

Systemy komputerowe (W02)

CPU - architektury
A. Pruszkowski

Procesor RISC-V

Systemy Komputerowe

■ RISC-V

- **Nowość na rynku**
- **Otwarta ISA (Instruction Set Architecture) z typowy RISC**
 - **brak opłat licencyjnych za tworzenie własnych mikroprocesorów czy mikrokontrolerów**
 - **specyfikacja żyje (!)**
 - ostatnia zatwierdzona wersja 2019.12.13
 - [<https://github.com/riscv/riscv-isa-manual/releases/download/draft-20200727-8088ba4/riscv-spec.pdf>]
 - tzw. model spójności pamięci Model RVWMO (RISC-V Weak Memory Ordering) wspiera budowę procesorów wielowątkowych
 - specyfikacja wprowadza pojęcie „hart” będącym odpowiednikiem sprzętowych wątków, których może być wiele w CPU, i każdy z nich może wykonywać swój własny program



Base	Version	Status
RVWMO	2.0	Ratified
RV32I	2.1	Ratified
RV64I	2.1	Ratified
<i>RV32E</i>	<i>1.9</i>	<i>Draft</i>
<i>RV128I</i>	<i>1.7</i>	<i>Draft</i>
Extension	Version	Status
M	2.0	Ratified
A	2.1	Ratified
F	2.2	Ratified
D	2.2	Ratified
Q	2.2	Ratified
C	2.0	Ratified
<i>Counters</i>	<i>2.0</i>	<i>Draft</i>
<i>L</i>	<i>0.0</i>	<i>Draft</i>
<i>B</i>	<i>0.0</i>	<i>Draft</i>
<i>J</i>	<i>0.0</i>	<i>Draft</i>
<i>T</i>	<i>0.0</i>	<i>Draft</i>
<i>P</i>	<i>0.2</i>	<i>Draft</i>
<i>V</i>	<i>0.7</i>	<i>Draft</i>
Zicsr	2.0	Ratified
Zifencei	2.0	Ratified
<i>Zam</i>	<i>0.1</i>	<i>Draft</i>
<i>Ztso</i>	<i>0.1</i>	<i>Frozen</i>

■ RISC-V

■ Nowość na rynku

■ Otwarta ISA (Instruction Set Architecture) z typowy RISC

■ specyfikacja ustala wiele wersji CPU - każda ma własną „ISA”

■ RV32I

- RV32M (dodane operacje mnożenia), RV32A (operacje atomowe), RV32F (operacje zmiennie przecinkowe), RV32D (operacje zmiennie przecinkowe rozszerzone dwukrotnie), RV32Q (operacje zmiennie przecinkowe rozszerzone czterokrotnie)

■ RV64I

- RV64M, RV64A, RV64F, RV64D, RV64Q

■ RV128I

- RV32E - tzw. „embedded microcontrollers”, jak RV32I ale tylko z rejestrami x0...x15
- RV32C, RV64C, RV128C - skompresowane instrukcje (większość ma 16bitów)

■ RISC-V - rejestry

- w specyfikacji RISC-V zdefiniowano stałą XLEN która określa liczbę bitów rejestrów stało przecinkowych
- **32 rejestry 32 bitowe (lub 64bitowe w CPU wersji 64bitowej, ...) stało przecinkowe**
 - x0 (alias: Zero) - zawsze wartość zero, nic nie można wpisać do jego treści
 - x1 (ra) - adres powrotu (Return Address)
 - x2 (sp) - wskaźnik stosu (Stack Pointer)
 - x3 (gp) - globalny wskaźnik (Global pointer)
 - x4 (tp) - wskaźnik wątku (Thread pointer)
 - x5...x7 (t0...t2) - rej. tymczasowe
 - x8 (s0/fp) - „zapisany rejestr” / wskaźnik ramki (Frame Pointer)
 - x9 (s1) - „zapisane rejestry”
 - x10...x11 (a0...a1) - argumenty funkcji / rezultat wykonania funkcji
 - x12...x17 (a2...a7) - kolejne argumenty funkcji
 - x18...x27 (s2...s11) - „zapisane rejestry”
 - x28...x31 (t3...t6) - rej. tymczasowe

■ RISC-V - instrukcje

■ Bazowy formaty instrukcji (typy i metoda kodowania)

31	30	25 24	21	20	19	15 14	12 11	8	7	6	0	
funct7		rs2			rs1	funct3		rd		opcode		R-type
imm[11:0]					rs1	funct3		rd		opcode		I-type
imm[11:5]			rs2		rs1	funct3		imm[4:0]		opcode		S-type
imm[12]	imm[10:5]		rs2		rs1	funct3		imm[4:1]	imm[11]	opcode		B-type
imm[31:12]								rd		opcode		U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]			rd		opcode		J-type	

■ Instrukcje mają postać

„kod operacji” rd, rs1, rs2

- rd - argument wynikowy (gdzie wynik zostanie zapisany)
- rs1 - argument źródłowy 1
- rs2 - argument źródłowy 2
- imm - argument wbudowany

■ RISC-V - instrukcje RV32I

■ Instrukcje typu U

- działania opierają się na liczbach całkowitych zapisanych w OPCODE (20 bitowe stałe)

■ LUI

- załadowanie do rd górnych 20 bitów z imm^{*}) (dolne ustawione na 0)
- $rd = imm[32...12] \mid zero[11...0]$
 - `lui a1,0x21`

■ AUIPC

- tworzy 32bitowy offset sumując PC z górnymi 20bitami z imm^{*}) (przyjmuje dolne 12 bitów jest równe zero)
- $rd = PC + (imm[32...12] \mid zero[11...0])$
 - `aui pc ra,0x0`

^{*}) imm jest tu rozszerzane ze znakiem z 12 bitowego do pełnej długości (32bitów)

■ RISC-V - instrukcje RV32I

■ Instrukcje typu I

- **działania opierają się na liczbach całkowitych zapisanych w rejestrach i w OPCODE (12 bitowe stałe)**
- **ADDI**
 - dodawanie imm^{*)} do rs1 i zapisanie w rd
 - `addi sp, sp, -16`
 - pseudo instrukcja: `MV rd,rs1` może być emulowana przez: `addi rd,rs1,0`
 - rozdział „RISC-V Assembly Programmer’s Handbook” specyfikacji zawiera „Table 25.2: RISC-V pseudoinstructions” opisująca wiele możliwości emulowania nie istniejących instrukcji
- **SLTI | SLTIU**
 - wstawia wartość 1 do rd jeżeli rs1 jest większy od imm^{*)}, w przeciwnym przypadku wstawia 0 do rd | jak w SLTI ale porównanie wykonywane jest bez znaku
- **ANDI | ORI | XORI**
 - operacje iloczynu | sumy | alternatywy rozłącznej na rs1 oraz imm^{*)} z zapisaniem wyniku w rd
 - pseudo instrukcja NOT może być zrealizowana przez: `XORI rd, rs1, -1`
- **SRLI | SRAI**
 - logiczne (starsze bity zapełniane zerami) | arytmetyczne (starsze bity zapełniane bitem znaku) przesunięcie w prawo

*) imm jest tu rozszerzane ze znakiem z 12 bitowego do pełnej długości (32bitów)

■ RISC-V - instrukcje RV32I

■ Instrukcje typu typu R

- działania opierają się na liczbach całkowitych zapisanych w rejestrach

■ ADD | SUB

- jak w typie I
- $rd = rs1 + rs2$ | $rd = rs1 - rs2$

■ AND | OR | XOR

- jak ANDI | ORI | XORI ale na rejestrach

■ SLT | SLTU

- wstawia wartość 1 do rd jeżeli rs1 jest większy od rs2, w przeciwnym przypadku wstawia 0 do rd | jak SLT ale bez znaku

■ SLL

- logiczne przesunięcie w lewo, młodsze bity zapełniane zerami
- $rd = rs1 \ll rs2[5..0]$

■ SRL | SRA

- logiczne | arytmetyczne przesunięcia w prawo
- $rd = rs1 \gg rs2[5..0]$

■ RISC-V - instrukcje RV32I

■ Instrukcje typu J

■ instrukcje sterowania transferem (skoki)

■ JAL

- bezwarunkowy skok relatywny (ang. relative jump)
- adres skoku wyznacza: $PC + imm_2$
- imm_2 to $imm[19..0]$ rozszerzone ze znakiem i pomnożone przez 2 (tzw. word alignment)
- PC użyte do operacji wyznaczania adresu wskazuje na aktualną instrukcję, ta wartość powiększona o 4 (następna instrukcja) trafia też do rd (typowo używane jest x1 lub x5)
- zasięg skoku +/- 1MiB

■ JALR

- bezwarunkowy skok nie bezpośredni (ang. indirect jump)
- imm jest tylko 12 bitowy ze znakiem
- adres skoku wyznacza: $rs1 + imm_2$

■ RISC-V - instrukcje RV32I

■ Instrukcje typu typu B

- w architekturze nie przewidziano rejestru flag przechowującego status po wykonanych operacjach(!), uznano że instrukcje rozgałęzienia z wbudowanym porównaniem rozwiążą problem
- **BEQ | BNE**
 - skok gdy rs1 i rs2 są sobie równe | są różne
 - gdy warunek nie jest spełniony - przejście do następnej instrukcji a gdy ma być wykonany skok - nowe miejsce w kodzie wyznacza: $PC + imm_2 * 2$
 - imm_2 to $imm[12...0]$ rozszerzone ze znakiem i pomnożone przez 2 (tzw. word alignment)
- **BLT | BLTU**
 - jak powyższe tyle, że skok gdy rs1 mniejsze od rs2 traktowane ze znakiem | bez znaku
- **BGE | BGEU**
 - jak powyższe tyle, że skok gdy rs1 większe od rs2 traktowane ze znakiem | bez znaku
- Operacje: BGT, BGTU, BLE oraz BLEU mogą być łatwo zrealizowane przez zamianę w kodzie miejscami rs1 z rs2 i użycie BLT, BLTU, BGE oraz BGEU - prostszy „krzem” bo mniej mnemoników, utrudnienie dla twórców kodu
- Zasięg ewentualnego skoku to +/- 4KiB

■ RISC-V - instrukcje RV32I

■ Architektura typu Load-Store

- Ograniczone tryby adresowania (typowe dla arch. risc)

