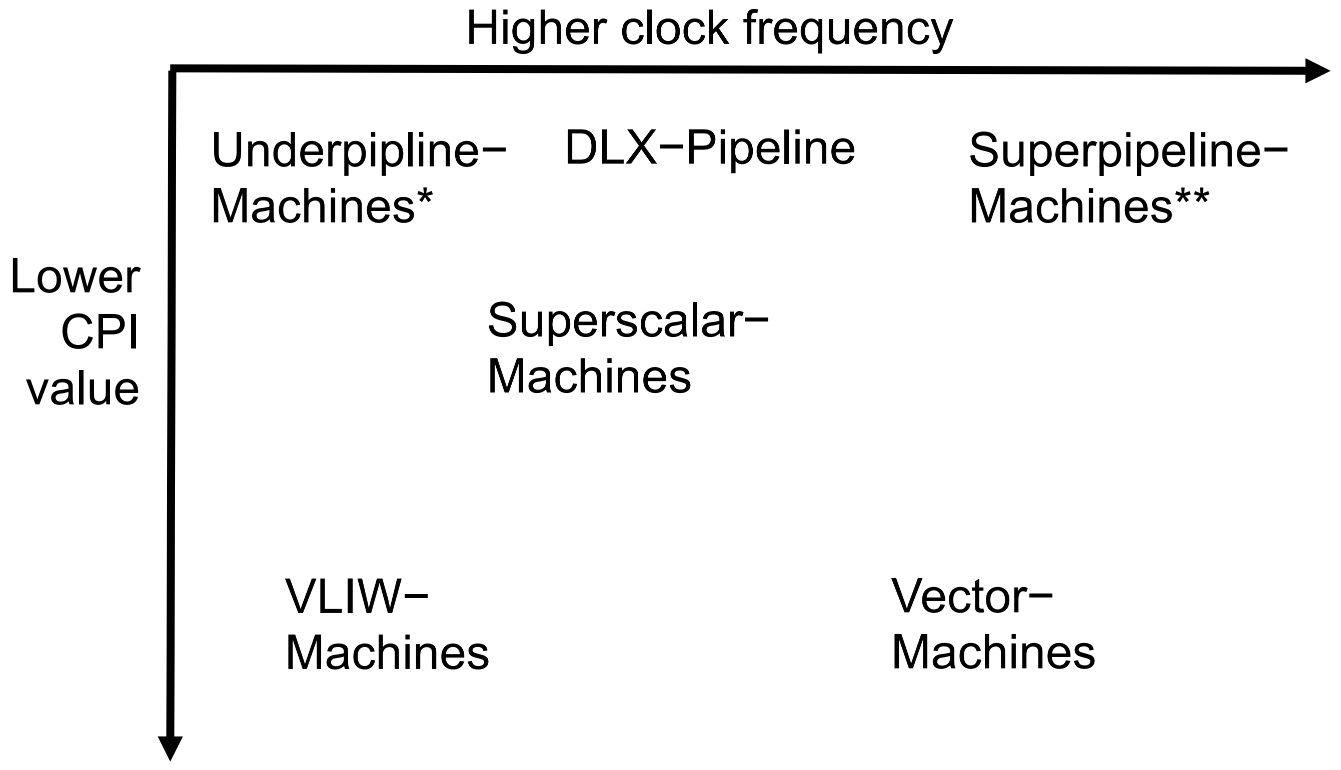
Stride is calculated dynamically and stored in a general purpose register.

Vector processors

Compiler technique for Vector machines

To use vector machines efficiently a vector compiler should manage the following topics:

- Detect dependencies between operands to
  - prevent data hazards
  - decide whether a loop could be vectorised
- split long vectors in segments and allocate the vector registers properly



