Rechnerorganisation – Advanced Arch. & ARM

Prof. Sarah Harris
Fachgebiet Eingebettete Systeme und ihre Anwendungen (ESA)
Fachbereich Informatik

TECHNISCHE UNIVERSITÄT DARMSTADT

SS 16

Logistics

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Today:** Advanced Microarchitecture
  ARM Architecture

- **18.07:** Klausur, 10 Uhr – 11:30 Uhr
  - wo die stattfindet, wird in der Woche davor durch Moodle bekanntgegeben

Advanced Microarchitecture

TECHNISCHE UNIVERSITÄT DARMSTADT

- Deep Pipelining
- Branch Prediction
- Superscalar Processors
- Out of Order Processors
- Register Renaming
- SIMD
- Multithreading
- Multiprocessors

Deep Pipelining

TECHNISCHE UNIVERSITÄT DARMSTADT

- 10-20 stages typical
- Number of stages limited by:
  - Pipeline hazards
  - Sequencing overhead
  - Power
  - Cost

Branch Prediction

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ideal pipelined processor: CPI = 1
- Branch misprediction increases CPI
- **Static branch prediction:**
  - Check direction of branch (forward or backward)
  - If backward, predict taken
  - Else, predict not taken
- **Dynamic branch prediction:**
  - Keep history of last (several hundred) branches in *branch target buffer*, record:
    - Branch destination
    - Whether branch was taken

Branch Prediction Example

TECHNISCHE UNIVERSITÄT DARMSTADT

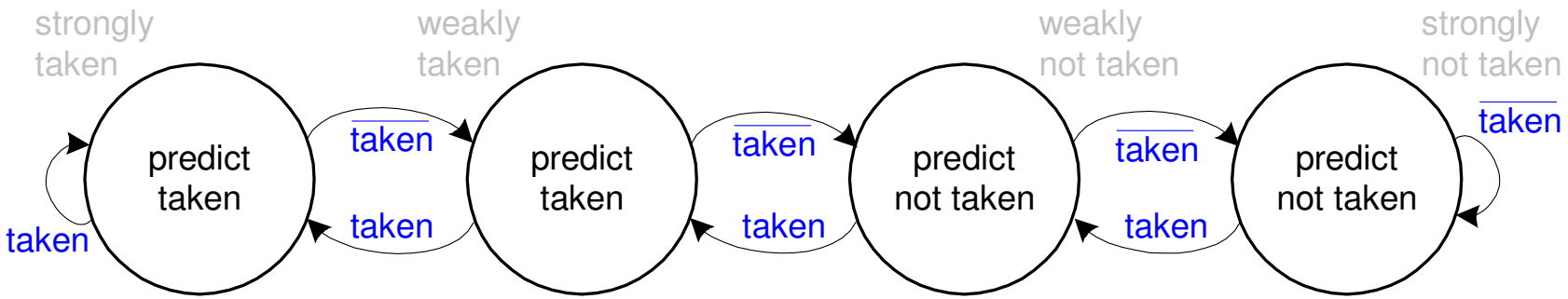
```
    add  $s1, $0, $0       # sum = 0
    add  $s0, $0, $0       # i   = 0
    addi $t0, $0, 10       # $t0 = 10
for:
    beq  $s0, $t0, done    # if i == 10, branch
    add  $s1, $s1, $s0     # sum = sum + i
    addi $s0, $s0, 1       # increment i
    j    for
done:
```

1-Bit Branch Predictor

TECHNISCHE UNIVERSITÄT DARMSTADT

- Remembers whether branch was taken the last time and does the same thing
- Mispredicts first and last branch of loop

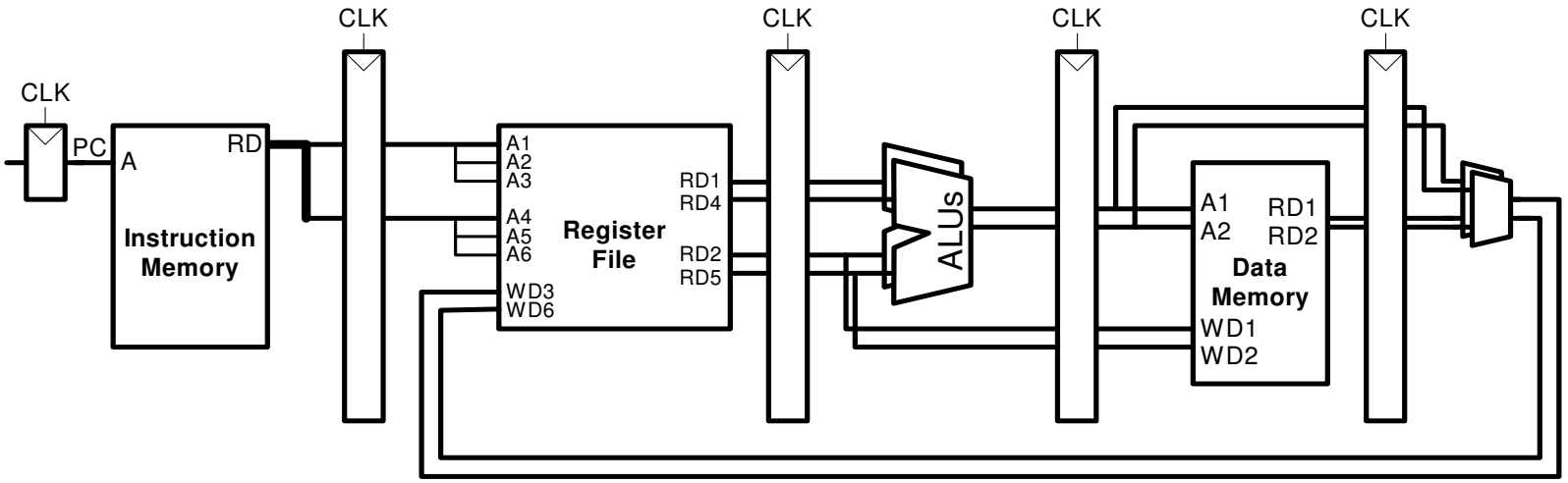
2-Bit Branch Predictor



Only mispredicts last branch of loop

Superscalar

- Multiple copies of datapath execute multiple instructions at once
- Dependencies make it tricky to issue multiple instructions at once

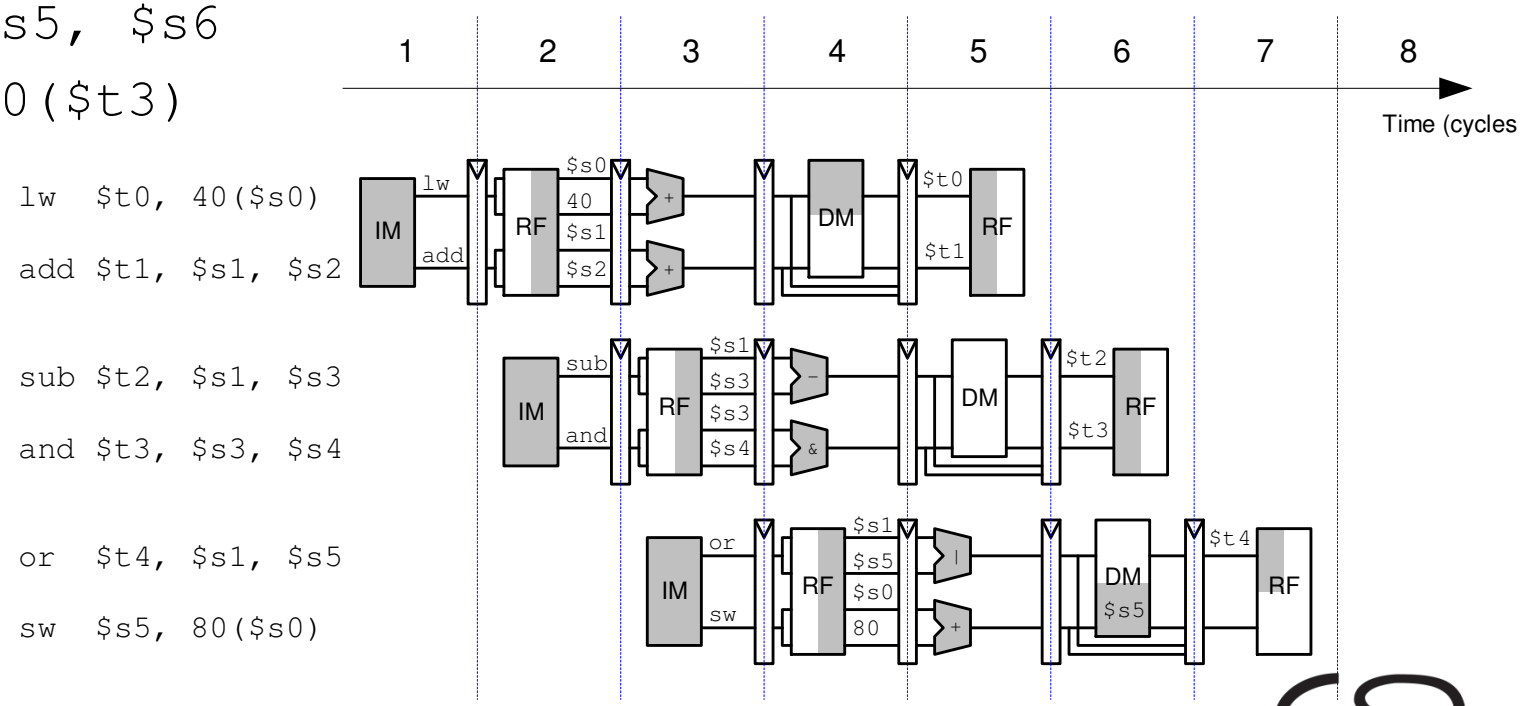


Superscalar Example

```
lw   $t0, 40($s0)
add  $t1, $t0, $s1
sub  $t0, $s2, $s3
and  $t2, $s4, $t0
or   $t3, $s5, $s6
sw   $s7, 80($t3)
```

