

## Systemy wbudowane

# Mikrokontrolery rodziny SAM7 – architektura, lista rozkazów, model pamięci

Dr inż. Wojciech Surtel

Skonstruowany całkowicie od podstaw w 1986 roku przez brytyjską firmę Acorn. **Acorn RISC Machine**, „przechrzczony” po kupieniu Acorna przez Olivetti na **Advanced RISC Machine**

Każda z instrukcji zawiera 4-bitowy kod, określający warunki jej wykonania, a także bit wskazujący na to, czy dana instrukcja może zmienić zawartość rejestru stanu procesora.

IF

Konstrukcja taka pozwala na eliminację wielu rozgałęzień programu, upraszcza znacznie logikę procesora, a równocześnie przyspiesza wykonanie programu dzięki zmniejszeniu objętości kodu.

Procesor **Strong ARM**, pracuje obecnie z zegarami w zakresie 100-300 MHz przy poborze mocy rzędu zaledwie 100 mW !!!

## Architektura RISC (Reduced Instruction Set Computers)

- ograniczona lista wykonywanych rozkazów,
- ograniczona ilość trybów adresowania,
- operacje wykonywane na rejestrach (brak rozkazów operujących na pamięci – poza rozkazami LOAD i STORE)
- operacje na danych w pamięci wykonywane są według schematu *Read-Modify-Write*,
- proste kody rozkazów -> uproszczenie dekodera rozkazów,
- prosta budowa rdzenia -> mniejsza ilość elementów -> zmniejszony pobór prądu.