■ Dostęp do pamięci możliwy jedynie za pomocą dwóch instrukcji

- efektywny adres odwołania do pamięci to $rs1 + imm[11...0]$

■ **LOAD, typ I**

- lw - załaduj do rd z pamięci spod efektywnego adresu słowo 32bitowe

- `lw ra, 12(sp)`

- lh | lhu - załaduj do rd z pamięci spod efektywnego adresu 16bitów, pozostałe bity w rd zapełnij bitem znaku | zerami

- lb | lbu - załaduj do rd z pamięci spod efektywnego adresu 8bitów, pozostałe bity w rd zapełnij bitem znaku | zerami

■ **STORE, typ S**

- sw | sh | sb - zapisz w pamięci pod efektywny adres wartość 32bitową | 16bitową | 8bitową z rs2

- dla sh i sb używaj młodszych bitów z rs2

- **RISC-V - instrukcje RV64I**
 - **Wspiera te same co RV32I**
 - **operuje na 64bitach - gdzie potrzeba rozszerza argumenty i wynik do takiej liczby bitów**
 - **Dostęp do pamięci**
 - lh | lhu | lb | lbu | sw | sh | sb - operują na przestrzeni 64bitowej
 - ld | sd - zapewniają odczyt | zapis 64bitów z | w pamięci z przestrzeni 64bitowej
- **RISC-V - instrukcje RV128I**
 - **operuje na 128bitach**
 - **przestrzeń pamięciowa 128bitowa**
 - **dodano instrukcje odwołań do pamięci**
 - LQ i SQ do obsługi pamięci za pomocą „quadword” (poczwórnych słów)
- **RISC-V - instrukcje RV32C**
 - **OPCODE 16 bitowe**
 - **oznaczenia mnemoników zaczynają się od C np.: C.ANDI, C.ADD, C.BEQZ, ...**
 - **Posiada instrukcję C.NOP**

Procesor „x86“

Systemy Komputerowe

■ X86 - Rejestry

- Rejestry segmentowe (16bitowe)
 - CS - seg. kodu
 - DS - seg. Danych
 - SS - seg. Stosu
 - ES | FS | GS - seg. dodatkowe
 - `add al,CS:EBX; add DS:EBX,12`
- Rejestry ogólnego przeznaczenia (32bitowe)
 - EAX - akumulator
 - EBX - adres pośredni/przemieszczenie
 - ECX - licznik pętli
 - EDX - wskaźnik portu I/O

Systemy Komputerowe

■ X86 - Rejestry

- Rejestry wskaźnikowe i indeksowe (32bitowe)
 - ESI - indeks źródłowy
 - EDI - indeks docelowy
 - EBP - wskaźnik bazowy segmentu stosu np.: SS:EBP
 - ESP - wskaźnik stosu

```
add EAX, DS:ESI
add DS:EDI, EAX
```
- Dla zgodności z kodem dla poprzednich generacji komputerów można używać:
 - rejestry 16 bitowe: AX, BX, CX, DX, BP, DI, SI, SP
 - rejestry 8 bitowe rejestry AL, BL, CL, DL, AH, BH, CH, DH
- Wskaźnik rozkazów EIP \equiv PC
 - element 32bitowy
- X86 w wersji 64bitowej
 - posiada 64bitowe rejestry z przedrostkiem „R” zamiast „E” np.:
 - RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, ... oraz dodatkowe rej. 64bitowe: r8...r15
 - IP nazywa się RIP (;->)

Systemy Komputerowe

■ X86 - Rejestry

- Rejestr stanu EFLAGS (32bitowy) - zawiera bity
 - CF (carry flag)
 - przeniesienie z najbardziej znaczącego bitu
 - PF (parity flag)
 - „1” gdy liczba jedynek wyniku ostatniej operacji jest parzysta
 - AF (auxiliary flag)
 - „1” gdy występuje przeniesienie z bitu numer 3 (kod BCD)
 - ZF (zero flag)
 - „1” gdy wynik ostatniej operacji jest zero
 - SF (sign flag)
 - „1” gdy wynik operacji jest ujemny (kod U2)
 - OF (overflow)
 - „1” gdy wynik operacji przekracza największą dodatnią lub najmniejszą ujemną liczbę reprezentowalną w kodzie U2 (nadmiar stałoprzecinkowy)
 - TF (trap flag)
 - wymuszenie pracy krokowej (po rozkazie skok do przerwania INT1)
 - IF (interrupt flag)
 - blokada przerwania maskowalnych
 - DF (direction flag)
 - zmiana SI/DI w kierunku adresów rosnących czy malejących
 - VM (virtual mode)
 - flaga ustawienia trybu wirtualnego 8086

Systemy Komputerowe

■ X86 - Rejestry

■ Rejestry sterujące:

- CR0...CR7
 - 32bitowe rejestry sterujące pracą procesora (control registers): tryb pracy, tryb pracy pamięci podręcznej "cache"
 - CR3 - używany w translacji pamięci związanej ze stronicowaniem
- DR0...DR7
 - 32bitowe rejestry uruchomieniowe (debug registers): adresy pułapek sprzętowych, status
- TR3...TR7,TR12
 - 32bitowe rejestry przeznaczone do testowanie procesora (test registers)
- GDTR(Global Descriptor Table Register), LDTR (Local Descriptor Table Register)
 - 48bitowe rejestry sterujące segmentacją pamięci
- IDTR (Interrupt Descriptor Table Register), TR (Task Register)
 - 16bitowe rejestry sterujące związane z segmentacją pamięci

Systemy Komputerowe

■ X86 - Instrukcje

- Instrukcje - typowo dwu argumentowe

„kod operacji” `arg.d, arg.z`

- Rozkazy przesłania

- zwykłe

- MOV - prześlij zawartość
- MOVSxx - prześlij zawartość łańcuchową
- XCHG - wymień wzajemnie zawartości
- SETC - ustaw arg. docelowy na 1 gdy CF=1
- SETZ - ustaw arg. docelowy na 1 gdy ZF=1
- SET...

- stosowe

- PUSH | POP - włoż na | zdejmij ze stosu rejestr
 - stos - specjalna przestrzeń pamięci danych, zarządzana przez rejestr SP/ESP
- PUSHA | POPA - włoż na | zdejmij ze stosu cały zestaw rejestrów
 - z zachowaniem kolejności: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI
- PUSHF | POPF - włoż na | zdejmij ze stosu rejestr EFLAGS
- ENTER | LEAVE - rezerwacja | zwolnienie na stosie ramki dla zmiennych lokalnych

Systemy Komputerowe

■ X86 - Instrukcje

■ Rozkazy przesłania

■ specjalne

- LAHF | SAHF- załaduj do AH zawartość EFLAGS | zapisz AH w EFLAGS
- LDS,LES,LFS,LGS,LSS - załaduj 48bitowy daleki adres (segment:przemieszczenie)
- LEA - zapisz efektywny adresu argumentu źródłowego
 - np.: lea eax, [ebp-4]
- ARPL - zmiana pola RPL selektora
- LGDT | SGDT - załaduj do | zapisz z rejestru Globalnej Tablicy Deskryptorów
- LIDT | SIDT - załaduj do | zapisz z rejestru Tablicy Deskryptorów Przerwań
- LLDT | SLDT - załaduj do | zapisz z rejestru Lokalnej Tablicy Deskryptorów
- LMSW - załaduj CR0 stanem maszyny
- LSL - załaduj wielkość wskazanego segmentu do rejestru (segmentacja)
- LTR | STR - załaduj do | zapisz z rejestru zadania
- CPUID - identyfikacja jednostki centralnej (CPU), zwraca np.:
 - Later Intel 486 and Pentium 0x01
 - Xeon 0x02
 - Atom 0x0A
 - IvyBridge-based processors 0x0D
- ...

Systemy Komputerowe

■ X86 - Instrukcje

■ Rozkazy arytmetyki całkowitoliczbowej

- ADD | ADC - dodanie | dodanie z uwzględnieniem flagi CF
- SUB | SBB - odjęcie | odjęcie z uwzględnieniem flagi CF
- INC | DEC - zwiększ | zmniejsz o 1 (lub inną wartość!)
- CMP - porównaj
- NEG - zaneguj ($dst=0-src$)
- MUL | IMUL - mnożenie całkowitoliczbowe arg. bez znaku | ze znakiem
- DIV | IDIV - dzielenie całkowitoliczbowe arg. bez znaku | ze znakiem
 - generuje przerwanie „dzielenia przez zero”
- XCHG - wzajemna wymiana wartości dwóch arg.
- XADD - jak XCHG i dodawanie
 - $T=src+dst$; $src=dst$; $dst=T$, gdzie T rej. Tymczasowy, żaden z dostępnych dla programisty

■ Rozkazy arytmetyki BCD

- DAA | DAS - korekcja dodawania | odejmowania w upakowanym kodzie BCD
- AAA | AAS - korekcja dodawania | odejmowania w nie upakowanym kodzie BCD
- AAD - korekcja dzielnej w nie upakowanym kodzie BCD przed dzieleniem
- AAM - korekcja wyniku w nie upakowanym kodzie BCD po mnożeniu

Systemy Komputerowe

■ X86 - Instrukcje

■ Operacje bitowe

■ Operacje logiczne

- NOT - negacja boolowska
- AND - iloczyn boolowski
- OR - suma boolowska
- XOR - alternatywa wykluczająca

■ Operacje na bitach

- TEST - porównanie logiczne dwóch argumentów (jak CMP tyle że operacją AND)
- BSF | BSR - przeszukanie w przód | w tył, arg. w celu znalezienia pozycji bitu równego „1”
- BT - skopiowanie do CF wartości bitu o numerze w arg2, z arg1
- BTS | BTR | BTC - skopiowanie do CF wartości bitu o numerze w arg2, a w arg1 w jego miejsce ustawienie | wyzerowanie | zanegowanie bitu wsk. w arg2

Systemy Komputerowe

■ X86 - Instrukcje

■ Rozkazy przemieszczeń i obrotów

- SAL | SAR - przesunięcie arytmetyczne w lewo | prawo z powielaniem najstarszego | najmłodszego bitu, nie mieszczące się bity kopiowane są do CF
- SHL | SHR - przesunięcie arytmetyczne w lewo | prawo z zerowaniem najstarszego | najmłodszego bitu, nie mieszczące się bity kopiowane są do CF
- SHLD | SHRD - jak SHL | SHR ale operacje wykonywane są na podwójnej precyzji argumentu
- RCL | RCR - rotacja w lewo | prawo, nie mieszczące wstawiane są do CF a wcześniejsza wartość CF kopiowana na następny | najmłodszy bit
- ROL | ROR - rotacja w lewo | prawo, nie mieszczące wstawiane są do CF i kopiowane na następny | najmłodszy bit