Ideal IPC: 2

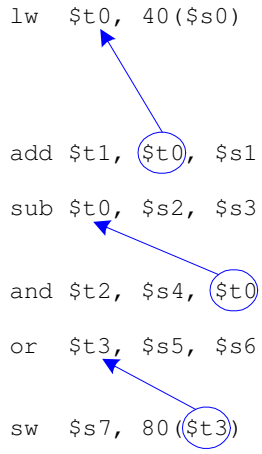
Actual IPC: 2



Superscalar with Dependencies

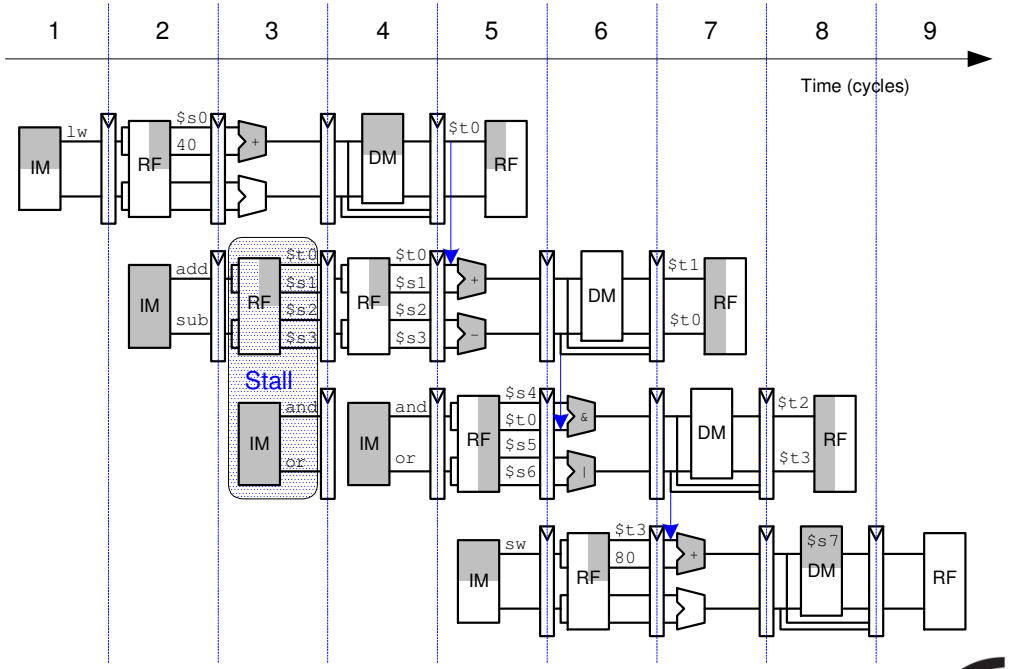
TECHNISCHE UNIVERSITÄT DARMSTADT

```
lw   $t0, 40($s0)
add  $t1, $t0, $s1
sub  $t0, $s2, $s3
and  $t2, $s4, $t0
or   $t3, $s5, $s6
sw   $s7, 80($t3)
```



Ideal IPC: **2**

Actual IPC: **6/5 = 1.2**



Embedded Systems & Applications

Out of Order Processor

TECHNISCHE UNIVERSITÄT DARMSTADT

- Looks ahead across multiple instructions
- Issues as many instructions as possible at once
- Issues instructions out of order (as long as no dependencies)
- **Dependencies:**
  - **RAW** (read after write): one instruction writes, later instruction reads a register
  - **WAR** (write after read): one instruction reads, later instruction writes a register
  - **WAW** (write after write): one instruction writes, later instruction writes a register

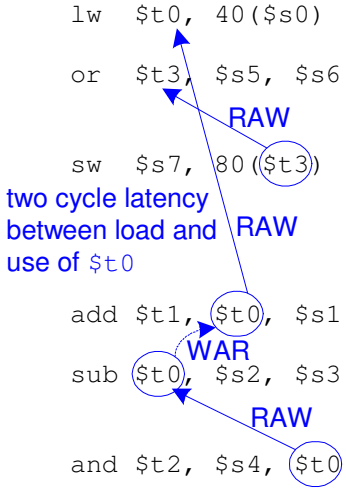
Out of Order Processor

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Instruction level parallelism (ILP):** number of instruction that can be issued simultaneously (average < 3)
- **Scoreboard:** table that keeps track of:
  - Instructions waiting to issue
  - Available functional units
  - Dependencies

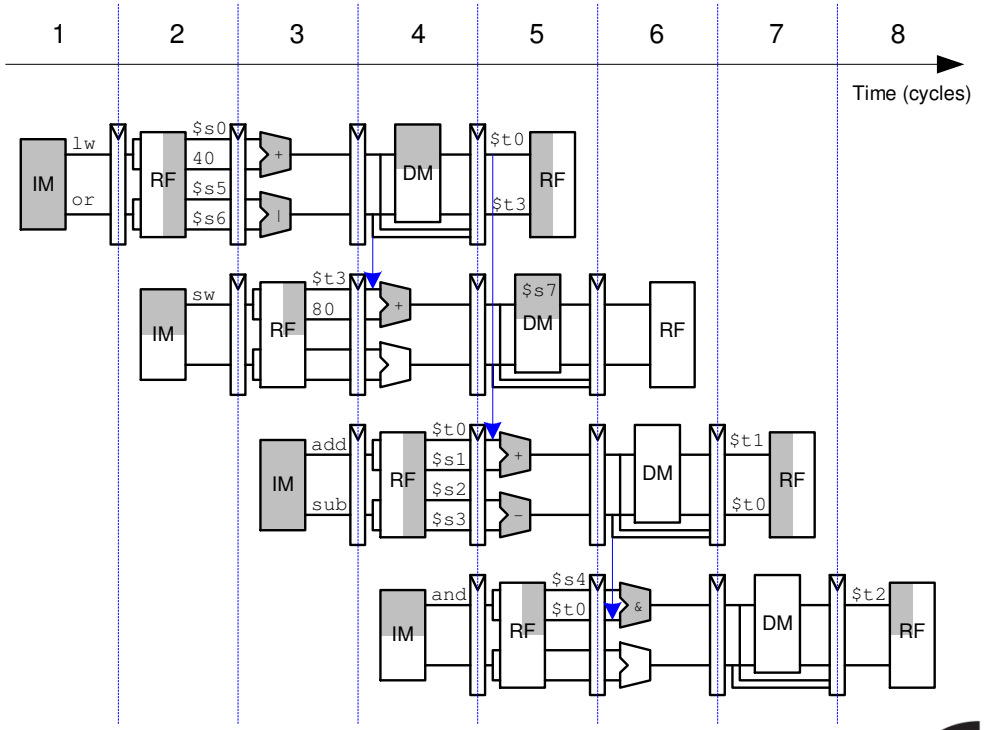
Out of Order Processor Example

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```



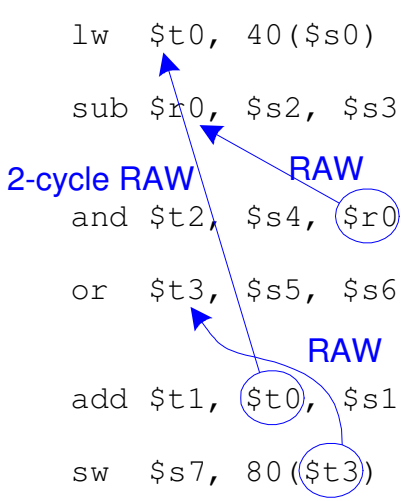
Ideal IPC: 2

Actual IPC: 6/4 = 1.5



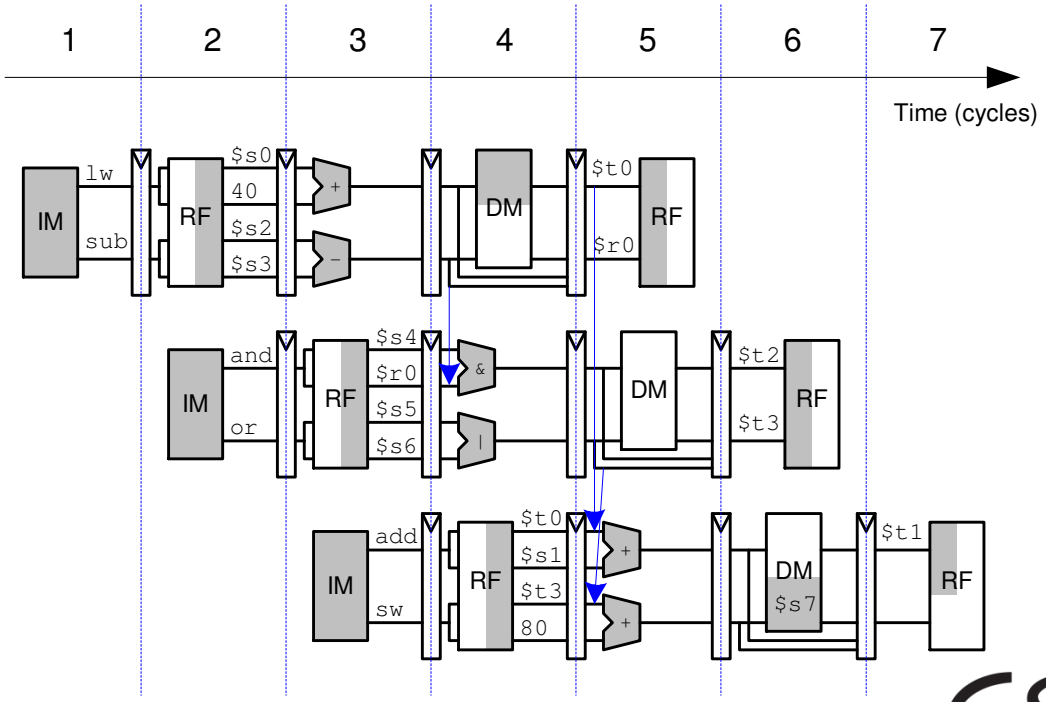
Register Renaming

```
lw   $t0, 40($s0)
add  $t1, $t0, $s1
sub  $t0, $s2, $s3
and  $t2, $s4, $t0
or   $t3, $s5, $s6
sw   $s7, 80($t3)
```