## Rdzeń ARMv4T (ARM7)

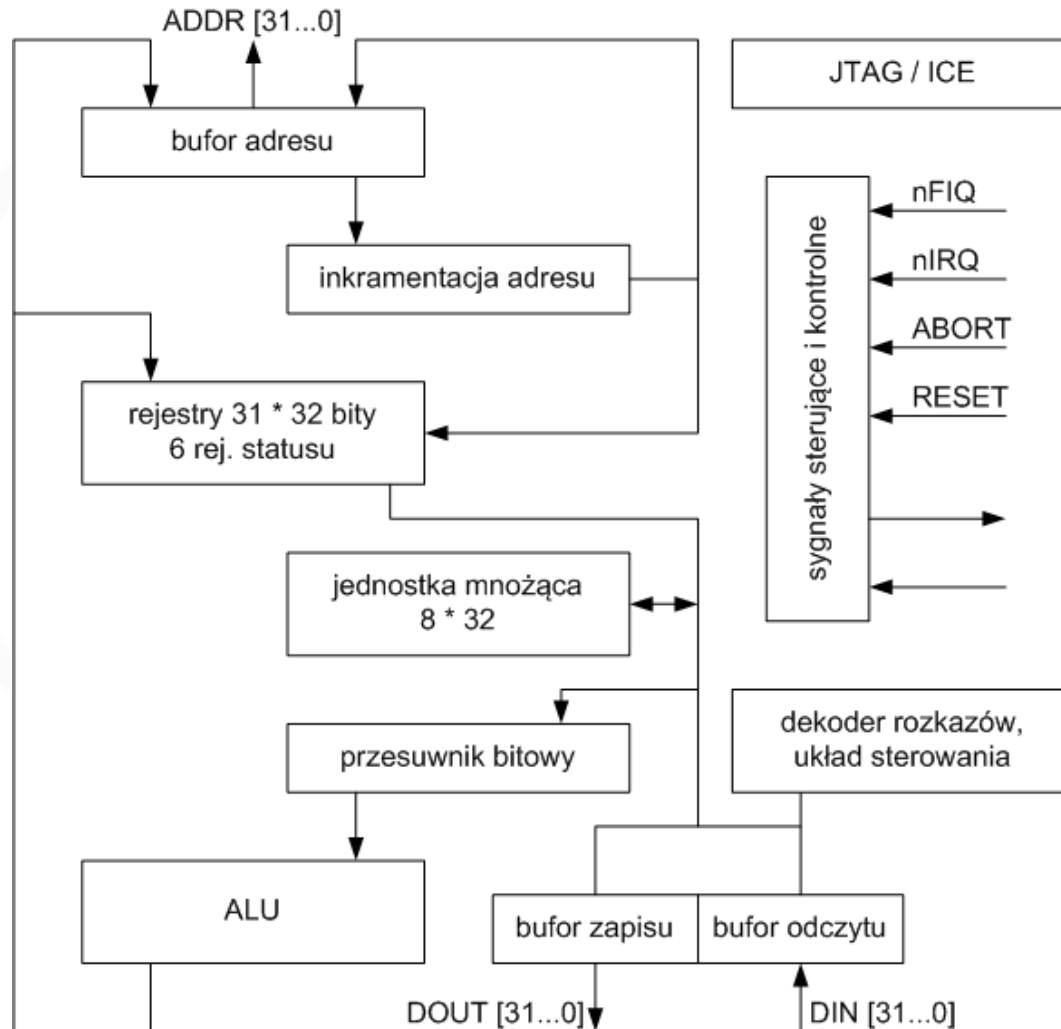
### Rejestry mikrokontrolerów RISC

- zwiększona liczba rejestrów roboczych,
- wszystkie rejestry są 32-bitowe.

### Architektura von Neumana

- brak podziału pamięci na pamięć danych i programu
- możliwość wykonywania kodu programu zarówno z pamięci Flash jak i RAM (możliwość modyfikacji kodu w **trakcie wykonywania programu**).
- 32-bitowa magistrala danych – możliwość jednoczesnego odczytu i zapisu danych, jednostkami 8, 16 lub 32-bitowymi (problem wyrównywania danych).

## Rdzeń ARMv4T (ARM7)



**Poglądowy  
schemat blokowy**

## Rdzeń ARMv4T (ARM7)

### Wyjątki obsługiwane przez rdzeń ARM7TDMI

- **SWI** – napotkanie specjalnej instrukcji SWI
- **Abort** – próba wykonania instrukcji, przy pobieraniu której wystąpił błąd w dostępie do pamięci lub próba zapisu/odczytu niewyrównanych danych,
- **Undefined** – napotkanie instrukcji, której rdzeń nie jest w stanie zdekodować,
- **nIRQ** – przerwanie – pojawienie się na linii wejściowej nIRQ stanu niskiego,
- **nFIQ** – przerwanie o wyższym priorytecie od nIRQ – wykorzystywane tam gdzie konieczna jest szybka reakcja,

### Tryby pracy rdzenia

- tryby zależne od rodzaju obsługiwanego wyjątku (np. po wystąpieniu wyjątku Abort rdzeń pracuje w trybie Abort),
- tryby pracy w czasie wykonywania programu głównego: Supervisor, System, User (różnią się poziomem praw dostępu do zasobów oraz obszarów pamięci mikrokontrolera),

## Rdzeń ARMv4T (ARM7)

### Tryby pracy rdzenia (z punktu widzenia instrukcji)

**tryb ARM** – rozkazy zakodowane na 32 bitach:

- dostępna pełna lista rozkazów,
- rozkazy mogą przyjmować większą liczbę parametrów,
- program złożony z instrukcji ARM działa szybciej,
- rozmiar programu jest większy.

**tryb Thumb** – rozkazy zakodowane na 16 bitach:

- program wykonywany jest wolniej ze względu na konieczność konwersji każdej instrukcji do pełnej instrukcji ARM przed jej wykonaniem,
- rozmiar programu jest mniejszy.

## Rdzeń ARMv4T (ARM7)

### Fazy wykonania rozkazu – praca potokowa

- każdy rozkaz wykonywany jest w trzech cyklach - pobranie, dekodowanie, wykonanie,
- przetwarzanie odbywa się z zastosowaniem potoku trójpoziomowego,
- efektywny czas wykonania rozkazu to jeden cykl.
















Cykl	Rozkaz 1	Rozkaz 2	Rozkaz 3
1	Pobranie		
2	Dekodowanie	Pobranie	
3	Wykonanie	Dekodowanie	Pobranie
4		Wykonanie	Dekodowanie
5			Wykonanie



## Rdzeń ARMv4T (ARM7)


### Rejestry dostępne w poszczególnych trybach

Rejestry ogólnego przeznaczenia

System and User	FIQ	Supervisor	Abort	IRQ	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	 r8_fiq	r8	r8	r8	r8
r9	 r9_fiq	r9	r9	r9	r9
r10	 r10_fiq	r10	r10	r10	r10
r11	 r11_fiq	r11	r11	r11	r11
r12	 r12_fiq	r12	r12	r12	r12
r13	 r13_fiq	 r13_svc	 r13_abt	 r13_irq	 r13_und
r14	 r14_fiq	 r14_svc	 r14_abt	 r14_irq	 r14_und
r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)