Systemy Komputerowe

■ X86 - Instrukcje

■ Rozkazy sterowania

■ Sterowanie programem

- JMP - skok bezwarunkowy
- JA | JNBE | JAE | JNB | JB | JNAE | JBE | JNA | JC | JCXZ | JE | JECXZ | JZ | JNC | JNE | JNZ | JNP | JPO | JP | JPE | JG | JNLE | JGE | JNL | JL | JNGE | JLE | JNG | JNO | JNS | JNZ | JO | JS - skoki warunkowe jeżeli ponad | nie poniżej lub równe |
- LOOP | LOOPZ | LOOPNE | LOOPNZ - rozkaz pętli wykonywanej aż warunek w którym licznik ECX=0 | ECX=0 i ZF=1 | ECX=0 i ZF=0 będą spełnione
 - LOOPZ ≡ LOOPE, LOOPNZ ≡ LOOPNE
- REP | REPE(Z) | REPNE(Z) - powtarzaj następną instrukcję dopóki warunek ECX=0 | ZF=0 | ZF=1 będzie spełniony
 - REPE ≡ REPZ, REPNE ≡ REPNEZ
- BOUND - sprawdzenie czy indeks podany jako arg1. leży w tablicy wskazanej przez arg2, i ograniczonej przez jej ostatni element wskazany przez arg3
- LOCK - wygenerowanie przez CPU sygnału LOCK#, w systemach wieloprocesorowych zapewnia to wykluczający dostęp do pamięci dzielonej z innymi procesorami

Systemy Komputerowe

■ X86 - Instrukcje

■ Rozkazy sterowania

■ Podprogramy i przerwania

■ CALL - skok do podprogramu

- dziejesz języki programowania C/C++/Java/Python bardzo mocno korzystają z funkcji/procedur
- typowo programista w ciele funkcji/procedury na jej początku zachowuje na stosie wszystkie używane przez siebie w funkcji/procedurze rejestry

■ RET - powrót z podprogramu

- typowo programista przed tą instrukcją odtwarza wszystkie używane przez siebie i zachowane w prologu funkcji/procedurze rejestry ze stosu w kolejności odwrotnej do zapamiętanych w CALL

■ IRET - powrót z podprogramu obsługi przerwania (przywraca rejestr EFLAGS)

- rejestr EFLAGS w tym procesorze zapamiętywany jest automatycznie podczas wywołania podprogramu obsługi przerwań - celem transparentnego wykonania takiego kodu

■ INT - przerwanie programowe

- można je wywołać jawnie w kodzie

■ INTO - wywołanie podprogramu obsługi wyjątku nadmiaru

Systemy Komputerowe

■ X86 - Instrukcje

■ Rozkazy sterowania

■ Sterowanie systemem

- HLT - zatrzymanie CPU aż do
 - przejścia w stan RESET
 - wywołana przerwania sprzętowego
- WAIT - wprowadzenie CPU w stan oczekiwania aż pojawi się sygnał na końcówce WAIT#
 - synchronizowanie pracy z koprocesorem
- NOP - rozkaz pusty
- RSM - wznowienie pracy z trybu zarządzania systemem (System Management Mode)
- INVD | WBINVD - opróżnienie pamięci podręcznej | opróżnienie i skopiowanie treści do głównej pamięci
- INVLPG - opróżnienie wpisów w pamięci TLB (Translation Lookaside Buffer) - część MMU
- VERR | VERW - sprawdzenie możliwości odczytu z | zapisu do segmentu bez generowania błędu ochrony

Systemy Komputerowe

■ X86 - Instrukcje

■ Rozkazy sterowania

■ Operacje na znacznikach

- STC | CLC | CMC - ustawianie | zerowanie | negowanie flagi CF
- STD | CLD - ustawianie | skasowanie znacznika DF
- STI | CLI - ustawienie | skasowanie znacznika odblokowania przerwań IF
- CLTS - ustawienie flagi TS (Task-Switched), wymagany poziom ochrony CPL=0

Systemy Komputerowe

■ X86 - Instrukcje

■ Rozkazy łańcuchowe

- MOVSB | MOVSW | MOVSD - przesłanie bajtu | słowa | podwójnego słowa z miejsca wskazanego przez ESI na pozycję wskazywaną przez EDI, równocześnie zwiększenie lub zmniejszenie (zależnie od flagi DF) ESI i EDI

: [EDI] = : [ESI]; ESI++/--; EDI ++/--

- CMPSB | CMPSW | CMPSD - porównanie łańcuchowe

: [EDI] ?= : [ESI]; ESI++/--; EDI ++/--

- SCASB | SCASW | SCASD - skanowanie łańcuchowe

: [EDI] ?= EAX; EDI ++/--

- LODSB | LODSW | LODSD - ładowanie łańcuchowe

EAX = : [ESI]; ESI ++/--

- STOSB | STOSW | STOSD - zapamiętanie łańcuchowe

: [EDI] = EAX; EDI ++/--

- Instrukcja REP wspiera operacje łańcuchowe, wykonywane są one automatycznie aż ZF stanie się 1 (zakładane jest że operacja łańcuchowa po każdym kroku modyfikuje ZF), np.:

```
mov ESI, 0x12345678 //miejsce w pamięci
```

```
mov AL, 0x00 //szukany znak
```

```
rep scasb //skanuj w poszukiwaniu znaku identycznego jak w AL.
```

```
... //tutaj ESI wskaże miejsce w pamięci gdzie jest znaleziony znak
```

Systemy Komputerowe

■ X86 - Instrukcje

■ Rozkazy wejścia-wyjścia

- IN - odczytanie bajtu lub słowa z portu wejściowego
 - in AL, DX
- OUT - zapisano do porty wyjściowego bajtu lub słowa
 - out DX, AL
- INSB | INSW | INSD - łańcuchowe odczytanie bajtu | słowa | podwójnego słowa z portu wejściowego
 - insb ES:[EDI], DX //ES:[EDI] - miejsce gdzie odczytany bajt będzie umieszczony, DX-numer portu
- OUTSB | OUTSW | OUTSD - łańcuchowe przeniesienie bajta | słowa | podwójnego słowa do portu wyjściowego
 - outsb DX, ES:[EDI] //DX-numer portu, ES:[EDI] - miejsce gdzie odczytany bajt będzie umieszczony

Systemy Komputerowe

■ X86 - Instrukcje

■ Rozkazy konwersji i translacji

- CBW - konwersja bajtu w słowo, z uwzględnieniem znaku
 - $AX \leq AL$
- CWDE - konwersja słowa w podwójne słowo, z uwzględnieniem znaku
 - $EAX \leq AX$
- CWD - konwersja słowa w podwójne słowo, z uwzględnieniem znaku
 - $DX:AX \leq AX$
- CDQ - konwersja podwójnego słowa w poczwórne słowo
 - $EDX:EAX \leq EAX$
- BSWAP - konwersja wartości 32bitowej zapisanej z użyciem grubokońcowej reprezentacji w wartość z użyciem cienkońcowej reprezentacji
 - $t=dst; dst[0..7]=t[24..31]; dst[8..15]=t[16..23]; dst[16..23]=t[8..15]; dst[24..31]=t[0..7];$
- XLATB - tłumaczenie indeksu (AL) do elementu w tablicy (wskazana przez BX) na wartość elementu (wynik w AL)
 - $AL = :[BX+AL]$

Systemy Komputerowe

■ X86 - Tryby adresowania

- Tryb adresowania - sposób odwoływania się w rozkazie do miejsca przechowywania argumentów lub sposób zapisania wyniku
 - Tryb rejestrowy
 - np.: ADD EAX, EBX
 - informacje które rejestry (tu EAX i EBX) użyć zapisana w OPCODE
 - Tryb natychmiastowy - argument znajduje się w OPCODE
 - np.: ADD EAX, 5
 - tutaj argument 5 jest przechowywany w OPCODE jako pewna jego liczba bitów (tzw. argument wbudowany)
 - Tryb bazowy (zwany pośrednim rejestrowym)
 - np.: ADD EAX, [EBX]
 - wskazanie [EBX] to odwołanie do pamięci zgodnie z wartością w EBX
 - Tryb przemieszczeniowy
 - np.: ADD EAX, [245AH]
 - odwołanie do pamięci z przemieszczeniem równym 245AH, wartość ta z reguły zapisana jako tzw. argument zewnętrzny - czyli jako zestaw bitów zapisanych zaraz po OPCODE o liczbie identycznej jak szerokość OPCODE