* Machines which assemble several pipeline stages resulting in a slower clock

** Machines with a higher clock frequency and longer pipeline, pipelining of all functional units

8. Case studies

Example Intel Processors

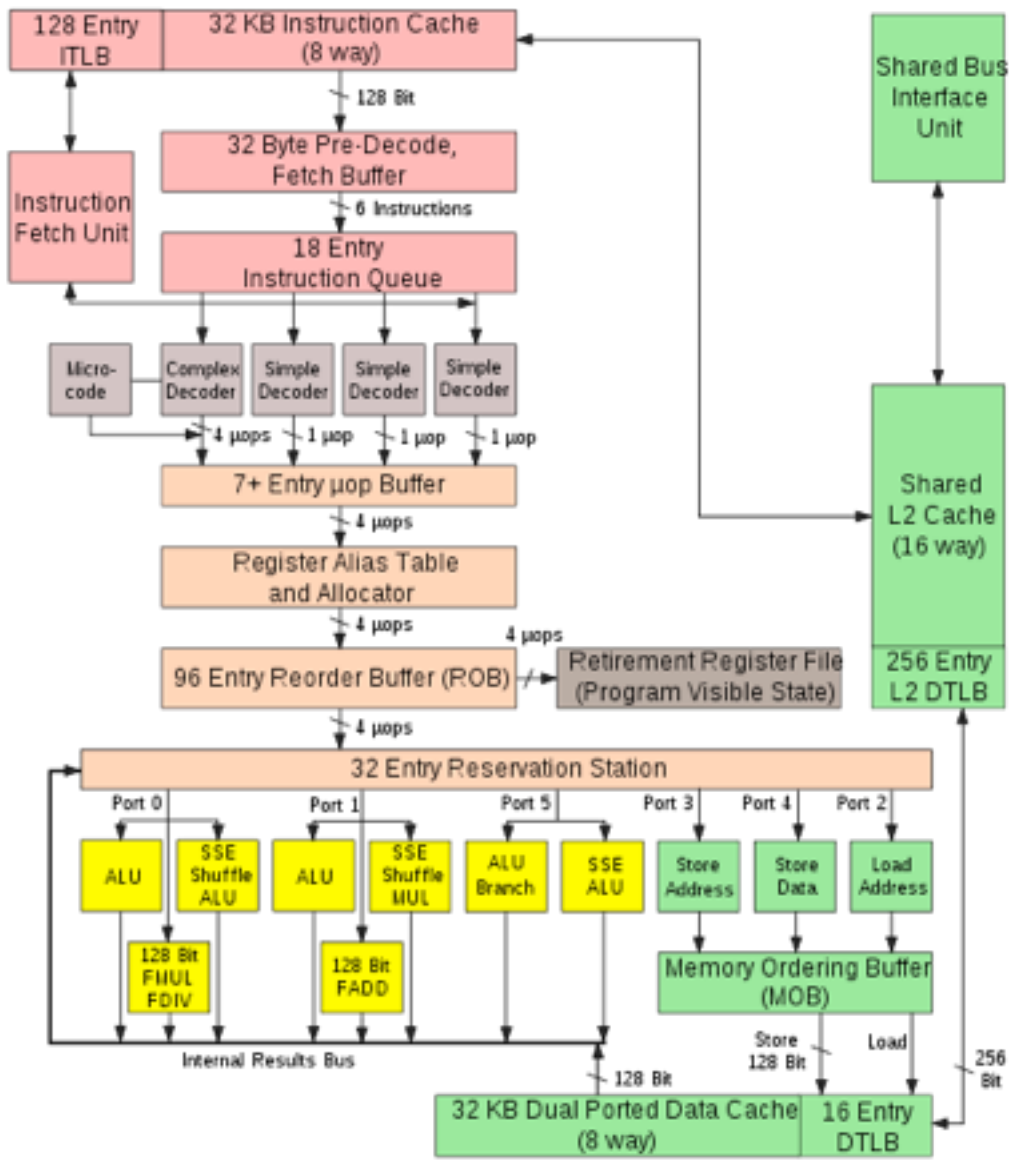
µP	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order	Power
486	1989	25 MHz	5	1	No	5 W
Pentium	1993	66 MHz	5	2	No	10 W
Pentium Pro	1997	200 MHz	10	3	Yes	29 W
P4 Willamette	2001	2000 MHz	22	3	Yes	75 W
P4 Prescott	2004	3600 MHz	31	3	Yes	103 W
Intel Core	2006	2930 MHz	14	4	Yes	75 W