Ideal IPC: 2

Actual IPC: 6/3 = 2

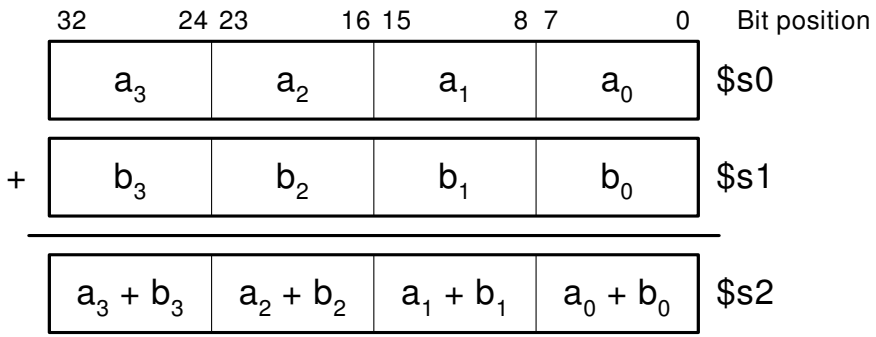


SIMD

TECHNISCHE UNIVERSITÄT DARMSTADT

- Single Instruction Multiple Data (SIMD)
  - Single instruction acts on multiple pieces of data at once
  - Common application: graphics
  - Perform short arithmetic operations (also called *packed arithmetic*)
- For example, add four 8-bit elements

```
padd8 $s2, $s0, $s1
```



Embedded Systems & Applications

Advanced Architecture Techniques

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Multithreading**
  - Wordprocessor: thread for typing, spell checking, printing
- **Multiprocessors**
  - Multiple processors (cores) on a single chip

Threading: Definitions

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Process:** program running on a computer
  - Multiple processes can run at once: e.g., surfing Web, playing music, writing a paper
- **Thread:** part of a program
  - Each process has multiple threads: e.g., a word processor may have threads for typing, spell checking, printing

Threads in Conventional Processor

TECHNISCHE UNIVERSITÄT DARMSTADT

- One thread runs at once
- When one thread stalls (for example, waiting for memory):
  - Architectural state of that thread stored
  - Architectural state of waiting thread loaded into processor and it runs
  - Called **context switching**
- Appears to user like all threads running simultaneously

Multithreading

TECHNISCHE UNIVERSITÄT DARMSTADT

- Multiple copies of architectural state
- Multiple threads **active** at once:
  - When one thread stalls, another runs immediately
  - If one thread can't keep all execution units busy, another thread can use them
- Does not increase instruction-level parallelism (ILP) of single thread, but increases throughput

Intel calls this "hyperthreading"

Multiprocessors

TECHNISCHE UNIVERSITÄT DARMSTADT

- Multiple processors (cores) with a method of communication between them
- Types:
  - **Homogeneous**: multiple cores with shared memory
  - **Heterogeneous:** separate cores for different tasks (for example, DSP and CPU in cell phone)
  - **Clusters:** each core has own memory system

Other Resources

TECHNISCHE UNIVERSITÄT DARMSTADT

- Patterson & Hennessy's: ***Computer Architecture: A Quantitative Approach***
- **Conferences:**
  - www.cs.wisc.edu/~arch/www/
  - ISCA (International Symposium on Computer Architecture)
  - HPCA (International Symposium on High Performance Computer Architecture)

ARM Architecture

- Developed in the 1980's by Advanced RISC Machines – now called ARM Holdings
- Nearly 10 billion ARM processors sold/year
- Almost all cell phones and tablets have multiple ARM processors
- Over 75% of humans use products with an ARM processor
- Used in servers, cameras, robots, cars, pinball machines, etc.

ARM Architecture

ARM is a **RISC** (reduced instruction set computer) architecture, but has some features typical of CISC (complex instruction set computer) architectures, namely:

- **conditional** execution
- wider range of:
  - operand **addressing modes**
  - memory **indexing modes**

Instruction Types: Examples

Data-Processing:

```
ADD R0, R1, R2      ; R0 = R1 + R2
SUB R0, R1, R2      ; R0 = R1 - R2
SUB R2, R3, #5      ; R2 = R3 - 5
```

Memory:

```
LDR R2, [R3]        ; R2 = mem[R3]
STR R4, [R5]        ; mem[R5] = R4
```

Branch:

```
B   Loop            ; branch to Loop label
BL  Loop            ; branch and link
```

ARM Register Set

Name	Use
R0	Argument / return value / temporary variable
R1-R3	Argument / temporary variables
R4-R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

Conditional Execution

ARM includes **condition flags** that can be:

- set by an instruction
- used to conditionally execute an instruction

Example uses:

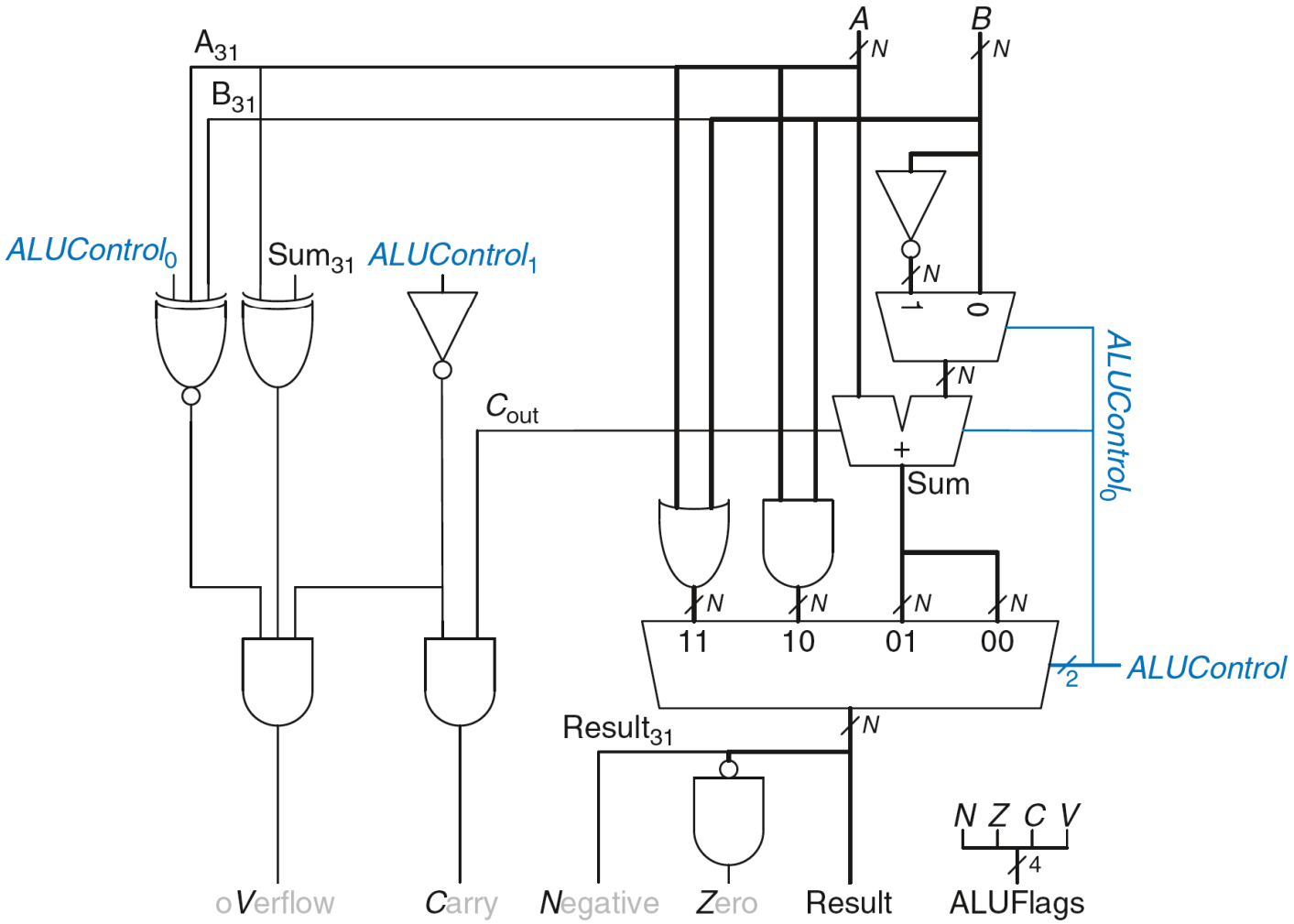
- **Conditional statements:** if/else, while loops, etc.: only want to execute code *if* a condition is true
- **Branching:** jump to another portion of code *if* a condition is true