Systemy Komputerowe

■ X86 - Tryby adresowania

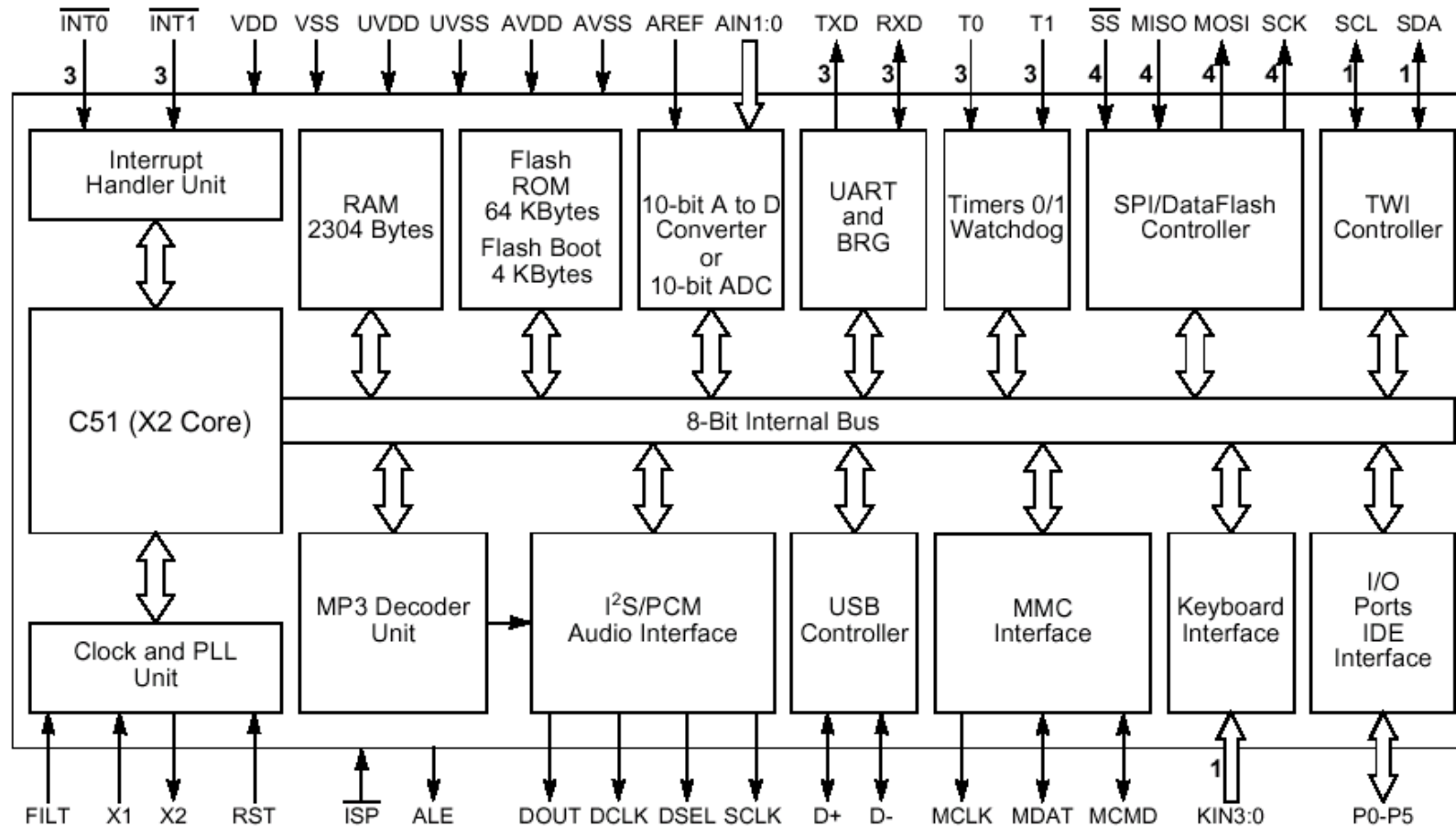
- Trybem adresowania - sposób odwoływania się w rozkazie do miejsca przechowywania argumentu, cd.
 - Tryb baza-przemieszczenie
 - np.: `ADD EAX, [EBX+4H]`
 - adres efektywny gdzie trzeba sięgnąć do pamięci to suma EBX i stałej 4H
 - Tryb indeks-skala-przemieszczenie
 - np.: `ADD EAX, [8*EBX + 4]`
 - adres efektywny gdzie trzeba sięgnąć do pamięci to iloczyn EBX i wartości 8 oraz dodany do stałej 4H
 - z reguły wartość używana w iloczynie (tu: 8) to 2^N gdzie N to mała liczba 1...6, liczba N mieści się w paru bitach OPCODE
 - Tryb baza-indeks-skala-przemieszczenie
 - `ADD EAX, [EBP + 8*EBX + 4]`
 - jak powyżej ale z dodatkowo dodaną zawartością EBP

Inne procesory

mikrokontrolery

Systemy Komputerowe

■ C51 (AT83C51SND1C) - ogólny widok wnętrza (głównie peryferii)



Systemy Komputerowe

■ C51 (AT83C51SND1C) - wycinek listy rozkazów

	0	1	2	3	4	5	6	7
0	NOP	JBC bit,rel [3B, 2C]	JB bit, rel [3B, 2C]	JNB bit, rel [3B, 2C]	JC rel [2B, 2C]	JNC rel [2B, 2C]	JZ rel [2B, 2C]	JNZ rel [2B, 2C]
1	AJMP (P0) [2B, 2C]	ACALL (P0) [2B, 2C]	AJMP (P1) [2B, 2C]	ACALL (P1) [2B, 2C]	AJMP (P2) [2B, 2C]	ACALL (P2) [2B, 2C]	AJMP (P3) [2B, 2C]	ACALL (P3) [2B, 2C]
2	LJMP addr16 [3B, 2C]	LCALL addr16 [3B, 2C]	RET [2C]	RETI [2C]	ORL dir, A [2B]	ANL dir, A [2B]	XRL dir, a [2B]	ORL C, bit [2B, 2C]
3	RR A	RRC A	RL A	RLC A	ORL dir, #data [3B, 2C]	ANL dir, #data [3B, 2C]	XRL dir, #data [3B, 2C]	JMP @A + DPTR [2C]
4	INC A	DEC A	ADD A, #data [2B]	ADDC A, #data [2B]	ORL A, #data [2B]	ANL A, #data [2B]	XRL A, #data [2B]	MOV A, #data [2B]
5	INC dir [2B]	DEC dir [2B]	ADD A, dir [2B]	ADDC A, dir [2B]	ORL A, dir [2B]	ANL A, dir [2B]	XRL A, dir [2B]	MOV dir, #data [3B, 2C]
6	INC @R0	DEC @R0	ADD A, @R0	ADDC A, @R0	ORL A, @R0	ANL A, @R0	XRL A, @R0	MOV @R0, #data [2B]
7	INC @R1	DEC @R1	ADD A, @R1	ADDC A, @R1	ORL A, @R1	ANL A, @R1	XRL A, @R1	MOV @R1, #data [2B]

Systemy Komputerowe

■ C51 (AT83C51SND1C) - wycinek opisu jednej instrukcji

ADDC A, <src-byte>

Function: Add with Carry

Description: ADDC simultaneously adds the byte variable indicated, the carry flag and the Accumulator contents, leaving the result in the Accumulator. The carry and auxiliary-carry flags are set respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.

OV is set if there is a carry-out of bit 6 but not out of bit 7, or a carry-out of bit 7 but not out of bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands or a positive sum from two negative operands.

Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

Example: The Accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B) with the carry flag set. The following instruction,

```
ADDC    A,R0
```

leaves 6EH (01101110B) in the Accumulator with AC cleared and both the Carry flag and OV set to 1.

ADDC A,R_n

Bytes: 1

Cycles: 1

Encoding:

0	0	1	1	1	r	r	r
---	---	---	---	---	---	---	---

Operation: ADDC
 $(A) \leftarrow (A) + (C) + (R_n)$

ADDC A,direct

Bytes: 2

Cycles: 1

Encoding:

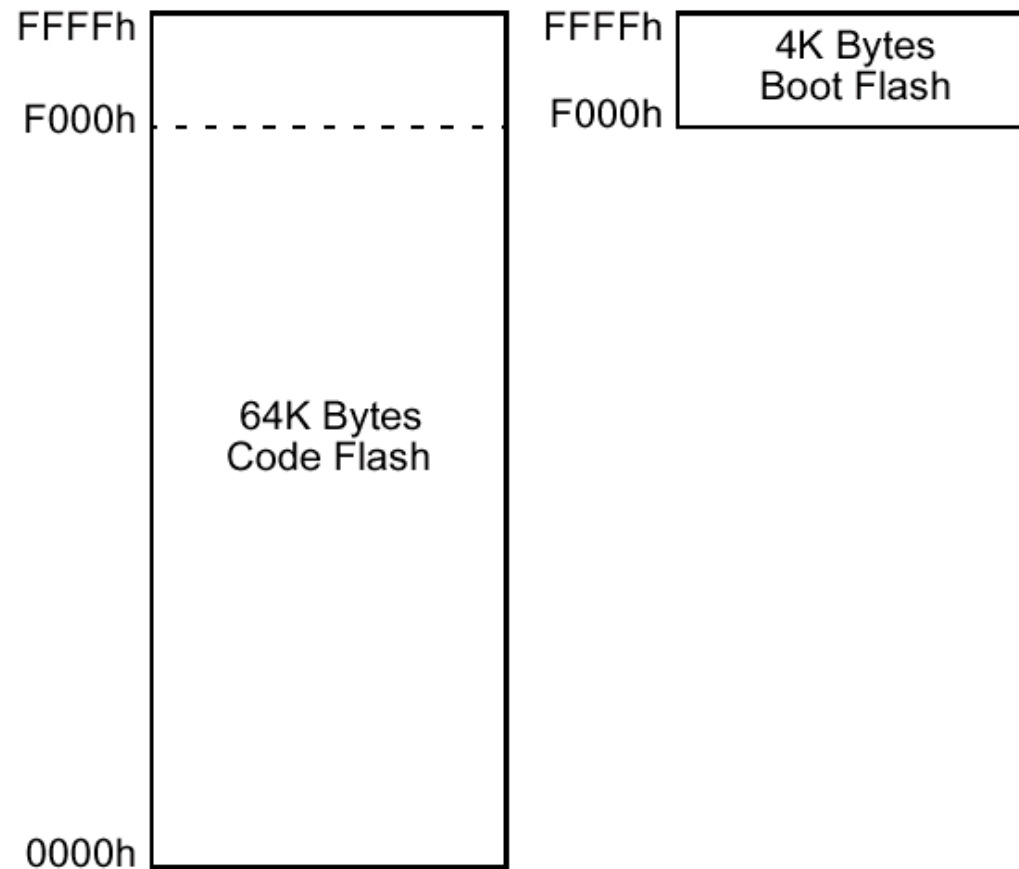
0	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---