Example Intel Pentium 4 Willamette

So called NetBurst−Architecture: Hyper−Pipeline with 20 stages

- allows high clock frequencies
- can simulataneously execute up to 128 Micro−Operations (Micro−Op: x86 instructions are separated into smaller RISC instructions)
- Implemented Rapid Execution Engine, so ALUs are clocked with doubled CPU frequency (e.g. integer instructions at P4 1,6 GHz with 3,2 GHz)
- can move 4 data words with 64 bit width in one cycle

Example Intel Core



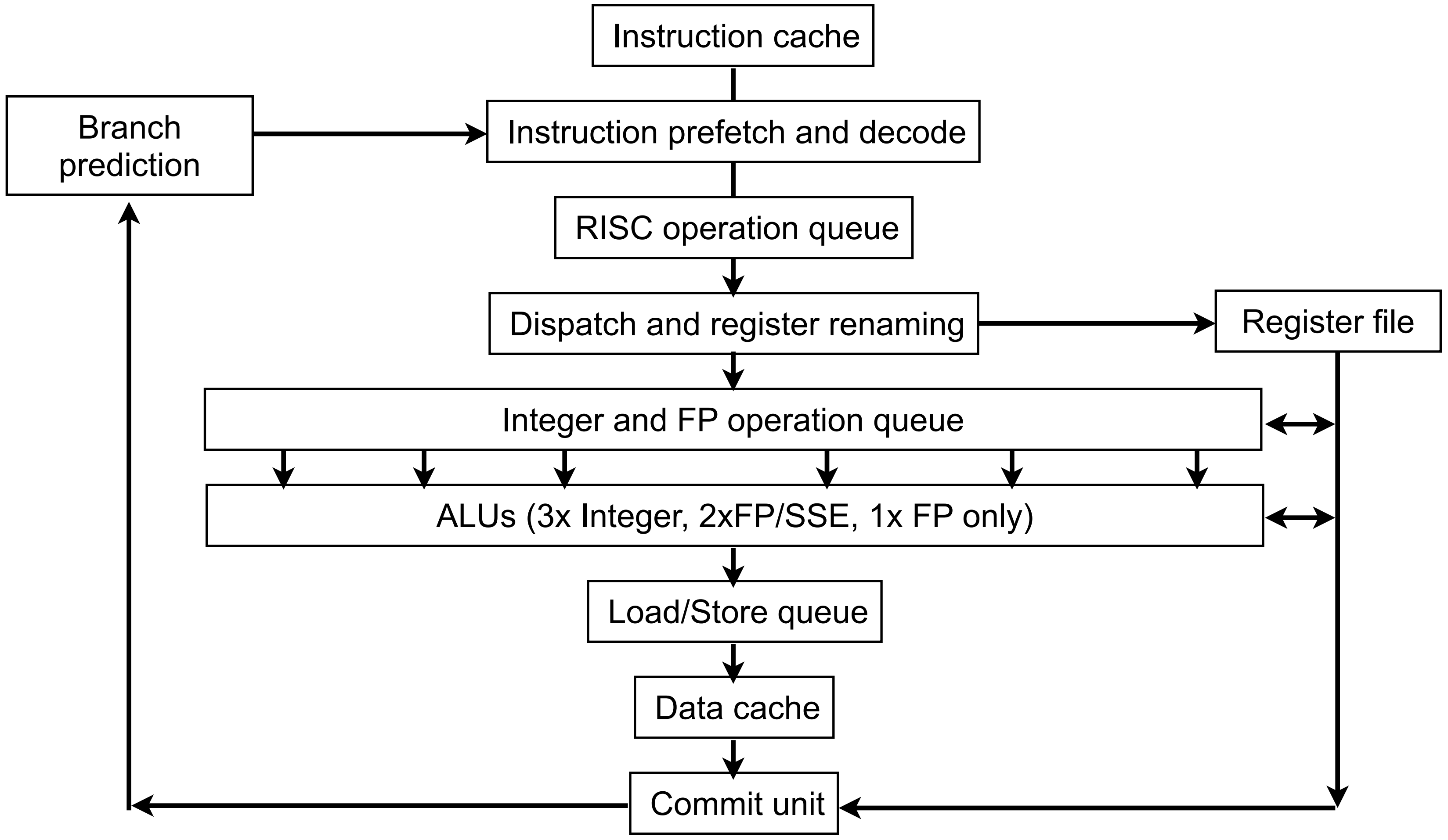
Intel Core 2 Architecture

Example AMD Opteron X4 (Barcelona)