ARM Condition Flags

Flag	Name	Description
N	**N**egative	Instruction result is negative
Z	**Z**ero	Instruction results in zero
C	**C**arry	Instruction causes an unsigned carry out
V	o**V**erflow	Instruction causes an overflow

- Set by ALU
- Held in *Current Program Status Register* (*CPSR*)

ARM ALU



ELSEVIER

Setting the Condition Flags: *NZCV*

Method 1: Compare instruction: `CMP`

Example: `CMP R5, R6`

- Performs: R5-R6
- Does not save result
- Sets flags

Setting the Condition Flags: *NZCV*

Method 1: Compare instruction: `CMP`

Example: `CMP R5, R6`

- Performs: R5-R6
- Does not save result
- Sets flags. If result:
  - Is 0, *Z*=1
  - Is negative, *N*=1
  - Causes a carry out, *C*=1
  - Causes a signed overflow, *V*=1

Setting the Condition Flags: *NZCV*

Method 2: Append instruction mnemonic with **`S`**

Example: `ADDS R1, R2, R3`

- Performs: R2 + R3
- Saves result in R1
- Sets flags: If result is 0 (Z=1), negative (N=1), etc.

Condition Mnemonics

- Instruction may be ***conditionally executed*** based on the condition flags
- Condition of execution is encoded as a ***condition mnemonic*** appended to the instruction mnemonic

Example:

```
CMP    R1, R2
SUBNE  R3, R5, R8
```

  - **NE:** not equal condition mnemonic
  - `SUB` will only execute if R1 ≠ R2 (i.e., Z = 0)

Condition Mnemonics

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\bar{Z}$
0010	CS / HS	Carry set / Unsigned higher or same	C
0011	CC / LO	Carry clear / Unsigned lower	$\bar{C}$
0100	MI	Minus / Negative	N
0101	PL	Plus / Positive of zero	$\bar{N}$
0110	VS	Overflow / Overflow set	V
0111	VC	No overflow / Overflow clear	$\bar{V}$
1000	HI	Unsigned higher	$\bar{Z}C$
1001	LS	Unsigned lower or same	$Z\ OR\ \bar{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\bar{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	$Z\ OR\ (N \oplus V)$
1110	AL (or none)	Always / unconditional	ignored

ARM Architecture

ARM is a **RISC** (reduced instruction set computer) architecture, but has some features typical of CISC (complex instruction set computer) architectures, namely:

- **conditional** execution
- **wider range of:**
  - operand **addressing modes**
  - memory **indexing modes**

Addressing Modes

How do we address operands?

- **Register Only**
- **Immediate**
- **Base**
- **PC-Relative**

Addressing Modes

How do we address operands?

- **Register Only**
- Immediate
- Base
- PC-Relative

Register Addressing

- Source and destination operands found in registers
- Used by data-processing instructions
- **Three submodes:**
  - Register-only
  - Immediate-shifted register
  - Register-shifted register

Register Addressing Examples

- **Register-only**

  Example: `ADD R0, R2, R7`

  `R0 = R2 + R7`

- **Immediate-shifted register**

  Example: `ORR R5, R1, R3, LSL #1`

  `R5 = R1 OR (R3 << 1)`

- **Register-shifted register**

  Example: `SUB R12, R9, R0, ASR R1`

  `R12 = R9 – (R0 >>> R1)`

Addressing Modes

How do we address operands?

- Register Only
- **Immediate**
- Base
- PC-Relative

Immediate Addressing

- Operands found in registers **and** immediates

  Example: `ADD R9, R1, #14`

- Unsigned immediate
- Encoded as
  - 8-bit immediate (*imm8*) (in this case = 14)
  - 4-bit rotation (*rot*) (in this case = 0)
  - 32-bit immediate = *imm8* ROR (*rot* x 2)
- Example: SUB R1, R2, #0x3400
  - *imm8* = 0x34
  - *rot* = 12: ROL 8 = ROR (32-8) = 24; 24/2 = 12

Addressing Modes

How do we address operands?

- Register Only
- Immediate
- **Base**
- PC-Relative

Base Addressing

- Address of operand is:

  base register +/- offset

- Offset can be a:
  - 12-bit Immediate
  - Register
  - Immediate-shifted Register

Base Addressing Examples

- **Immediate offset**

  Example: `LDR R0, [R8, #-11]`

  (R0 = mem[R8 - 11])

- **Register offset**

  Example: `LDR R1, [R7, R9]`

  (R1 = mem[R7 + R9])

- **Immediate-shifted register offset**

  Example: `STR R5, [R3, R2, LSL #4]`

  (R5 = mem[R3 + (R2 << 4)])

Addressing Modes

How do we address operands?

- Register Only
- Immediate
- Base
- **PC-Relative**

PC-Relative Addressing

- Used for branches
- Branch instruction format:
  - Operands are PC and a signed 24-bit immediate (*imm24*)
  - Changes the PC
  - New PC is relative to the old PC
  - *imm24* indicates the number of words away from PC+8
- PC = (PC+8) + (SignExtended(*imm24*) x 4)

ARM Architecture

ARM is a **RISC** (reduced instruction set computer) architecture, but has some features typical of CISC (complex instruction set computer) architectures, namely:

- **conditional** execution
- **wider range of:**
  - operand **addressing modes**
  - memory **indexing modes**

Memory Indexing Modes

Mode	Address	Base Reg. Update
Offset	Base register ± Offset	No change
Preindex	Base register ± Offset	Base register ± Offset
Postindex	Base register	Base register ± Offset

Examples

- **Offset:**

```
LDR R1, [R2, #4]     ; R1 = mem[R2+4]
```

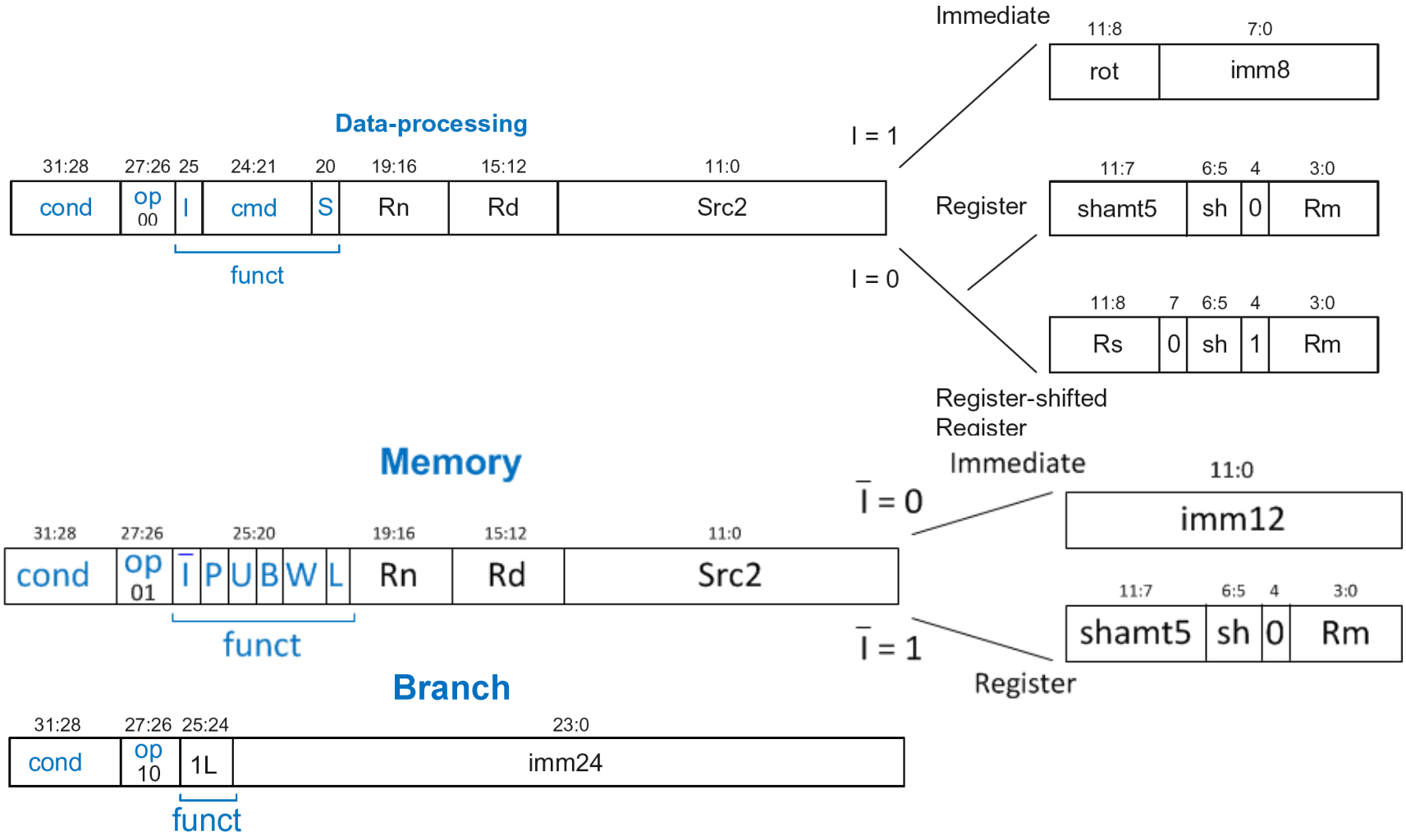
- **Preindex:**

```
LDR R3, [R5, #16]!   ; R3 = mem[R5+16]
                     ; R5 = R5 + 16
```