direct address

Operation: ADDC
 $(A) \leftarrow (A) + (C) + (\text{direct})$

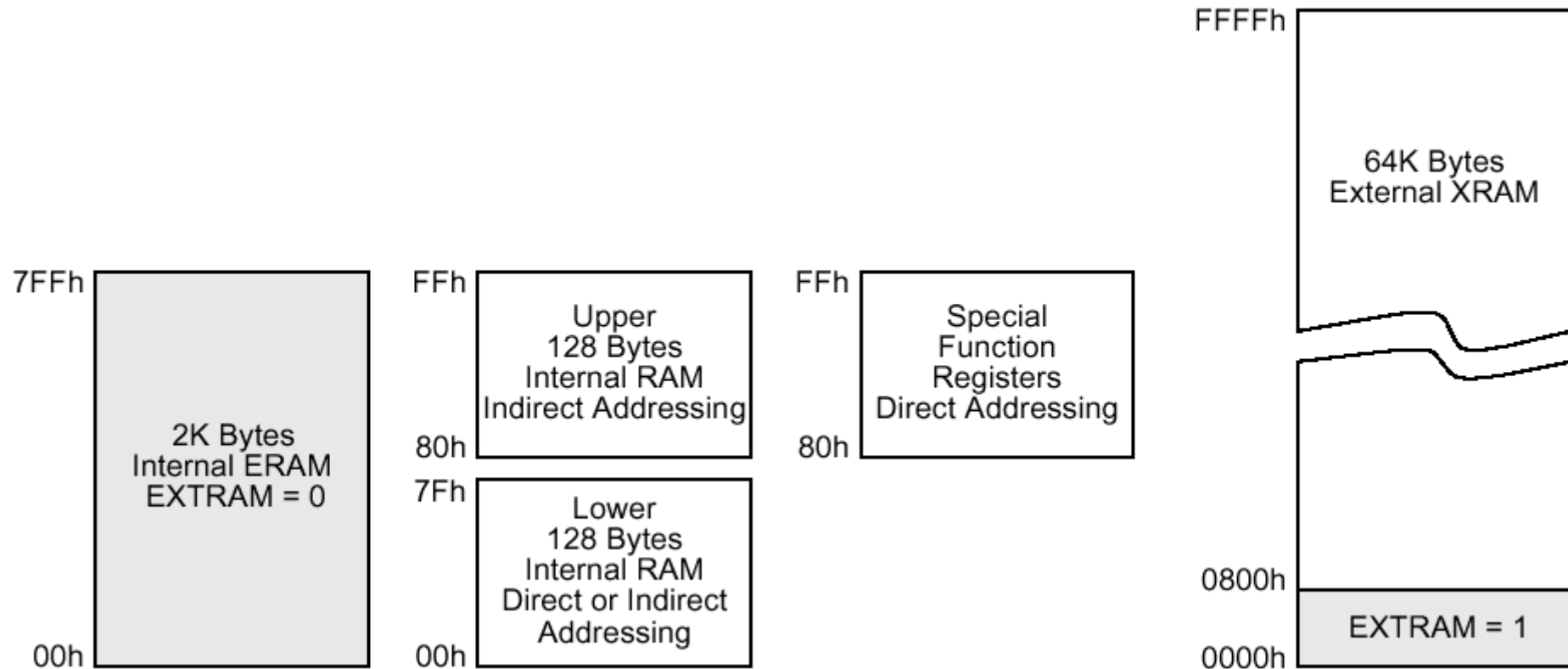
Systemy Komputerowe

- C51 (AT83C51SND1C) - przestrzeń pamięci kodu



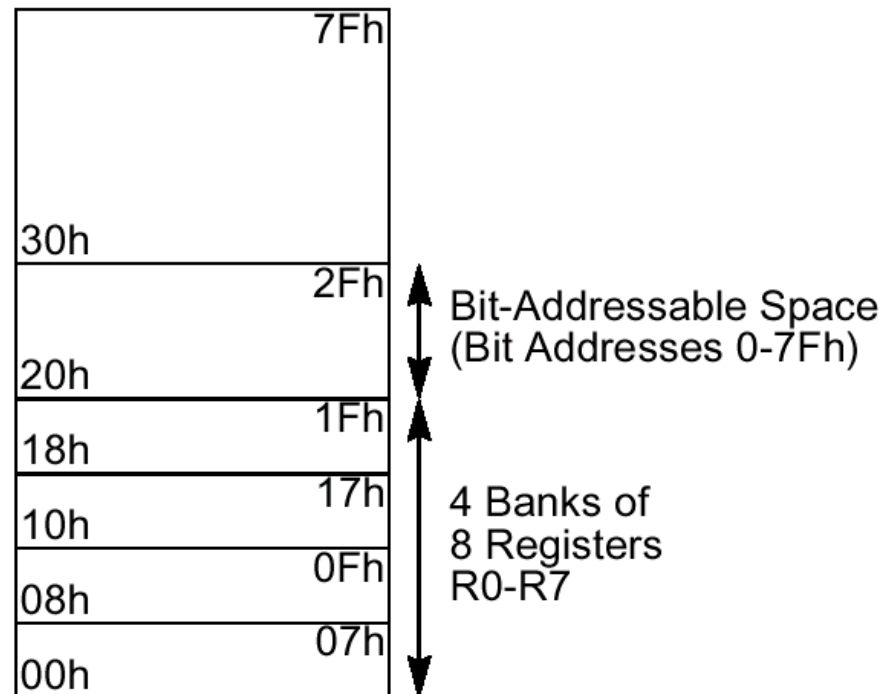
Systemy Komputerowe

■ C51 (AT83C51SND1C) - przestrzeń pamięci danych



Systemy Komputerowe

- **C51 (AT83C51SND1C): DATA - pamięć danych (adresy 0x00...0x7f)**
 - Pamięć zwana jako „Internal RAM” może działać z
 - adresowaniem bezpośrednim („Direct Addressing”)
 - adresowaniem pośrednim („Indirect Addressing”)



Systemy Komputerowe

■ C51 (AT83C51SND1C): SFR (Special Function Register)

- Obszar rejestrów specjalnego przeznaczenia (adresy 0x80...0xFF)

- Przykładowa mapa

pamięci jednej z odmian C51

0F8H								0FFH
0F0H	B 00000000							0F7H
0E8H								0EFH
0E0H	ACC 00000000							0E7H
0D8H								0DFH
0D0H	PSW 00000000							0D7H
0C8H	T2CON 00000000	T2MOD XXXXXX00	RCAP2L 00000000	RCAP2H 00000000	TL2 00000000	TH2 00000000		0CFH
0C0H								0C7H
0B8H	IP XX000000							0BFH
0B0H	P3 11111111							0B7H
0A8H	IE 0X000000							0AFH
0A0H	P2 11111111		AUXR1 XXXXXXXX0				WDTRST XXXXXXXX	0A7H
98H	SCON 00000000	SBUF XXXXXXXXXX						9FH
90H	P1 11111111							97H
88H	TCON 00000000	TMOD 00000000	TL0 00000000	TL1 00000000	TH0 00000000	TH1 00000000	AUXR XXX00XX0	8FH
80H	P0 11111111	SP 00000111	DP0L 00000000	DP0H 00000000	DP1L 00000000	DP1H 00000000	PCON 0XXX0000	87H

Systemy Komputerowe

■ C51 (AT83C51SND1C): SFR - konsekwencje mapowania rejestrów

■ Akumulator

```
MOV A, #5
```

- działa identycznie jak

```
MOV ACC, #5
```

- gdzie „A” to nazwa rejestru, gdy „ACC” to rejestr „SFR” pod adresem 0xE0

■ Wskaźnik stosu

- SP - mapowany pod adresem 0x81

■ Rejestr flag

- PSW - mapowany pod adresem 0xD0

■ Wskaźnik DPTR

- dzielony na dwie części: DPL i DPH - mapowane pod adresami 0x82 i 0x83

■ Licznik rozkazów

- IP (*interrupt priority*) to nie PC!!!, nie jest mapowany w SFR

Systemy Komputerowe

- **C51 (AT83C51SND1C): SFR - obszar rejestrów specjalnego rejestr PSW (flagi)**
 - PSW - mapowany pod adresem 0xD0
 - PSW.0 (P) - parzystość
 - PSW.1 (F1) - w 80C52 flaga kontrolowana przez aplikacje użytkownika
 - PSW.2 (Overflow) - przepełnienie
 - PSW.3 (RS0) - młodszy bit aktualnie używanego numeru banku rejestrów (Register Bank Switch)
 - PSW.4 (RS1) - starszy bit aktualnie używanego numeru banku rejestrów (Register Bank Switch)
 - PSW.5 (F0) - flaga ogólnego zastosowania
 - PSW.6 (AC) - przeniesienie z młodszych 4 bitów do starszego 4 bitów (Auxiliary Carry)
 - PSW.7 (CY) - przeniesienie z najstarszego bitu

Systemy Komputerowe

■ C51 (AT83C51SND1C): IDATA - pamięć dostępna pośrednio

■ Dostęp bezpośredni

```
MOV direct, #5
```

- dla $0 < \text{„direct”} < 0x7f$ trafi do „DATA”
- dla $\geq 0x80$ trafi do „SFR” - co oznacza „Direct addressing”
 - dla $\text{direct} = 0xE0$, będzie to równoważne z `MOV ACC, #5`

■ Dostęp pośredni

```
MOV @R0, #5
```

- gdyby jednak R0 było równe $0xE0$, wartość 5 nie trafi do ACC, ale do pamięci IDATA („Internal RAM Indirect Addressing”)

■ Stos (jako przykład adresowania pośredniego) może być umieszczony w pamięci IDATA

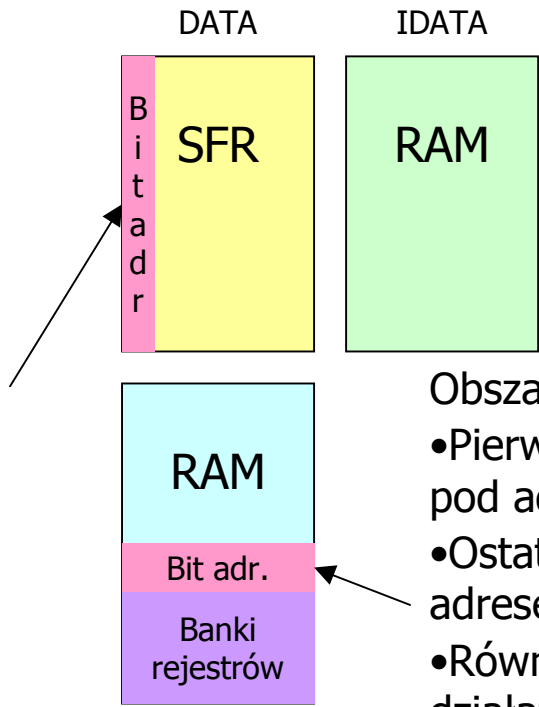
- decyzje gdzie - podejmuje programista z reguły pierwszych instrukcjach kodu aplikacji

```
MOV SP, #0xF0
```

Systemy Komputerowe

- **C51 (AT83C51SND1C) - adresowanie bitowe**
 - **specyficzny sposób odwołań do pamięci**
 - **Po określonych lokacjach można odwoływać się tradycyjnie i bitowo**

Tutaj bitowo adresowalne są wyłącznie rejestry o adresach podzielnych bez reszty przez 8, np.: PSW ma adres „bajtowy” 0xD0, ale jego bit PSW.7 ma adres „bitowy” 0xD7