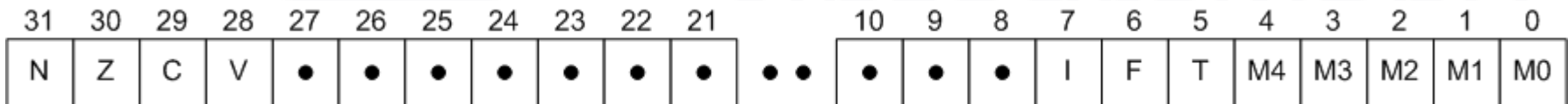
Rejestry statusowe

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	 SPSR_fiq	 SPSR_svc	 SPSR_abt	 SPSR_irq	 SPSR_und

 - rejestry bankowane

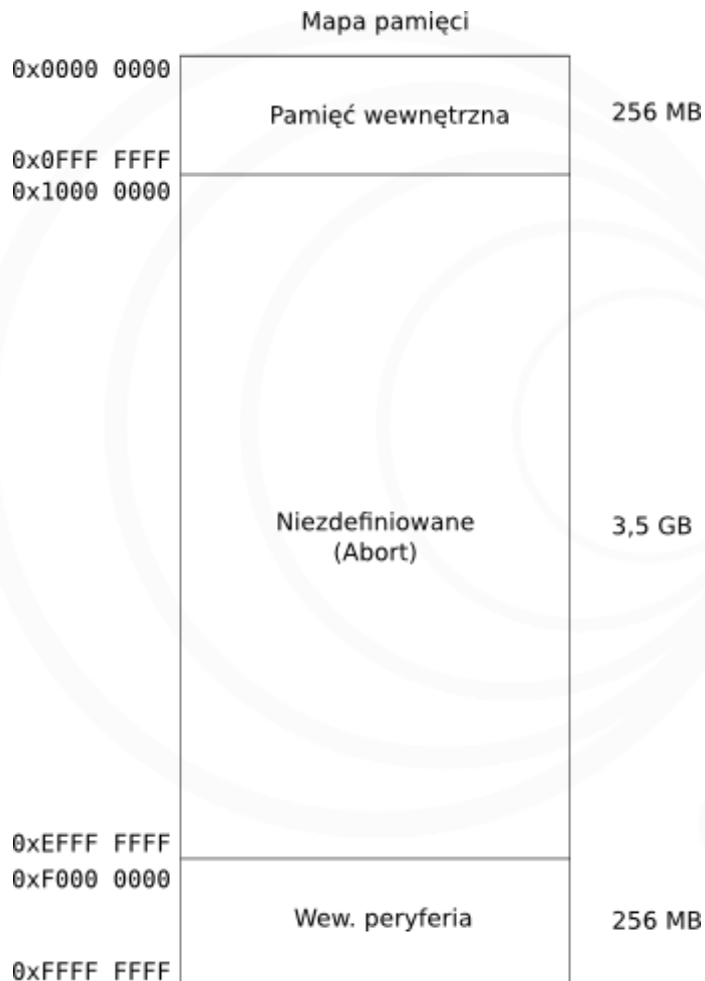
## Rdzeń ARMv4T (ARM7)

## Rejestr statusowy CPSR



Bit	Znaczenie
N	wynik ostatniego rozkazu jest ujemny
Z	wynik ostatniego rozkazu jest równy 0
C	w wyniku ostatniego rozkazu nastąpiło przeniesienie na najstarszym bicie (liczby bez znaku)
V	w wyniku ostatniego rozkazu nastąpiło przepełnienie (liczby ze znakiem)
I	blokada przerwania IRQ
F	blokada przerwania FIQ
T	praca w trybie Thumb
M0...M4	bieżący tryb operacyjny

## Rdzeń ARMv4T (ARM7)



## Przestrzeń adresowa ARM7TDMI

Cała przestrzeń adresowa rdzenia generalizując podzielona jest na 3 przestrzenie :

- 1) pamięć kodu/danych,
- 2) przestrzeń nie wykorzystana ,
- 3) rejestry sterujące urządzeniami peryferyjnymi.