- **Postindex:**

```
LDR R8, [R1], #8     ; R8 = mem[R1]
                     ; R1 = R1 + 8
```

Instruction Formats



Programming Building Blocks

- **High-level Constructs:**
  - **if/else statements**
  - **for loops**
  - arrays
  - function calls

if Statement

C Code

```
if (i < j)
  f = g + h;

f = f - i;
```

ARM Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j

    CMP R3, R4        ; set flags with R3-R4
    BGE L1            ; if i>=j, skip if block
    ADD R0, R1, R2    ; f = g + h

L1
    SUB R0, R0, R2    ; f = f - i
```

if Statement: Alternate Code

C Code

```
if (i < j)
  f = g + h;
f = f - i;
```

ARM Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j

CMP   R3, R4      ; set flags with R3-R4
ADDLT R0, R1, R2  ; if (i==j) f = g + h
SUB   R0, R0, R2  ; f = f - i
```

if Statement: Alternate Code

ARM Assembly v1

```
;R0=f, R1=g, R2=h, R3=i, R4=j

 CMP R3, R4
 BGE L1
 ADD R0, R1, R2  ; f = g + h

L1
 SUB R0, R0, R2  ; f = f - i
```

ARM Assembly v2

```
;R0=f, R1=g, R2=h, R3=i, R4=j

 CMP   R3, R4      ; set flags with R3-R4
 ADDLT R0, R1, R2 ; if (i==j) f = g + h
 SUB   R0, R0, R2 ; f = f - i
```

for Loops

C Code

```
// adds numbers from 1-9
int sum = 0


for (i=1; i!=10; i=i+1)
   sum = sum + i;
```

ARM Assembly Code

```
; R0 = i, R1 = sum
  MOV   R0, #1          ; i = 1
  MOV   R1, #0          ; sum = 0

FOR
  CMP R0, #10           ; R0-10
  BEQ DONE              ; if (i==10)
                        ; exit loop
  ADD R1, R1, R0        ; sum=sum + i
  ADD R0, R0, #1        ; i = i + 1
  B   FOR               ; repeat loop

DONE
```

for Loops: Decremented Loops

In ARM, decremented loop variables are more efficient

C Code

```
// adds numbers from 1-9
int sum = 0


for (i=9; i!=0; i=i-1)
   sum = sum + i;
```

ARM Assembly Code

```
; R0 = i, R1 = sum
  MOV   R0, #9          ; i = 9
  MOV   R1, #0          ; sum = 0

FOR
  ADD   R1, R1, R0      ; sum=sum + i
  SUBS  R0, R0, #1      ; i = i - 1
                        ; and set flags
  BNE   FOR             ; if (i!=0)
                        ; repeat loop
```

Saves 2 instructions per iteration:

- Decrement loop variable & compare: `SUBS R0, R0, #1`
- Only 1 branch – instead of 2

for Loops: Decremented Loops

In ARM, decremented loop variables are more efficient

ARM Assembly Code

```
; R0 = i, R1 = sum
  MOV   R0, #1      ; i = 1
  MOV   R1, #0      ; sum = 0
FOR
  CMP R0, #10       ; R0-10
  BEQ DONE          ; if (i==10)
                    ; exit loop
  ADD R1, R1, R0    ; sum=sum + i
  ADD R0, R0, #1    ; i = i + 1
  B   FOR           ; repeat loop
DONE
```

ARM Assembly Code

```
; R0 = i, R1 = sum
  MOV   R0, #9          ; i = 9
  MOV   R1, #0          ; sum = 0
FOR
  ADD   R1, R1, R0      ; sum=sum + i
  SUBS  R0, R0, #1      ; i = i - 1
                        ; and set flags
  BNE   FOR             ; if (i!=0)
                        ; repeat loop
```

Saves 2 instructions per iteration:

- Decrement loop variable & compare: `SUBS R0, R0, #1`
- Only 1 branch – instead of 2

Programming Building Blocks

- **High-level Constructs:**
  - if/else statements
  - for loops
  - **arrays**
  - **function calls**

Arrays using for Loops

C Code

```
int array[200];
int i;

for (i=199; i >= 0; i = i - 1)
     array[i] = array[i] * 8;
```

ARM Assembly Code

```
; R0 = array base address, R1 = i
  MOV R0, 0x60000000
  MOV R1, #199

FOR
  LDR  R2, [R0, R1, LSL #2]     ; R2 = array(i)
  LSL  R2, R2, #3               ; R2 = R2<<3 = R3*8
  STR  R2, [R0, R1, LSL #2]     ; array(i) = R2
  SUBS R0, R0, #1               ; i = i - 1
                                ; and set flags
  BPL  FOR                      ; if (i>=0) repeat loop
```

Functions

ARM Assembly Code

```
  MOV R0, #4          ; arg0 = 4
  MOV R1, #5          ; arg1 = 5
  MOV R2, #6          ; arg2 = 6
  MOV R3, #7          ; arg3 = 7
  BL  DIFFOFSUMS      ; call diffofsums
  ...
DIFFOFSUMS
  STR R4, [SP, #-4]! ; save R4 on stack
  ADD R8, R0, R1      ; R8 = f + g
  ADD R9, R2, R3      ; R9 = h + i
  SUB R4, R8, R9      ; result = (f + g) - (h + i)
  MOV R0, R4          ; put return value in R0
  LDR R4, [SP], #4    ; restore R4 from stack
  MOV PC, LR          ; return to caller
```

Function Example

C Code

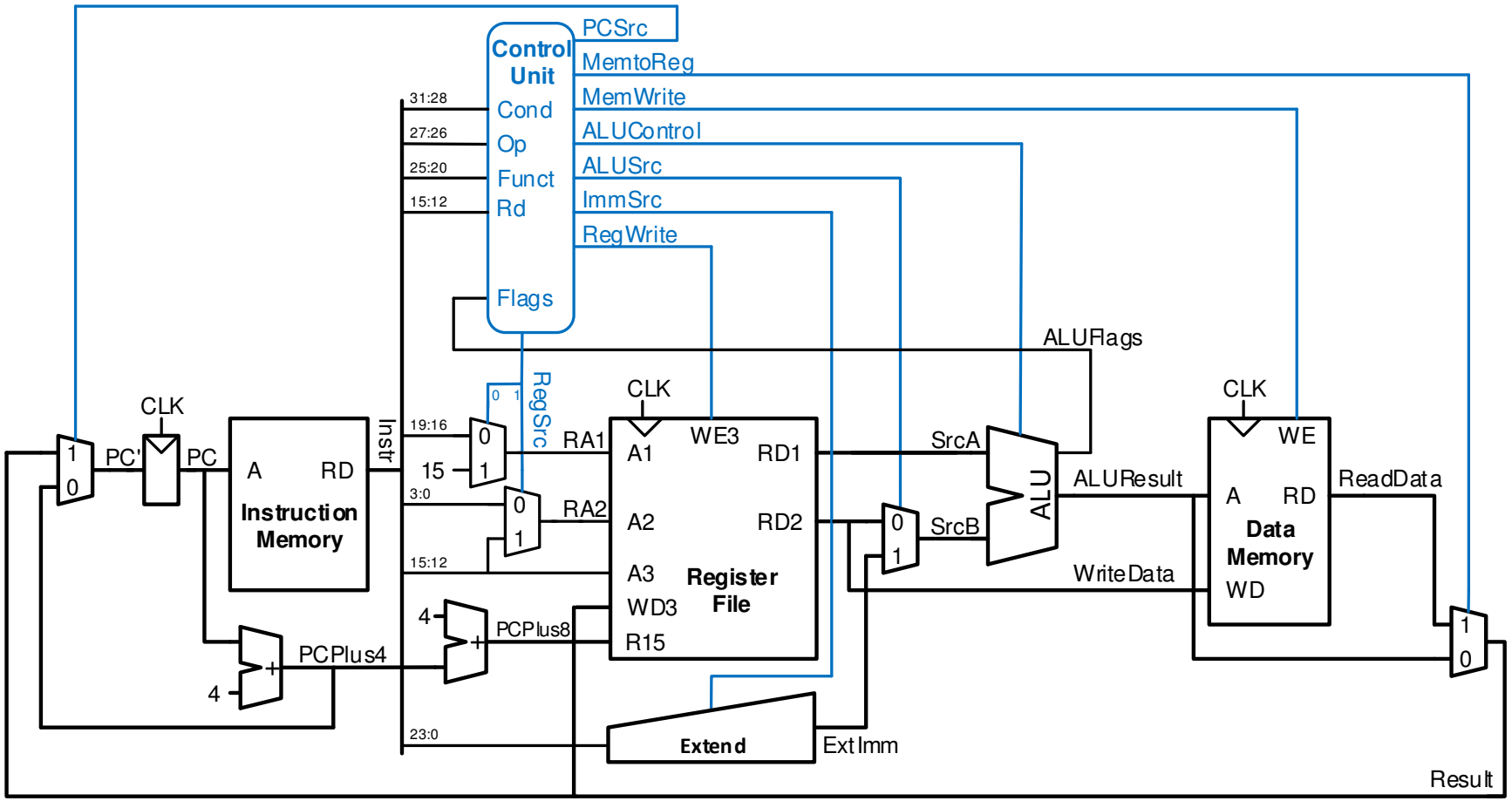
```
int f1(int a, int b) {
  ...
  x = f2(10);
  return x;
}
int f2(int p) {
  ...
  return p + 5;
}
```

ARM Assembly Code

```
F1
  PUSH {R4, R5, LR}
  ...
  MOV   R0, #10
  BL    F2
RETURN
  POP  {R4, R5, LR}
  MOV   PC, LR


F2
  PUSH {R4}
  ...
  ADD   R0, R0, 5
  POP  {R4}
  MOV   PC, LR
```

Single-Cycle ARM Processor



ARM Summary

Many similarities to MIPS, but some differences:

- Conditional Execution
- More addressing modes
- More indexing modes
- PC part of register file

Because of this:

- More complex instruction encoding
- (Slightly) more complex hardware
- Denser code in many cases

ARM Summary

Why is ARM winning in the mobile world?