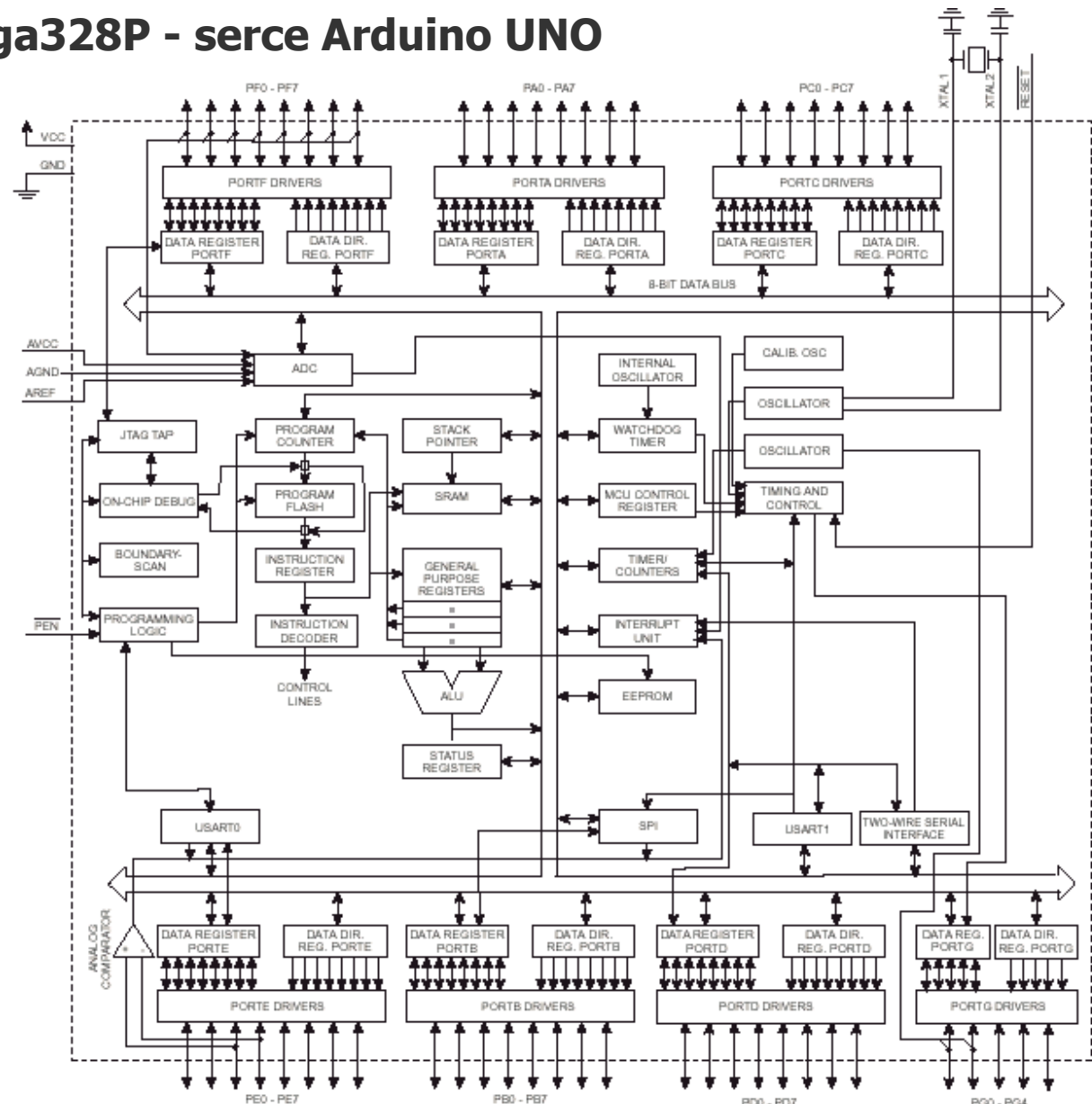
Obszar o specjalnym adresowaniu bitowym:

- Pierwszy bit adresowalny bitowo - leży w „bajcie” DATA pod adresem „bajtowym” 0x20 ale ma adres „bitowy” 0x00
- Ostatni bit adresowalny bitowo leży w „bajcie” DATA pod adresem „bajtowym” 0x2F i ma adres „bitowy” 0x7F
- Równoważne zapisy (tylko niektóre instrukcje tak działają)

```
CLR 0x2F.6  
CLR 0x7E
```

Systemy Komputerowe

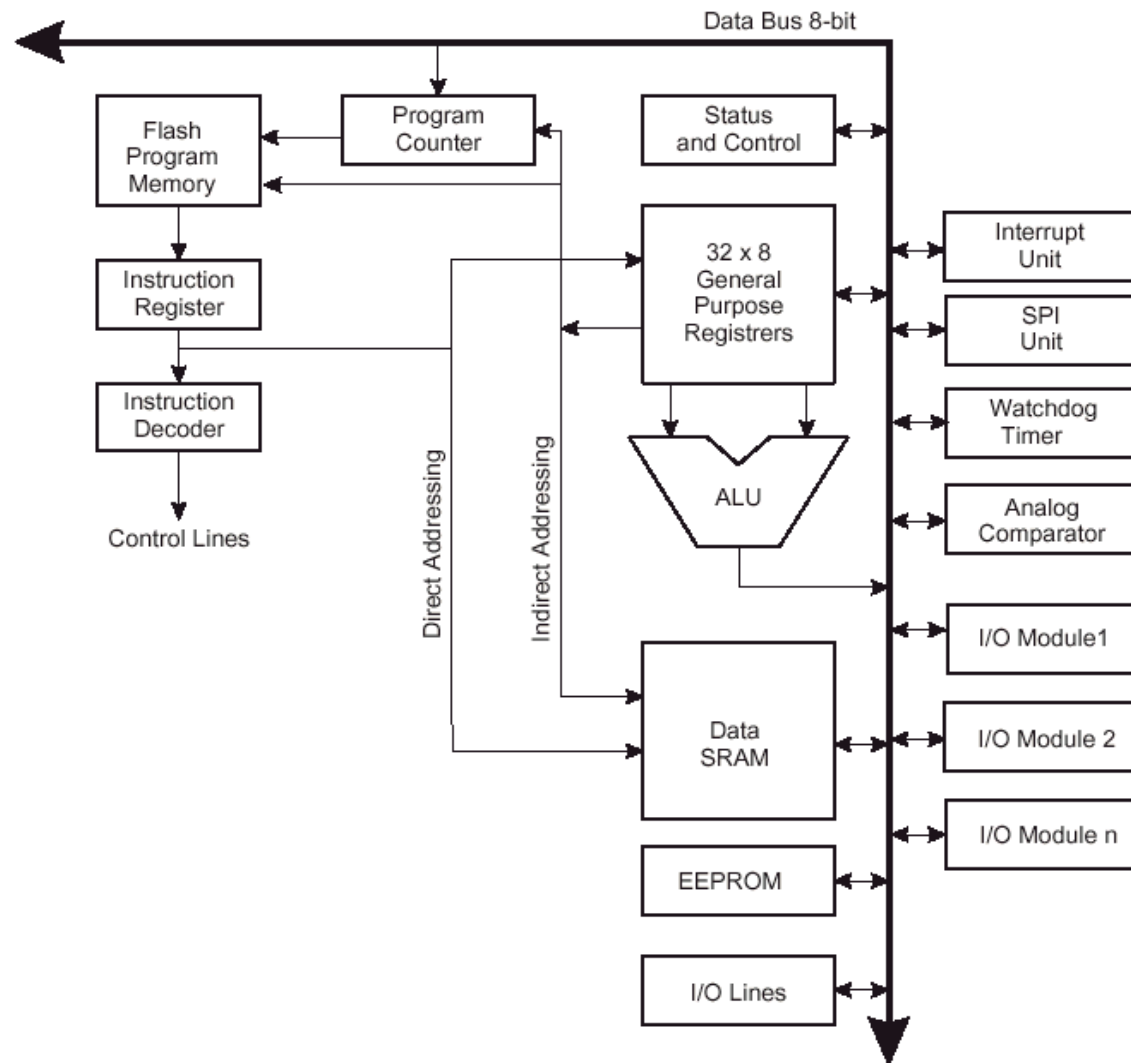
- AVR (ATmega128) - ogólny widok wnętrza
 - bogatszy brat ATmega328P - serce Arduino UNO



Źródło: Materiały informacyjne firmy Atmel

Systemy Komputerowe

■ AVR (ATmega128) - uproszczony widok wnętrza



Systemy Komputerowe

■ AVR (ATmega128) - wycinek listy rozkazów

Mnemonics	Operands	Description	Operation	Flags	#Clock Note
Arithmetic and Logic Instructions					
ADD	Rd, Rr	Add without Carry	$Rd \leftarrow Rd + Rr$	Z,C,N,V,S,H	1
ADC	Rd, Rr	Add with Carry	$Rd \leftarrow Rd + Rr + C$	Z,C,N,V,S,H	1
ADIW	Rd, K	Add Immediate to Word	$Rd+1:Rd \leftarrow Rd+1:Rd + K$	Z,C,N,V,S	2
SUB	Rd, Rr	Subtract without Carry	$Rd \leftarrow Rd - Rr$	Z,C,N,V,S,H	1
SUBI	Rd, K	Subtract Immediate	$Rd \leftarrow Rd - K$	Z,C,N,V,S,H	1
SBC	Rd, Rr	Subtract with Carry	$Rd \leftarrow Rd - Rr - C$	Z,C,N,V,S,H	1
SBCI	Rd, K	Subtract Immediate with Carry	$Rd \leftarrow Rd - K - C$	Z,C,N,V,S,H	1
SBIW	Rd, K	Subtract Immediate from Word	$Rd+1:Rd \leftarrow Rd+1:Rd - K$	Z,C,N,V,S	2
AND	Rd, Rr	Logical AND	$Rd \leftarrow Rd \bullet Rr$	Z,N,V,S	1
ANDI	Rd, K	Logical AND with Immediate	$Rd \leftarrow Rd \bullet K$	Z,N,V,S	1
OR	Rd, Rr	Logical OR	$Rd \leftarrow Rd \vee Rr$	Z,N,V,S	1
ORI	Rd, K	Logical OR with Immediate	$Rd \leftarrow Rd \vee K$	Z,N,V,S	1
EOR	Rd, Rr	Exclusive OR	$Rd \leftarrow Rd \oplus Rr$	Z,N,V,S	1
COM	Rd	One's Complement	$Rd \leftarrow \$FF - Rd$	Z,C,N,V,S	1
NEG	Rd	Two's Complement	$Rd \leftarrow \$00 - Rd$	Z,C,N,V,S,H	1
SBR	Rd,K	Set Bit(s) in Register	$Rd \leftarrow Rd \vee K$	Z,N,V,S	1
CBR	Rd,K	Clear Bit(s) in Register	$Rd \leftarrow Rd \bullet (\$FFh - K)$	Z,N,V,S	1

Systemy Komputerowe

■ AVR (ATmega128) - wycinek opisu jednej instrukcji

ADC - Add with Carry

Description:

Adds two registers and the contents of the C flag and places the result in the destination register Rd.