## Rdzeń ARMv4T (ARM7)

Wewnętrzna pamięć 0-256MB

0x0000 0000 0x000F FFFF 0x0010 0000	Pamięć startowa. Flash ,ROM albo RAM*	1 MB
0x001F FFFF 0x0020 0000	Wew. Flash	1 MB
0x002F FFFF 0x0030 0000	Wew. SRAM	1 MB
0x003F FFFF 0x0040 0000	Wew. ROM	1 MB
0x00FF FFFF	Zarezerwowane	252 MB

**Pierwsze 256 MB przestrzeni adresowej ARM7TDMI**

## Rdzeń ARMv4T (ARM7)

### Organizacja pamięci rdzenia

Pierwszych 256MB pamięci rdzenia zawiera :

- Przestrzeń rozruchowa od 0x0000 0000 - do tej przestrzeni mapowana jest pamięć Flash, SRAM lub ROM.
- Pamięć Flash od 0x0010 0000 – pamięć NAND. Dla rdzenia AT91SAM7X256 jest 256kB, to 256kB jest „zawijane” w obrębie 1MB. Komórka o adresie 0x0010 001F to ta sama co 0x0014 001F, 0x0018 001F i 0x001C 001F.
- Pamięć SRAM od 0x0020 0000 – pamięć operacyjna. Szybka pamięć statyczna o rozmiarze 64kB. Ta pamięć jest mapowana pod adresy 0x0010 0000, 0x0011 0000, 0x0012 0000 itd.
- Pamięć ROM od 0x0030 0000 – pamięć tylko do odczytu. Dostarczane przez producenta.

## Mapa pamięci aplikacji

### Program zapisany do pamięci FLASH i stamtąd uruchamiany

DATA (0x00200000-0x0020FFFF),

CODE (0x00100000-0x0013FFFF),

CONST (0x00100000-0x0013FFFF)

Pamięć FLASH jest zawsze dostępna od adresu

0x00100000

Po resecie jest mapowana od adresu

0x00000000

## Mapa pamięci aplikacji

### Program załadowany do pamięci RAM przy użyciu JTAG

DATA (0x00208000-0x0020FFFF),

CODE (0x00200000-0x00207FFF),

CONST (0x00200000-0x00207FFF)

Pamięć RAM jest zawsze dostępna od adresu

0x00200000

Po resecie jest mapowana od adresu

0x00000000

### DEFINICJA STAŁYCH SYMBOLICZNYCH

Ustawienie bitów sterujących M4-M0

Mode\_USR EQU 0x10

Mode\_FIQ EQU 0x11

Mode\_IRQ EQU 0x12

Mode\_SVC EQU 0x13

Mode\_ABT EQU 0x17

Mode\_UND EQU 0x1B

Mode\_SYS EQU 0x1F



## DEFINICJA STAŁYCH SYMBOLICZNYCH

**Definicje stałych symbolicznych określających rozmiary stosów**

```
UND_STACK_SIZE    EQU    1*4
SVC_STACK_SIZE    EQU    1*4
ABT_STACK_SIZE    EQU    1*4
FIQ_STACK_SIZE    EQU    32*4
IRQ_STACK_SIZE    EQU    64*4
USR_STACK_SIZE    EQU    256*4
```

// Rezerwacja obszaru pamięci RAM przeznaczonego na stosy

AREA STACK, DATA, READWRITE, ALIGN=4

// ❶

DS USR\_STACK\_SIZE // alokacja pamięci na stos dla trybu USR/SYS

DS SVC\_STACK\_SIZE // alokacja pamięci na stos dla trybu SVC

DS IRQ\_STACK\_SIZE // alokacja pamięci na stos dla trybu IRQ

DS FIQ\_STACK\_SIZE // alokacja pamięci na stos dla trybu FIQ

DS ABT\_STACK\_SIZE // alokacja pamięci na stos dla trybu ABT

DS UND\_STACK\_SIZE // alokacja pamięci na stos dla trybu UND

Stack\_Base:

// etykieta

## Stos pamięci

// Inicjalizacja wskaźników stosów

```
LDR    R0, =Stack_Base
MSR    CPSR_c, #Mode_UND|I_Bit|F_Bit
MOV    SP, R0
SUB    R0, R0, #UND_STACK_SIZE
MSR    CPSR_c, #Mode_ABT|I_Bit|F_Bit
MOV    SP, R0
SUB    R0, R0, #ABT_STACK_SIZE
MSR    CPSR_c, #Mode_FIQ|I_Bit|F_Bit
MOV    SP, R0
SUB    R0, R0, #FIQ_STACK_SIZE
MSR    CPSR_c, #Mode_IRQ|I_Bit|F_Bit
MOV    SP, R0
SUB    R0, R0, #IRQ_STACK_SIZE
MSR    CPSR_c, #Mode_SVC|I_Bit|F_Bit
MOV    SP, R0
SUB    R0, R0, #SVC_STACK_SIZE
MSR    CPSR_c, #Mode_USR
MOV    SP, R0
```

// ⑥

```
// załadowanie adresu początku obszaru stosów
// wejście w tryb UND
// inicjalizacja wskaźnika stosu dla tego trybu
// korekta wartości R0 o ilość zarezerwowanych bajtów
// wejście w tryb ABT
// inicjalizacja wskaźnika stosu dla tego trybu
// wejście w tryb FIQ
// inicjalizacja wskaźnika stosu dla tego trybu
// wejście w tryb IRQ
// inicjalizacja wskaźnika stosu dla tego trybu
// wejście w tryb SVC
// inicjalizacja wskaźnika stosu dla tego trybu
// wejście w tryb USR
// inicjalizacja wskaźnika stosu dla tego trybu
```

## Rdzeń ARMv4T (ARM7)

## Kontroler przerwań

Przerwania są charakterystycznym elementem architektury mikroukładów. Model kontrolera przerwań omawianego układu można uprościć do dwu obwodów:

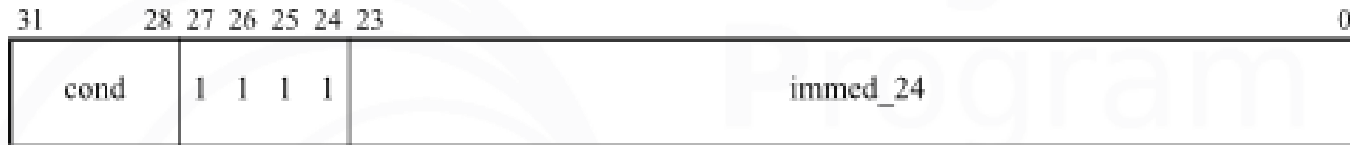
- Obwód żądania przerwania - odpowiada za to aby zgłosić zapotrzebowanie.
- Obwód obsługi przerwania – wystawia on odpowiedni adres obsługi procedury.

Wyjątek	Priorytet	Adres wektora	Tryb pracy rdzenia
Reset	1 (najwyższy)	0x0000 0000	Supervisor
Undefined Instruction	6	0x0000 0004	Undefined
Software Interrupt	6	0x0000 0008	Supervisor
Prefetch Abort	5	0x0000 000C	Abort
Data Abort	2	0x0000 0010	Abort
IRQ	4	0x0000 0018	IRQ
FIQ	3	0x0000 001C	FIQ

- Wyjątki IRQ oraz FIQ są zgłaszane przez urządzenia peryferyjne (zwane przerwaniem), natomiast źródłem pozostałych jest rdzeń SAM7.
- Aby mogło zaistnieć przerwanie – mając na myśli skok do adresu 0x18 lub 0x1C - musi zostać wyzerowany bit 6 (FIQ) i/lub bit 7 (IRQ).
- Są 32 linie przerwań, każda z linii (źródło) przerwań może mieć nadany priorytet w zakresie 0-7. Każdej z tych linii możliwe jest przyporządkowanie indywidualnej procedury obsługi.

## Rdzeń ARMv4T (ARM7)

### Przerwanie programowe



- Wyjątek *SWI (Software Interrupt)* można rozumieć jako „przerwanie wywołane w programie”.
- Rdzeń ARM, napotykając w kodzie maszynowym specjalną instrukcję SWI wywołującą ten wyjątek, przechodzi do jego obsługi.
- Wyjątek SWI zgłaszany może być np. przez program użytkownika pracujący pod kontrolą systemu operacyjnego w celu poinformowania o awarii lub zapotrzebowaniu na pewien zasób sprzętowy.
- Wyjątek SWI można bez większych problemów zgłaszać w programie pisanym w języku C - wystarczy umieścić w kodzie prościutką, jednolinijkową wstawkę asemblera zawierającą samą instrukcję o mnemoniku SWI.