(almost all mobile devices have an ARM core)

- Mobile offered a **new platform**: not tied to legacy software (like x86 for PCs).
- Offers some features of CISC architectures (conditional execution, addressing modes, indexing modes) that
  - allows programs to be **more dense**: useful for mobile devices with limited memory
  - while still keeping **hardware requirements low** (i.e., keeps power, chip area, cost, etc. low)
- Other companies (like MIPS) didn't evolve and take advantage of mobile revolution – got left behind.

Weiterführende Themen der Mikroarchitektur (auf Deutsch)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Tiefe Pipelines
- Sprungvorhersage
- Superskalare Prozessoren
- Out of Order-Prozessoren
- Umbenennen von Registern
- SIMD
- Multithreading
- Multiprozessoren

Tiefe Pipelines

TECHNISCHE UNIVERSITÄT DARMSTADT

- Üblicherweise 10-20 Stufen
  - Ausnahmen
    - Fehlkonstruktionen (Intel P4, >30 Stufen)
    - Anwendungsspezifische Spezialprozessoren (ggf. Hunderte von Stufen)

- Grenzen für Pipeline-Tiefe
  - Pipeline Hazards
  - Zusätzlicher Zeitaufwand für sequentielle Schaltungen
  - Elektrische Leistungsaufnahme und Energiebedarf
  - Kosten

Sprungvorhersage

TECHNISCHE UNIVERSITÄT DARMSTADT

- Idealer Pipelined-Prozessor: CPI = 1
- Fehler der Sprungvorhersage erhöht CPI

- Statische Sprungvorhersage:
  - Prüfe Sprungrichtung (vorwärts oder rückwärts)
  - Falls rückwärts: Sage “Springen” vorher
  - Sonst: Sage “Nicht Springen” vorher

- Dynamische Sprungvorhersage:
  - Führe Historie der letzten (einige Hundert) Verzweigungen in *Branch Target Buffer*, speichert:
    - Sprungziel
    - Wurde Sprung das letzte Mal / die letzten Male genommen?

Beispiel: Sprungvorhersage

TECHNISCHE UNIVERSITÄT DARMSTADT

```
  add  $s1, $0, $0       # sum = 0
  add  $s0, $0, $0       # i   = 0
  addi $t0, $0, 10       # $t0 = 10
for:
  beq  $s0, $t0, done    # falls i == 10, springe
  add  $s1, $s1, $s0     # sum = sum + i
  addi $s0, $s0, 1       # inkrementiere i
  j    for
done:
```

1-Bit Sprungvorhersage

TECHNISCHE UNIVERSITÄT DARMSTADT

- Speichert, ob die Verzweigung das letzte Mal genommen wurde
  - … und sagt genau dieses Verhalten für das aktuelle Mal vorher

- Fehlvorhersagen
  - Einmal bei Austritt aus der Schleife bei Schleifenende
  - Dann wieder bei erneutem Eintritt in Schleife

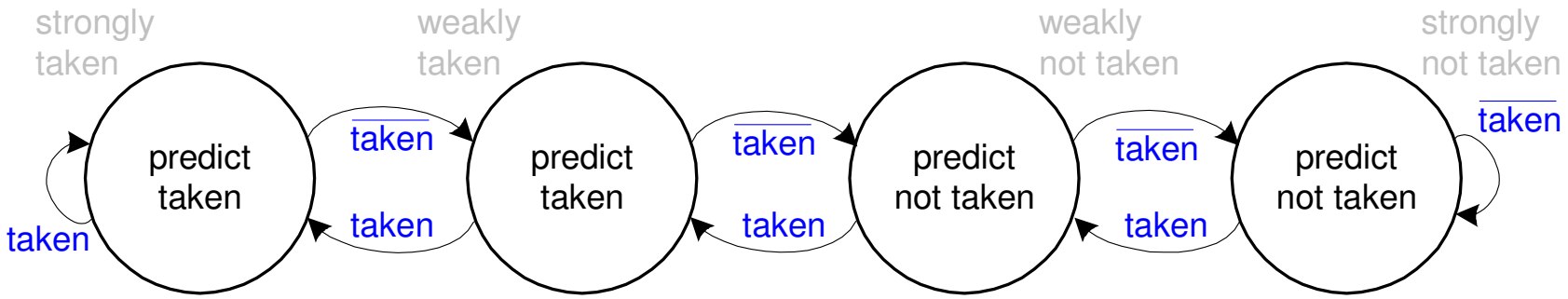
```
    add  $s1, $0, $0      # sum = 0
    add  $s0, $0, $0      # i   = 0
    addi $t0, $0, 10      # $t0 = 10
for:
    beq  $s0, $t0, done   # falls i == 10, springe
    add  $s1, $s1, $s0    # sum = sum + i
    addi $s0, $s0, 1      # inkrementiere i
    j    for
done:
```

Embedded Systems & Applications

2-Bit Sprungvorhersage

TECHNISCHE UNIVERSITÄT DARMSTADT

- Falsche Vorhersage nur beim letzten Sprung aus Schleife heraus



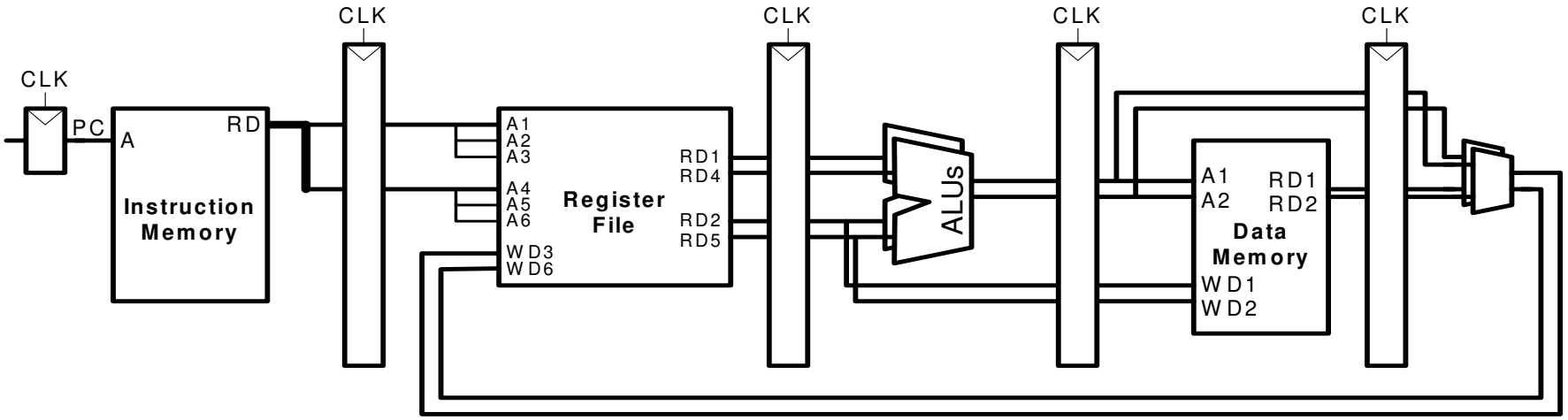

```
        add  $s1, $0, $0       # sum = 0
        add  $s0, $0, $0       # i   = 0
        addi $t0, $0, 10       # $t0 = 10
for:
        beq  $s0, $t0, done    # falls i == 10, springe
        add  $s1, $s1, $s0     # sum = sum + i
        addi $s0, $s0, 1       # inkrementiere i
        j    for
done:
```

Embedded Systems & Applications

Superskalare Mikroarchitektur

TECHNISCHE UNIVERSITÄT DARMSTADT

- Mehrere Instanzen des Datenpfades führen mehrere Instruktionen gleichzeitig aus
- Abhängigkeiten zwischen Instruktionen erschweren parallele Ausführung