- ❖ Rops - x86 instructions are separated into smaller RISC operations
- ❖ Queues for up to 106 operations (24 integer, 36 FP, 44 Load/Store)
- ❖ Extensive bypass network
- ❖ 12 stage integer RISC pipeline, 17 stages FP pipeline
- ❖ Explicit register renaming (72 physical for 16 logical registers)
- ❖ Execution rate of 3 RISC operations per clock cycle

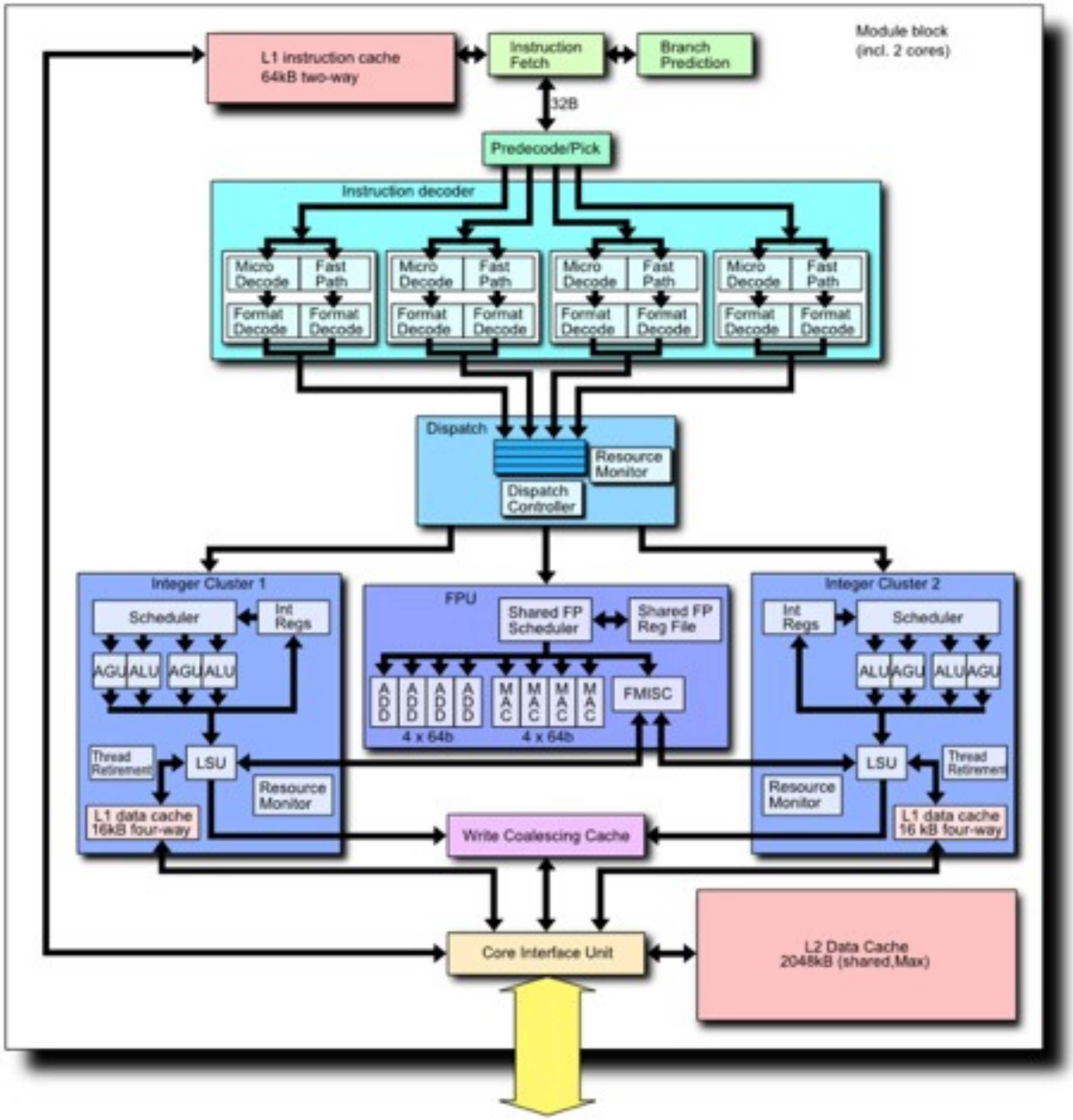
Source: Patterson, Hennessy: Computer Organization and Design