### Rdzeń ARMv4T (ARM7)

### Przerwania współdzielone

Blok urządzeń systemowy (AT91C\_ID\_SYS) dysponuje jednym, wspólnym przerwaniem SYS (ang. shared interrupt) o numerze ID=1, które obejmuje następujące urządzenia:

- timery PIT, RTT, WDT,
- interfejs diagnostyczny DBGU,
- Sterownik DMA PMC,
- Układ zerowania procesora RSTC,
- Sterownik pamięci MC.

W procedurze obsługi przerwania SYS należy sprawdzić kolejno stan wszystkich urządzeń, czy występują przerwania odmaskowane. Jeżeli przerwanie jest aktywne należy sprawdzić flagę sygnalizującą przerwanie w rejestrze statusu danego urządzenia. Jeżeli flaga jest ustawiona należy wykonać program związany z obsługą przerwania od danego urządzenia.

# Mikrokontrolery rodziny SAM7 – lista rozkazów

## Zestawienie instrukcji:

Grupa	Mnemonic	Rozwinięcie	Opis
Arytmetyczne	ADD	add	dodawanie
	ADDC	add with carry	dodawanie z uwzględnieniem bitu <i>carry</i>
	SUB	subtract	odejmowanie
	SUBC	substr. with carry	odejmowanie z uwzględnieniem bitu <i>carry</i>
Logiczne	RSB	revers subtract	odejmowanie w odwrotnej kolejności
	RSC	revers subtract with carry	odejmowanie w odwrotnej kolejności z uwzględnieniem bitu <i>carry</i>
	CMP	compare	porównanie
	CMN	comp. negative	porównanie ze zmienionym znakiem arg2
	AND	and	iloczyn logiczny
	BIC	bit clear	zerowanie bitów
	ORR	or	suma logiczna
	EOR	exor	różnica symetryczna

# Mikrokontrolery rodziny SAM7 – lista rozkazów

## Zestawienie instrukcji:

Grupa	Mnemonik	Rozwinięcie	Opis
Log.	TST TEQ	test test equivalence	test test identyczności
Mnożenia	MUL MLA	multiply multiply – accumulate	mnożenie mnożenie z dodawaniem
	UMULL SMULL	unsigned multiply signed multiply	mnożenie bez znaku mnożenie ze znakiem
	UMLAL	unsigned multiply – accumulate	mnożenie z dodawaniem bez znaku
	SMLAL	signed multiply – accumulate	mnożenie z dodawaniem ze znakiem
Skoki	B BL BX	branch branch with link branch and exchange	rozgałęzienie (skok) rozgałęzienie (skok) z zachow. PC rozgałęzienie (skok) ze zmianą trybu ARM/Thumb

# Mikrokontrolery rodziny SAM7 – lista rozkazów

## Zestawienie instrukcji:

Grupa	Mnemonik	Rozwinięcie	Opis
Prześłań	MOV	move	przesłania i ładowania do rejestrów
	MVN	move not	j.w. z negacją
	LDR	load register	przesłanie z pamięci do rejestru
	STR	store register	przesłanie z rejestru do pamięci
	LDM	load multiply register	przesłanie z pamięci do wielu rej.
	STM	store multiply register	przesłanie z wielu rej. do pamięci
	SWP	swap register and memory	wymiana zawartości rej. i pamięci
	MRS	move xPSR to register	przesłanie do rej. statusowego z rej.
	MSR	move register to xPSR	przesłanie do rej. z rej. statusowego
Inne	SWI	software interrupt	przerwanie programowe