Embedded Systems & Applications

Beispiel: Superskalare Ausführung

TECHNISCHE UNIVERSITÄT DARMSTADT

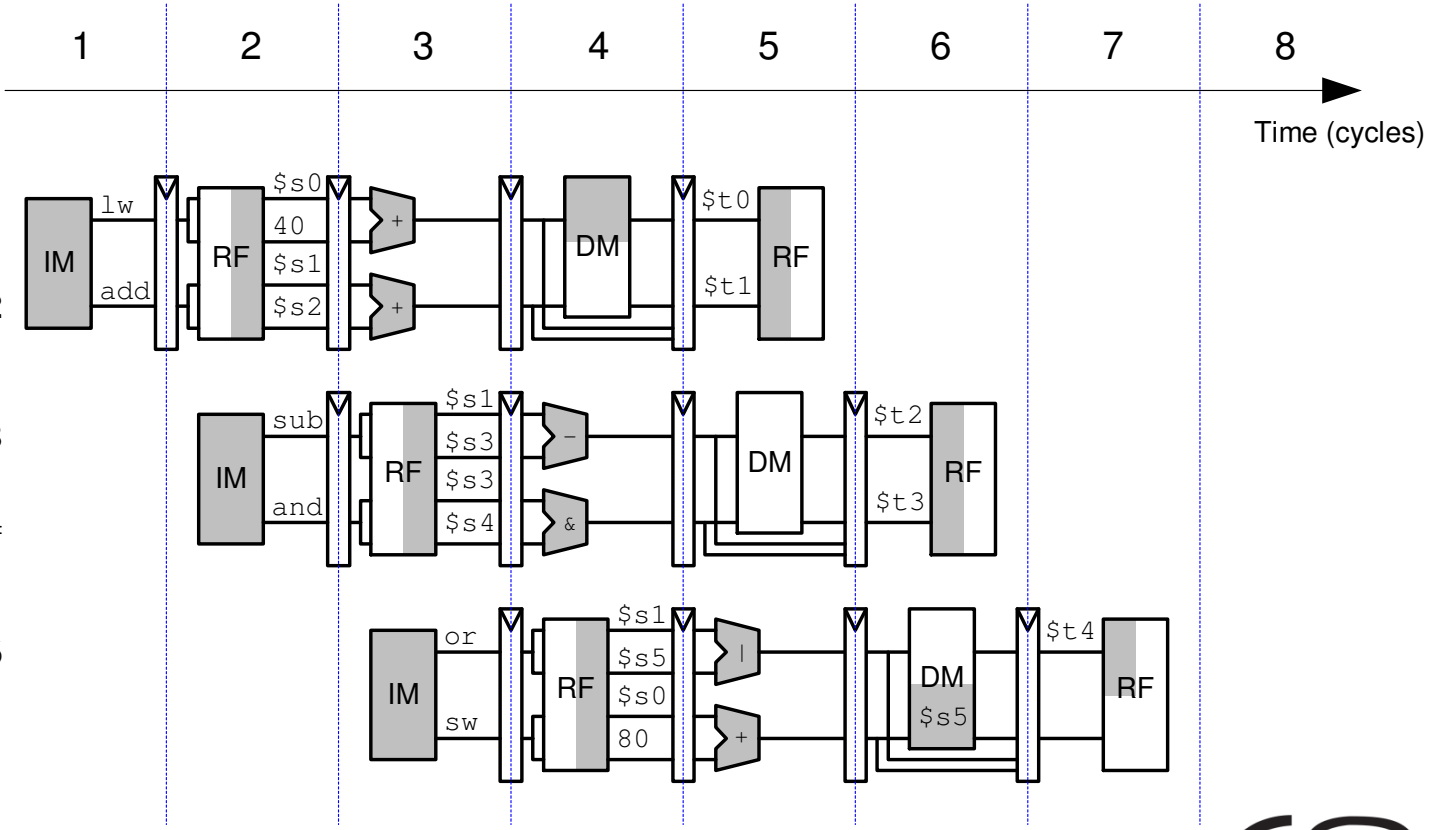
Idealer IPC-Wert: 2

Erreichter IPC-Wert: 2

```
lw  $t0, 40($s0)
add $t1, $s1, $s2

sub $t2, $s1, $s3
and $t3, $s3, $s4

or  $t4, $s1, $s5
sw  $s5, 80($s0)
```



Embedded Systems & Applications

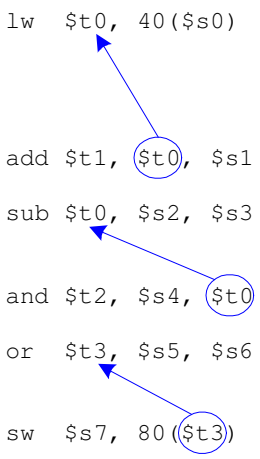
Beispiel: Superskalare Ausführung mit Abhängigkeiten

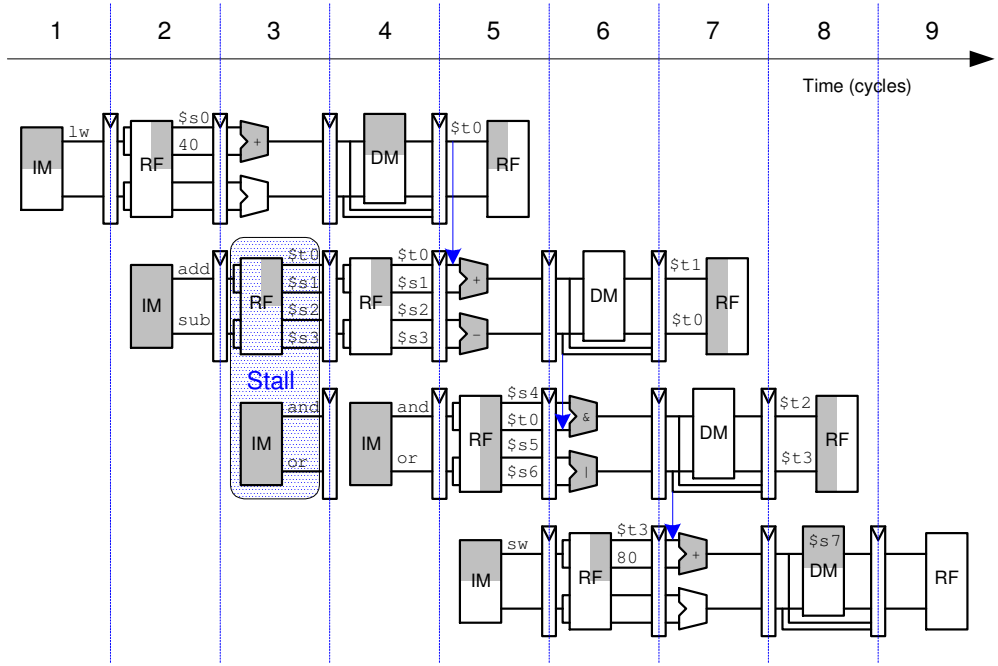
TECHNISCHE UNIVERSITÄT DARMSTADT

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```

Idealer IPC-Wert: 2,00

Erreichter IPC-Wert: 6/5 = 1,20





Embedded Systems & Applications

Out of Order-Mikroarchitektur

TECHNISCHE UNIVERSITÄT DARMSTADT

- Kann Ausführungsreihenfolge von Instruktion umsortieren
- Sucht im voraus nach parallel startbaren Instruktionen
- Startet Instruktionen in beliebiger Reihenfolge
  - Solange keine Abhängigkeiten verletzt werden!
- Abhängigkeiten
  - **RAW** (read after write)
    - Spätere Instruktion darf Register erst lesen, nachdem es vorher geschrieben wurde
  - **WAR** (write after read, anti-dependence)
    - Spätere Instruktion darf Register erst schreiben, nachdem es vorher gelesen wurde
  - **WAW** (write after write, output dependence)
    - Reihenfolge von in Register schreibenden Instruktionen muss eingehalten werden

Out of Order-Mikroarchitektur

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Parallelismus auf Instruktionsebene (*instruction level parallelism, ILP)***
  - Anzahl von parallel startbaren Instruktionen (i.d.R. < 3)

- **Scoreboard**
  - Tabelle im Prozessor
  - Verwaltet
    - Auf Start wartende Instruktionen
    - Verfügbare Recheneinheiten (z.B. ALUs)
    - Abhängigkeiten