Example AMD Opteron X4 (Barcelona)

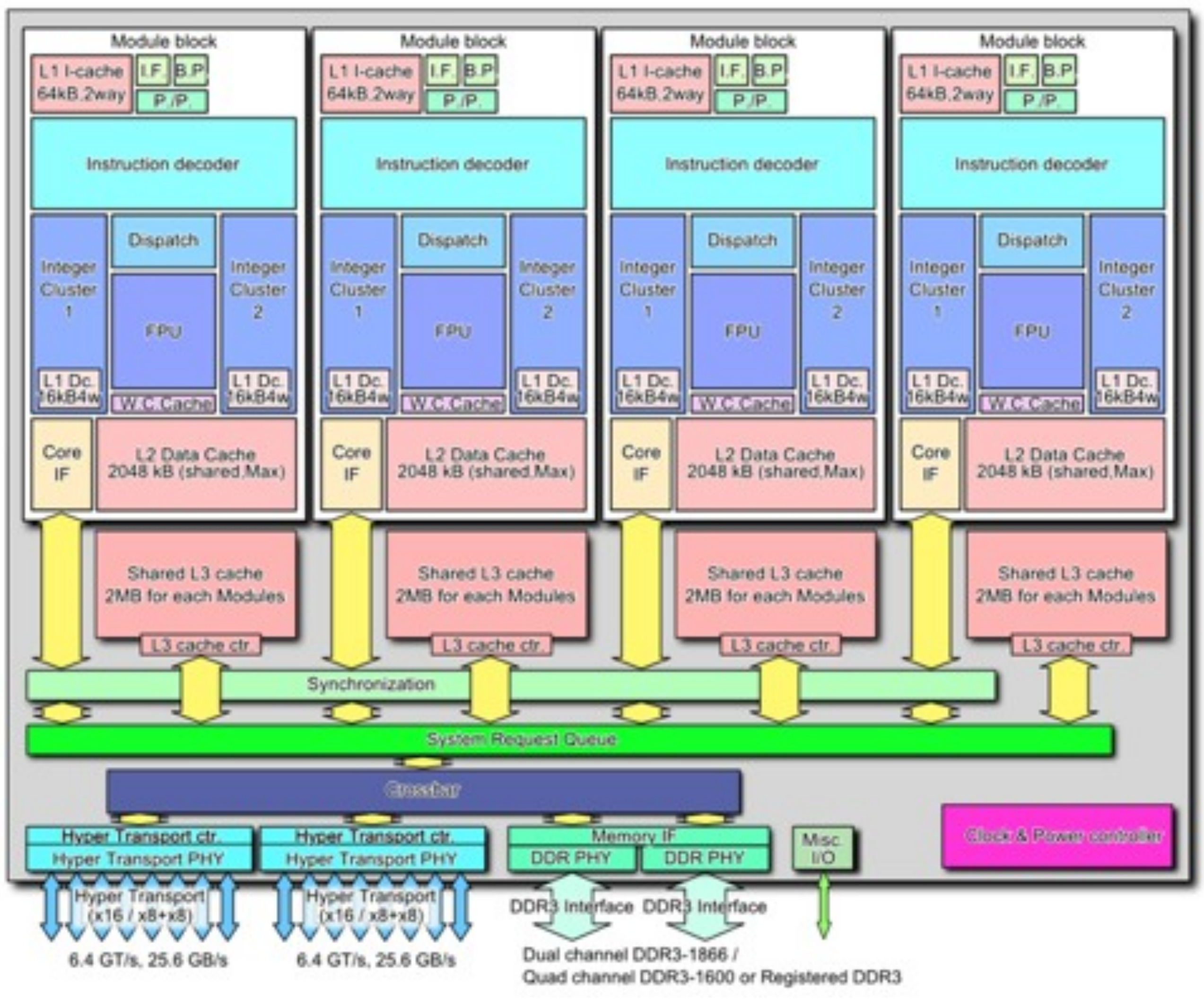

Instruction cache
Branch prediction
Instruction prefetch and decode
RISC operation queue
Dispatch and register renaming
Register file
Integer and FP operation queue
ALUs (3x Integer, 2xFP/SSE, 1x FP only)
Load/Store queue
Data cache
Commit unit