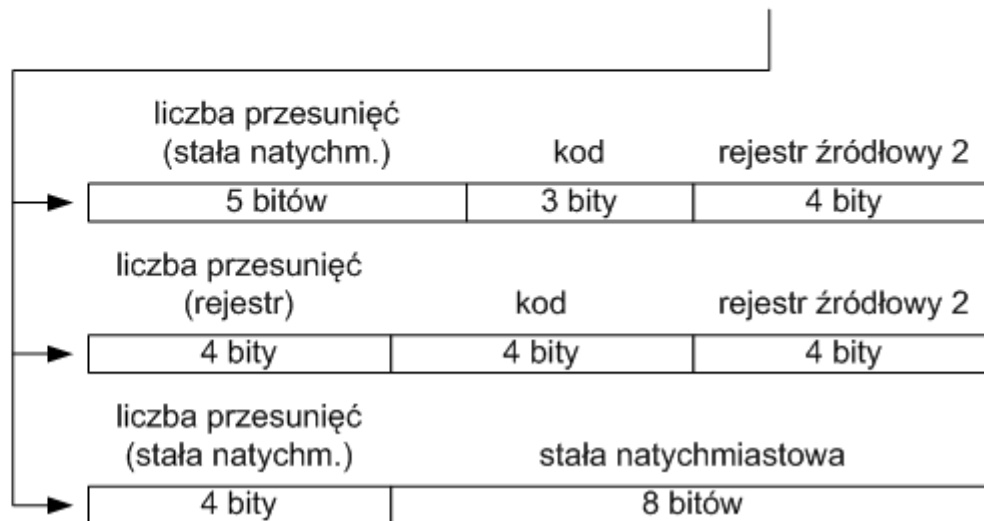


## Zestawienie instrukcji:

### Kodowanie rozkazów arytmetycznych i logicznych w trybie ARM:

- regularna budowa kodu – uproszczenie dekodera
- rozkazów -> zmniejszenie rozmiarów struktury

warunek	kod grupy rozkazów	kod rozkazu	uaktualnienie CPSR	rejestr docelowy	pierwszy operand	pole Operand_2
4 bity	3 bity	4 bity	1 bit	4 bity	4 bity	12 bitów



## Mikrokontrolery rodziny SAM7 – lista rozkazów

**Warunkowe wykonanie instrukcji (zależnie od stanu flag w rejestrze CPSR):**

Mnemonik	Rozwinięcie	Warunek	Stan flag
EQ	equal	równy	Z=1
NE	not equal	nie równy	Z=0
CS	carry set (unsigned higher or same)	ustawiona flaga przeniesienia; większy lub równy (liczby bez znaku)	C=1
CC	carry clear (unsigned lower)	wyzerowana flaga przeniesienia; mniejszy (liczby bez znaku)	C=0
MI	negative (minus)	ujemny	N=1
PL	positive or zero (plus)	dodatni lub zerowy	N=0
VS	overflow set	ustawiona flaga przepełn.	V=1
VC	overflow clear	wyzerowana flaga przepełn.	V=0

## Mikrokontrolery rodziny SAM7 – lista rozkazów

**Warunkowe wykonanie instrukcji (zależnie od stanu flag w rejestrze CPSR):**

Mnemonik	Rozwinięcie	Warunek	Stan flag
HI	unsigned higher	większy (liczby bez znaku)	C=1 and Z=0
LS	unsigned lower or same	mniejszy lub równy (liczby bez znaku)	C=0 or Z=1
GE	greater or equal	większy lub równy	N=V
LT	less then	mniejszy	N<>V
GT	greater then	większy	Z=0 and (N=V)
LE	less then or equal	mniejszy lub równy	Z=1 or (N<>V)
AL	always	zawsze (mnemonik można pominąć)	bez znaczenia