Operation:

(i) $Rd \leftarrow Rd + Rr + C$

Syntax:

(i) ADC Rd,Rr

Operands:

$0 \leq d \leq 31, 0 \leq r \leq 31$

Program Counter:

$PC \leftarrow PC + 1$

16-bit Opcode:

0001	11rd	dddd	rrrr
------	------	------	------

Status Register (SREG) Boolean Formulae:

I	T	H	S	V	N	Z	C
-	-	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow	\Leftrightarrow

H: $Rd3 \cdot Rr3 + Rr3 \cdot \overline{Rd3} + \overline{Rd3} \cdot Rr3$

Set if there was a carry from bit 3; cleared otherwise

S: $N \oplus V$, For signed tests.

V: $Rd7 \cdot Rr7 \cdot \overline{Rd7} + \overline{Rd7} \cdot Rr7 \cdot Rd7$

Set if two's complement overflow resulted from the operation; cleared otherwise.

N: R7

Set if MSB of the result is set; cleared otherwise.

Z: $\overline{R7} \cdot \overline{R6} \cdot \overline{R5} \cdot \overline{R4} \cdot \overline{R3} \cdot \overline{R2} \cdot \overline{R1} \cdot \overline{R0}$

Set if the result is \$00; cleared otherwise.

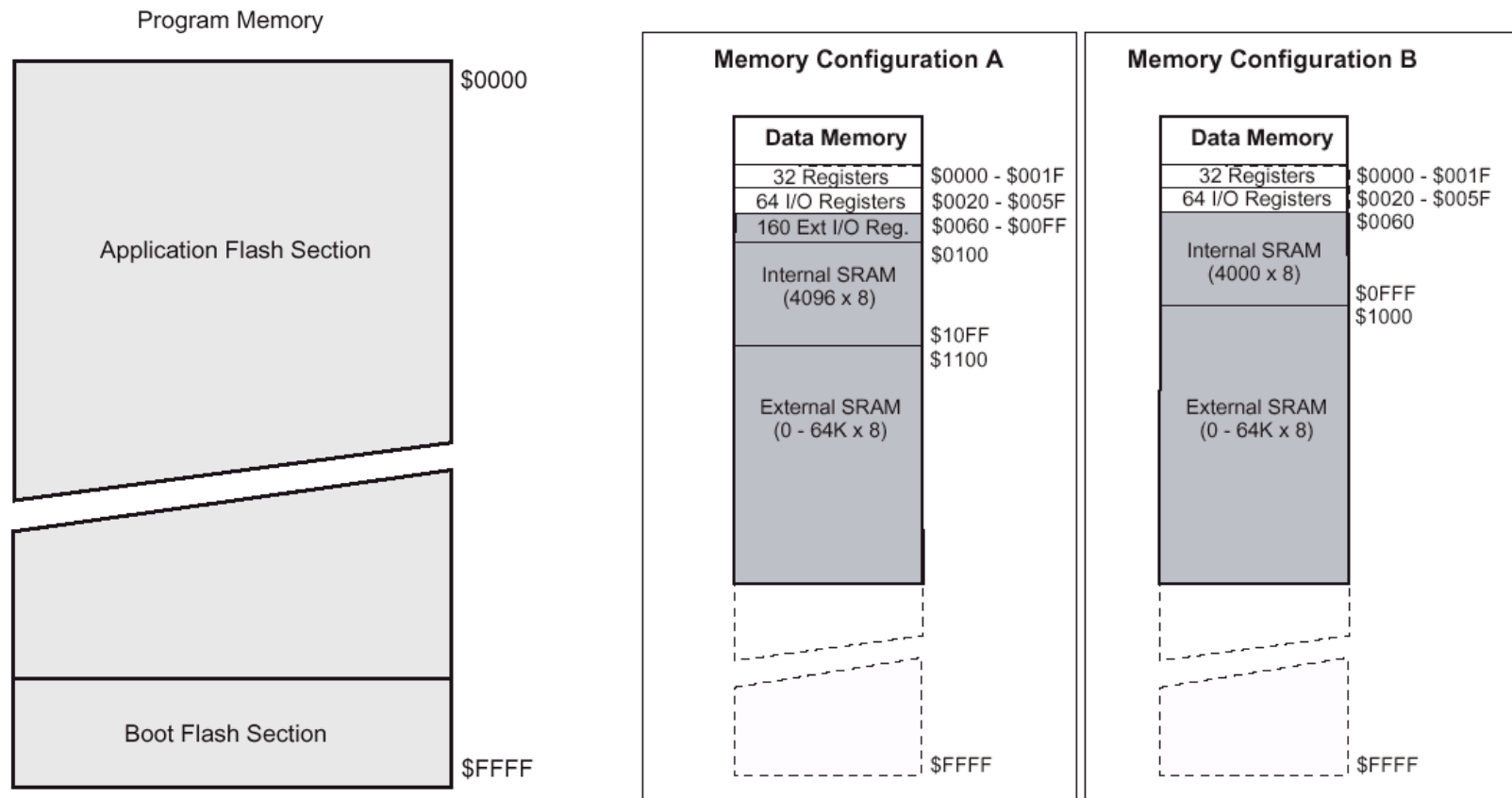
C: $Rd7 \cdot Rr7 + Rr7 \cdot \overline{Rd7} + \overline{Rd7} \cdot Rr7$

Set if there was carry from the MSB of the result; cleared otherwise.

R (Result) equals Rd after the operation.

Systemy Komputerowe

AVR (ATmega128) - przestrzeń pamięci



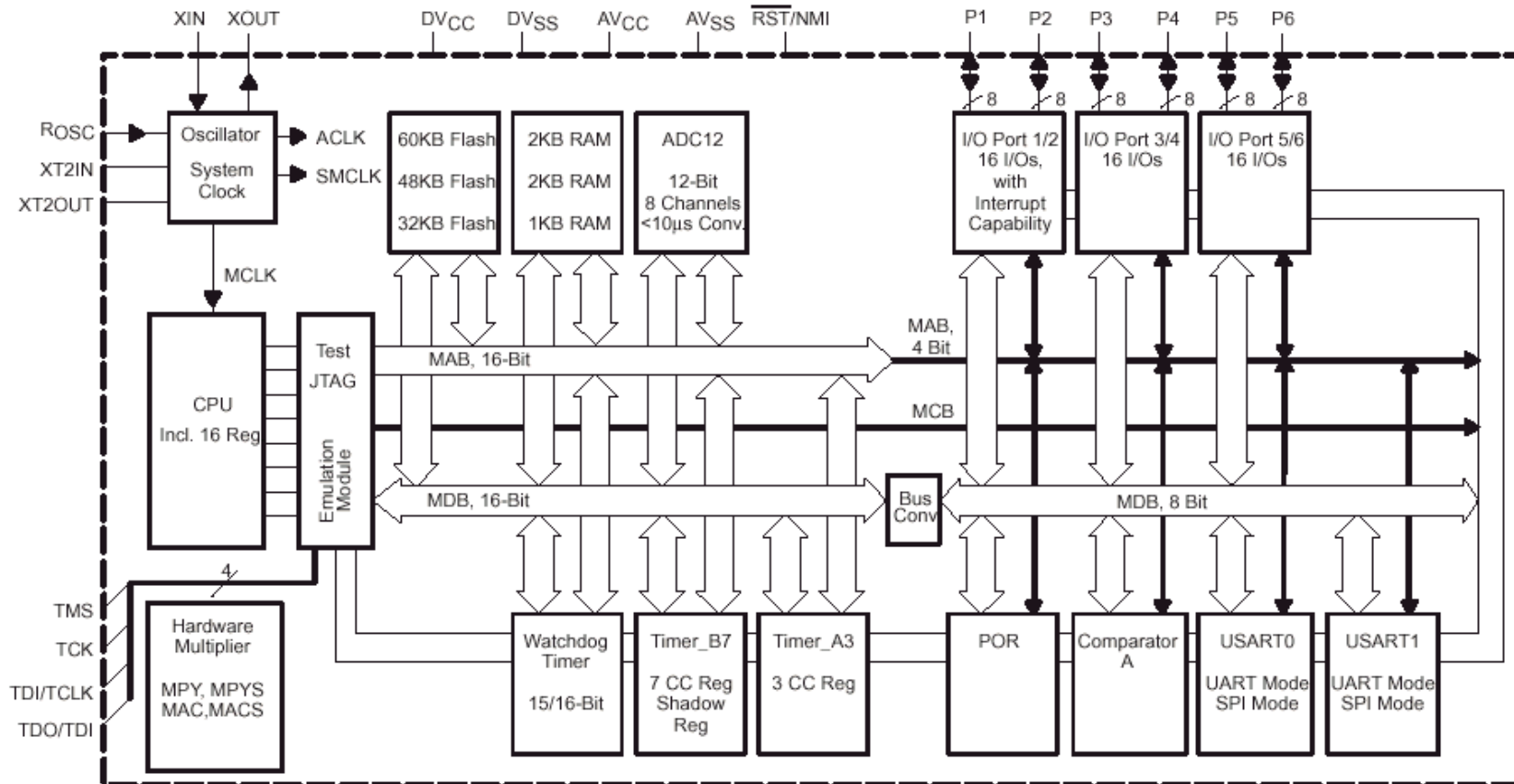
Systemy Komputerowe

■ AVR (ATmega128) - rejestry

	7	0	Addr.	
General Purpose Working Registers	R0		\$00	
	R1		\$01	
	R2		\$02	
	...			
	R13		\$0D	
	R14		\$0E	
	R15		\$0F	
	R16		\$10	
	R17		\$11	
	...			
	R26		\$1A	X-register Low Byte
	R27		\$1B	X-register High Byte
	R28		\$1C	Y-register Low Byte
	R29		\$1D	Y-register High Byte
	R30		\$1E	Z-register Low Byte
	R31		\$1F	Z-register High Byte