Source: Patterson, Hennessy: Computer Organization and Design

Example AMD Bulldozer Architecture

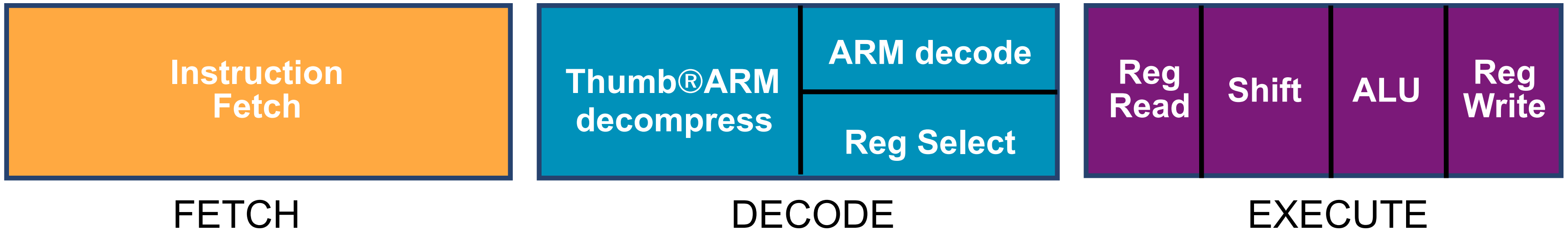


Example AMD Bulldozer Architecture

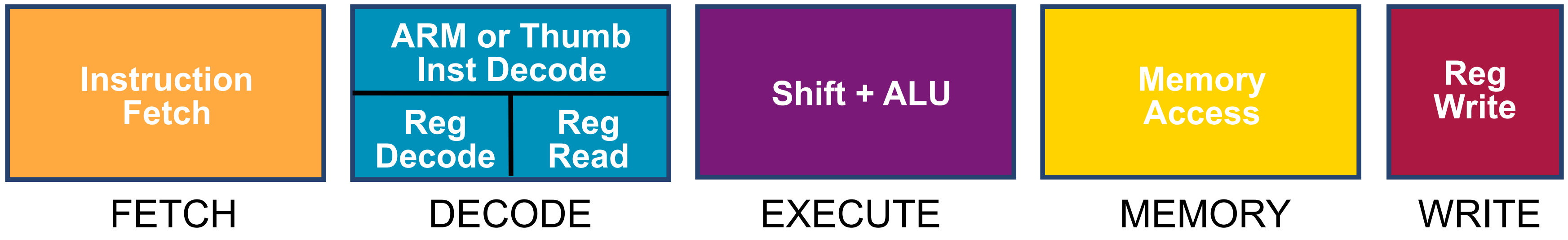


ARM Core

ARM7TDMI



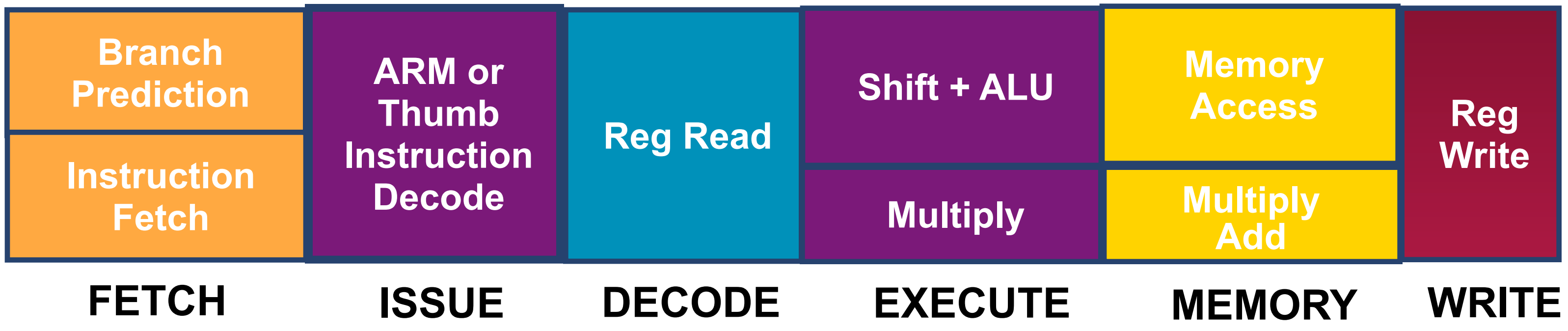
ARM9TDMI



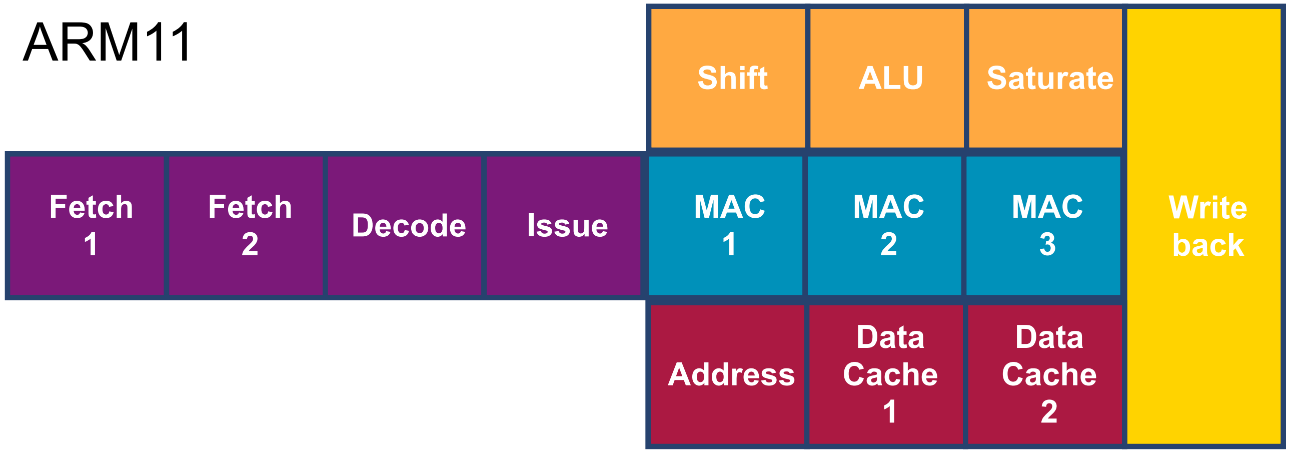
Source: ARM teaching material

ARM10 vs. ARM11 Pipelines

ARM10



ARM11



Source: ARM teaching material

Example ARM Cortex A15

- Long pipeline (15 - 24 stages) for ARM cores
  - instruction fetch and decode needs already 12 stages
- Out-of-Order instruction execution
- Supports eight cores with cache-coherence
- Two buffers for branch prediction
- Register renaming (128 physical regs for 16 logical regs)
- HW support for OS virtualization
- No multithreading

Source: Elektronik 17/2011

Example ARM Cortex A15

Out-of-Order instruction execution

- Last decode stage feeds up to 3 instructions into execution queues
- 2 Integer and 1 mult/div queues: 8 entries
- 2 FP and 2 load/store queues: 16 entries
- Blocked instructions can be overtaken by others (except store operations)
- speculative instruction execution

Source: Elektronik 17/2011

Example ARM Cortex A15

Branch prediction

- Special branch unit in pipeline (parallel to ALUs)
- Branch History Table (BHT) with 2048 entries contains state of last branch instructions
- Branch Target Buffer (BTB) with 256 entries
- Global BHT with 8192 entries
- MicroBTB with 64 entries as very fast predictor (can be overridden by main BTB)
- Loop buffer for small loops with up to 32 instructions for power reduction

Source: Elektronik 17/2011

9. AL-components

Arithmetic-Logic components

No script for this chapter available

Several components like adders and multipliers will be presented at the black board.

Please visit the according exercise for details.