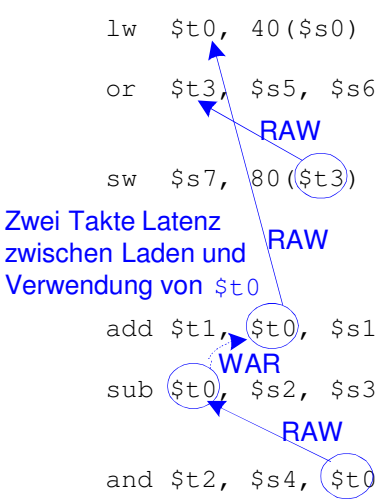
Beispiel: Out of Order-Mikroarchitektur

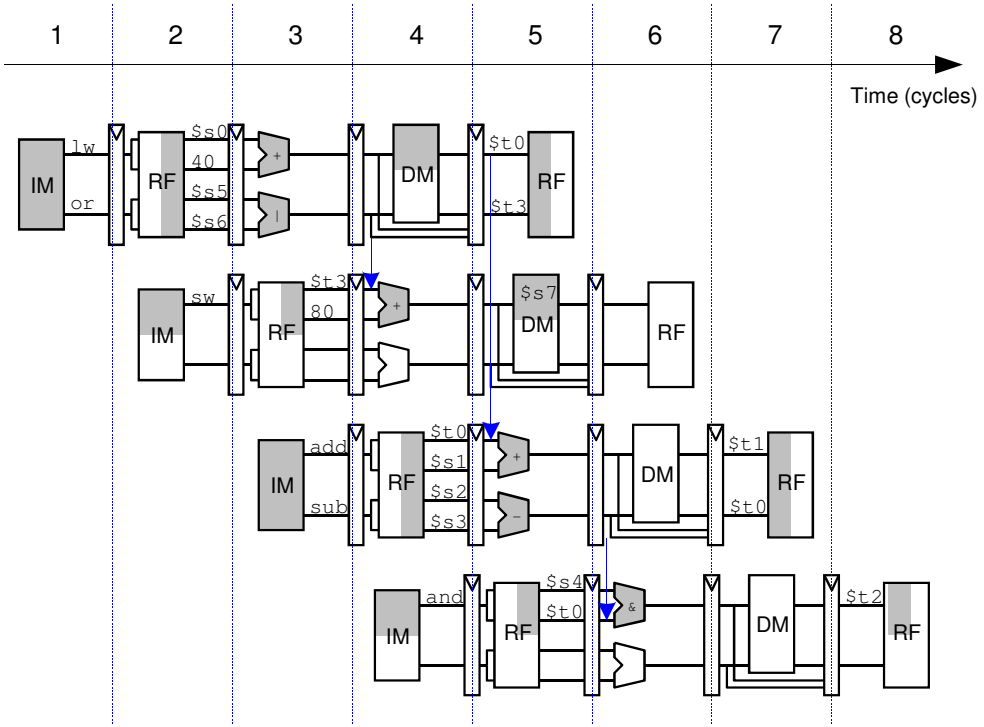
TECHNISCHE UNIVERSITÄT DARMSTADT

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```

Idealer IPC-Wert: 2,0

Erreichter IPC-Wert: 6/4 = 1,5





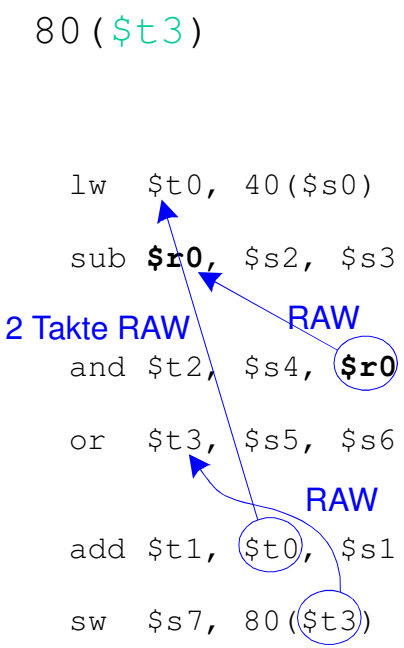
Embedded Systems & Applications

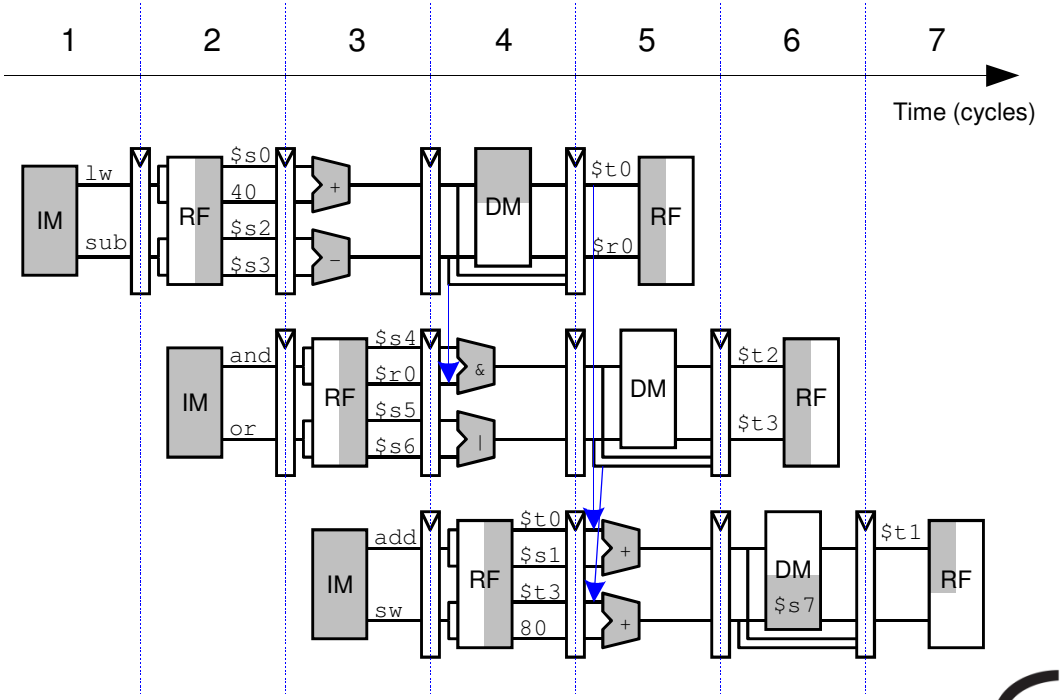
Umbenennen von Registern

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```

Idealer IPC-Wert: 2,0

Erreichter IPC-Wert: 6/3 = 2,0

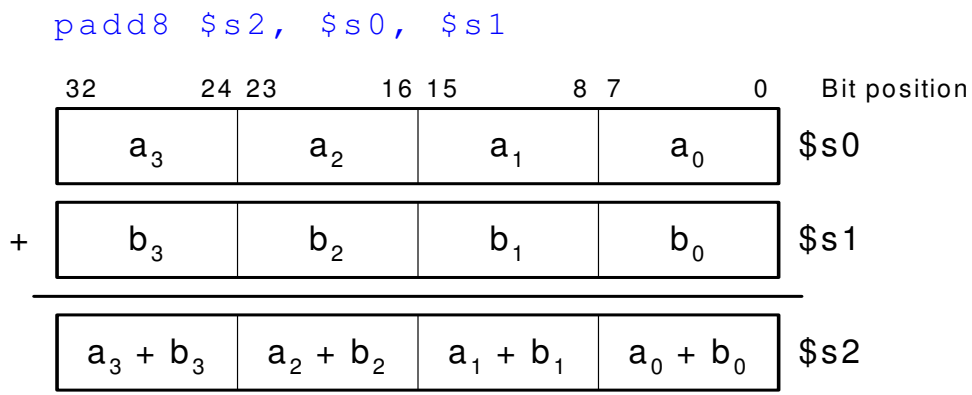




SIMD

TECHNISCHE UNIVERSITÄT DARMSTADT

- Single Instruction Multiple Data (SIMD)
  - Eine Instruktion wird auf mehrere Datenelemente gleichzeitig angewandt
  - Häufige Anwendung: Graphik, Multimedia
  - Oft: Führe schmale arithmetische Operatione aus
    - Auch genannt: gepackte Arithmetik
    - Beispiel: Addiere gleichzeitig vier Bytes
- ALU muss verändert werden
  - Kein Übertrag mehr zwischen einzelnen Bytes



Embedded Systems & Applications

Fortgeschrittene Mikroarchitekturtechniken

TECHNISCHE UNIVERSITÄT DARMSTADT

- Multithreading
  - Beispiel: Textverarbeitung
  - Threads (parallel laufende, weitgehend unabhängige Instruktionsfolgen)
    - Texteingabe
    - Rechtschreibprüfung
    - Drucken

- Multiprozessoren
  - Viele weitgehend unabhängige Prozessoren auf einem Chip
  - Am weitesten verbreitet heute in Grafikkarten (Hunderte von Prozessoren)
    - Aber auch in Spezialprozessoren, z.B. für UMTS Nachfolger LTE

Genauer: Multithreading

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Prozesse:** Auf dem Computer gleichzeitig laufende Programme
  - z.B. Web-Browser, Musik im Hintergrund, Textverarbeitung

- **Thread:** Parallele Ausführung als Teil eines Programmes
- Ein Prozess kann mehrere Threads enthalten

- In konventionellem Prozessor
  - Jeweils ein Thread wird ausgeführt
  - Wenn eine Thread-Ausführung einen *Stall* hat (z.B. Warten auf Speicher)
    - Sichere Architekturzustand des Threads
    - Lade Architekturzustand eines anderen, derzeit inaktiven aber lauffähigen Threads
    - Starte neuen Thread
    - Vorgang wird Kontextumschaltung (*context switching)* genannt
  - Alle Threads laufen scheinbar gleichzeitig

Embedded Systems & Applications

Multithreading auf Mikroarchitekturebene

TECHNISCHE UNIVERSITÄT DARMSTADT

- Mehrere Instanzen des Architekturzustandes im Prozessor
- Mehrere Threads nun gleichzeitig aktiv
  - Sobald ein Thread *stalled* wird sofort ein anderer gestartet
  - Kein Sichern/Laden von Architekturzustand mehr
  - Falls ein Thread nicht alle Recheneinheiten ausnutzt, kann dies ein anderer Thread tun

- Erhöht nicht den Grad an ILP innerhalb eines Threads
- Erhöht aber Durchsatz des Gesamtsystems mit mehreren Threads

Multiprozessoren

TECHNISCHE UNIVERSITÄT DARMSTADT

- Mehrere unabhängige Prozessorkerne mit einem dazwischenliegenden Kommunikationsnetz

- Arten von Multiprocessing:
  - **Symmetric multiprocessing (SMT)**: mehrere gleiche Kerne mit einem gemeinsamen Speicher
  - **Asymmetric multiprocessing:** unterschiedliche Kerne für unterschiedliche Aufgaben
    - Beispiel: CPU in Handy für GUI, DSP für Funksignalverarbeitung
  - **Clusters:** Jeder Kern hat seinen eigenen Speicher

Weiterführende Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Patterson & Hennessy
  Computer Architecture: A Quantitative Approach

- Konferenzen:
  - www.cs.wisc.edu/~arch/www/
  - ISCA (International Symposium on Computer Architecture)
  - HPCA (International Symposium on High Performance Computer Architecture)