Systemy Komputerowe

■ MSP430 (MSP430F1xx) - ogólny widok wnętrza



Systemy Komputerowe

■ MSP430 (MSP430F1xx) - lista rozkazów

	000	040	080	0C0	100	140	180	1C0	200	240	280	2C0	300	340	380	3C0
0xxx																
4xxx																
8xxx																
Cxxx																
1xxx	RRC	RRC.B	SWPB		RRA	RRA.B	SXT		PUSH	PUSH.B	CALL		RETI			
14xx																
18xx																
1Cxx																
20xx	JNE/JNZ															
24xx	JEQ/JZ															
28xx	JNC															
2Cxx	JC															
30xx	JN															
34xx	JGE															
38xx	JL															
3Cxx	JMP															
4xxx	MOV, MOV.B															
5xxx	ADD, ADD.B															
6xxx	ADDC, ADDC.B															
7xxx	SUBC, SUBC.B															
8xxx	SUB, SUB.B															
9xxx	CMP, CMP.B															
Axxx	DADD, DADD.B															
Bxxx	BIT, BIT.B															
Cxxx	BIC, BIC.B															
Dxxx	BIS, BIS.B															
Exxx	XOR, XOR.B															
Fxxx	AND, AND.B															

Systemy Komputerowe

■ MSP430 (MSP430F1xx) - wycinek opisu jednej instrukcji

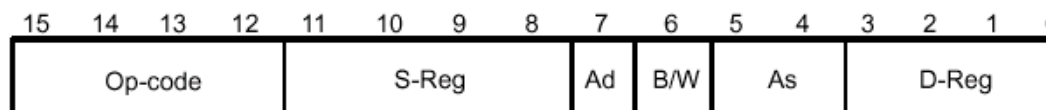


Table 3-11 lists and describes the double operand instructions.

Table 3-11. Double Operand Instructions

Mnemonic	S-Reg, D-Reg	Operation	Status Bits			
			V	N	Z	C
MOV(.B)	src, dst	src → dst	-	-	-	-
ADD(.B)	src, dst	src + dst → dst	*	*	*	*
ADDC(.B)	src, dst	src + dst + C → dst	*	*	*	*
SUB(.B)	src, dst	dst + .not.src + 1 → dst	*	*	*	*
SUBC(.B)	src, dst	dst + .not.src + C → dst	*	*	*	*
CMP(.B)	src, dst	dst - src	*	*	*	*
DADD(.B)	src, dst	src + dst + C → dst (decimally)	*	*	*	*
BIT(.B)	src, dst	src .and. dst	0	*	*	*
BIC(.B)	src, dst	.not.src .and. dst → dst	-	-	-	-
BIS(.B)	src, dst	src .or. dst → dst	-	-	-	-
XOR(.B)	src, dst	src .xor. dst → dst	*	*	*	*
AND(.B)	src, dst	src .and. dst → dst	0	*	*	*

Systemy Komputerowe

■ MSP430 (MSP430F1xx) - przestrzeń pamięci

		MSP430F133	MSP430F135	MSP430F147 MSP430F1471	MSP430F148 MSP430F1481	MSP430F149 MSP430F1491
Memory Main: interrupt vector Main: code memory	Size	8KB	16KB	32KB	48KB	60KB
	Flash	0FFFFh - 0FFE0h	0FFFFh - 0FFE0h	0FFFFh - 0FFE0h	0FFFFh - 0FFE0h	0FFFFh - 0FFE0h
Information memory	Flash	0FFFFh - 0E000h	0FFFFh - 0C000h	0FFFFh - 08000h	0FFFFh - 04000h	0FFFFh - 01100h
Boot memory	Size	256 Byte	256 Byte	256 Byte	256 Byte	256 Byte
	Flash	010FFh - 01000h	010FFh - 01000h	010FFh - 01000h	010FFh - 01000h	010FFh - 01000h
	Size	1KB	1KB	1KB	1KB	1KB
	ROM	0FFFh - 0C00h	0FFFh - 0C00h	0FFFh - 0C00h	0FFFh - 0C00h	0FFFh - 0C00h
RAM	Size	256 Byte	512 Byte	1KB	2KB	2KB
		02FFh - 0200h	03FFh - 0200h	05FFh - 0200h	09FFh - 0200h	09FFh - 0200h
Peripherals	16-bit	01FFh - 0100h	01FFh - 0100h	01FFh - 0100h	01FFh - 0100h	01FFh - 0100h
	8-bit	0FFh - 010h	0FFh - 010h	0FFh - 010h	0FFh - 010h	0FFh - 010h
	8-bit SFR	0Fh - 00h	0Fh - 00h	0Fh - 00h	0Fh - 00h	0Fh - 00h

Systemy Komputerowe

■ MSP430 (MSP430F1xx) - rejestry

Program Counter	PC/R0
Stack Pointer	SP/R1
Status Register	SR/CG1/R2
Constant Generator	CG2/R3
General-Purpose Register	R4
General-Purpose Register	R5
General-Purpose Register	R6
General-Purpose Register	R7
General-Purpose Register	R8
General-Purpose Register	R9
General-Purpose Register	R10
General-Purpose Register	R11
General-Purpose Register	R12
General-Purpose Register	R13
General-Purpose Register	R14
General-Purpose Register	R15

Systemy Komputerowe

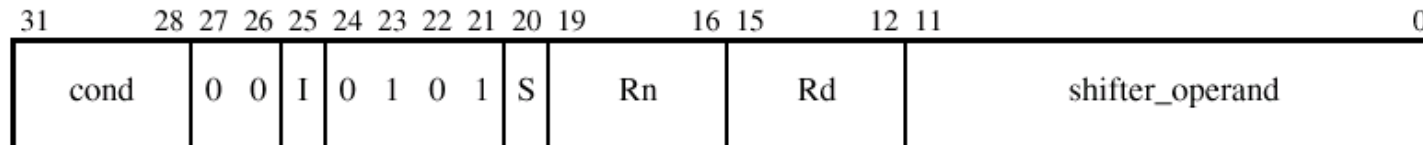
■ CPU z rdzeniem ARM - wycinek listy

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Data processing immediate shift	cond [1]	0	0	0	opcode			S	Rn				Rd				shift amount			shift	0	Rm											
Miscellaneous instructions: See Figure A3-4	cond [1]	0	0	0	1	0	x	x	0	x																0	x						
Data processing register shift [2]	cond [1]	0	0	0	opcode			S	Rn				Rd				Rs			0	shift	1	Rm										
Miscellaneous instructions: See Figure A3-4	cond [1]	0	0	0	1	0	x	x	0	x																0	x	x	1	x			
Multiplies: See Figure A3-3 Extra load/stores: See Figure A3-5	cond [1]	0	0	0	x				x				x				x				1	x	x	1	x								
Data processing immediate [2]	cond [1]	0	0	1	opcode			S	Rn				Rd				rotate			immediate													
Undefined instruction	cond [1]	0	0	1	1	0	x	0	0	x																							
Move immediate to status register	cond [1]	0	0	1	1	0	R	1	0	Mask				SBO				rotate			immediate												
Load/store immediate offset	cond [1]	0	1	0	P	U	B	W	L	Rn				Rd				immediate															
Load/store register offset	cond [1]	0	1	1	P	U	B	W	L	Rn				Rd				shift amount			shift	0	Rm										

Systemy Komputerowe

■ CPU z rdzeniem ARM - wycinek opisu jednej instrukcji

A4.1.2 ADC



ADC (Add with Carry) adds two values and the Carry flag. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the addition.

ADC can optionally update the condition code flags, based on the result.

Syntax

ADC{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

where:

<cond> Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If <cond> is omitted, the AL (always) condition is used.

S Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR. If S is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when S is specified:

- If <Rd> is not R15, the N and Z flags are set according to the result of the addition, and the C and V flags are set according to whether the addition generated a carry (unsigned overflow) and a signed overflow, respectively. The rest of the CPSR is unchanged.
- If <Rd> is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

**Czy trzeba programować w
assemblerze?**

Systemy Komputerowe

- **Jak dzisiaj tworzy się oprogramowanie współpracujące ze sprzętem?**
 - **Zasadniczo w C i coraz częściej w C++**
 - **Po co assembler?**
 - **Podstawa dla twórców kompilatorów**
 - **Gdy coś w kodzie działa nie tak**
 - **Gdy fragmenty kodu wygenerowanego z C/C++ działają za wolno**
 - **Gdy kod ma mieścić się w bardzo małych pamięciach**
 - **zagadnienie o źródłach ekonomicznych, MCU z wyliczeń finansowych ma kosztować mniej niż 1% kosztów produkcji całego urządzenia (zabawki, gadżety, ...)**
 - **Gdy chcemy sprawić aby nikt nie był w stanie zrozumieć kodu (;->>)**

Systemy Komputerowe

- Łączenie asemblera z C
 - Najczęściej łączy się C z asemblerem, np.:

```
#define BIT(n) PORTD=clrClkAndData;\nasm __volatile__ (\n    "sbrc %2," #n\n    "sbi 18,3"\n    "sbi 18,5"\n    "sbic 16,2"\n    "ori %0,1<<" #n\n    : "=d" (spiIn) : "0" (spiIn), \n    "r" (spiOut))\n\nuint8_t spi(uint8_t spiOut) {\n    uint8_t spiIn = 0;\n    uint8_t clrClkAndData;\n        BIT(7);\n        BIT(6);\n        ... \n        BIT(0); \n    return spiIn;\n}
```

Po pre-procesingu, kompilacji i linkowaniu:

```
//BIT(7)\nldi    r30, 0x32\nldi    r31, 0x00\nldd    r24, Y+1\nst     Z, r24\nldd    r24, Y+2\nldd    r25, Y+3\nsbrc   r25, 7\nsbi    0x12, 3 ;0x12=18\nsbi    0x12, 5 ;0x12=18\nsbic   0x10, 2 ;0x10=16\nori    r24, 0x80 ;0x80=1<<7\nstd    Y+2, r24\n\n//BIT(6)\nldi    r30, 0x32\nldi    r31, 0x00\nldd    r24, Y+1\nst     Z, r24\nldd    r24, Y+2\nldd    r25, Y+3\nsbrc   r25, 6\nsbi    0x12, 3\nsbi    0x12, 5\nsbic   0x10, 2\nori    r24, 0x40 ;0x40=1<<6\nstd    Y+2, r24
```

Dziękuję za uwagę