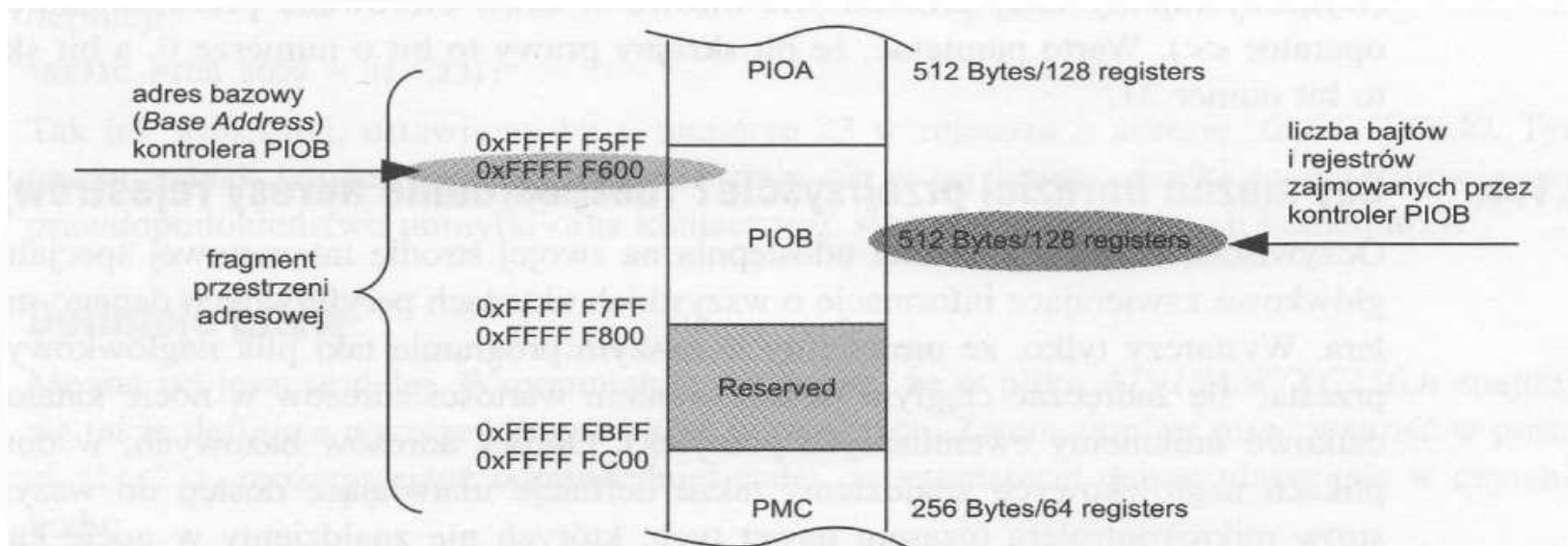
### Rdzeń ARMv4T (ARM7)

#### Kontroler PIO

- Rdzeń AT91SAM7X256/128 posiada dwa kontrolery PIO: PIOA i PIOB. Zadaniem każdego z kontrolerów jest multipleksowanie zestawu wyprowadzeń urządzeń peryferyjnych.
- Każdy kontroler steruje 32 liniami. Każda linia może być przypisana do jednego z dwu urządzeń peryferyjnych A lub B. Lub też działać jako uniwersalne wyjście/wejście. Na każdej linii

# Mikrokontrolery rodziny SAM7 – architektura

## Bezpośredni wpis do pamięci



**Zespół układów peryferyjnych System Controller (SYSC) czyli układów najbardziej podstawowych dla mikrokontrolera (w skład SYSC wchodzi także kontroler PIO).**

**Np.: rejestry konfiguracyjne kontrolera PIOB znajdują się pomiędzy adresami 0xFFFFF600 a 0xFFFFF7FF.**

# Mikrokontrolery rodziny SAM7 – architektura

## Bezpośredni wpis do pamięci

Offset	Register	Name	Access	Reset Value
0x0000	PIO Enable Register	PIO_PER	Write-only	–
0x0004	PIO Disable Register	PIO_PDR	Write-only	–
0x0008	PIO Status Register	PIO_PSR	Read-only	0x0000 0000
0x000C	Reserved			
0x0010	Output Enable Register	PIO_OER	Write-only	–
0x0014	Output Disable Register	PIO_ODR	Write-only	–
0x0018	Output Status Register	PIO_OSR	Read-only	0x0000 0000
0x001C	Reserved			
0x0020	Glitch Input Filter Enable Register	PIO_IFER	Write-only	–
0x0024	Glitch Input Filter Disable Register	PIO_IFDR	Write-only	–
0x0028	Glitch Input Filter Status Register	PIO_IFSR	Read-only	0x0000 0000
0x002C	Reserved			
0x0030	Set Output Data Register	PIO_SODR	Write-only	–
0x0034	Clear Output Data Register	PIO_CODR	Write-only	–
0x0038	Output Data Status Register	PIO_ODSR	Read-only	0x0000 0000

sposób dostępu do rejestru

adres rejestru  
liczony od adresu  
bazowego

pełna nazwa  
rejestru

skrótowa nazwa  
rejestru (zazwyczaj  
taka, jakiej użyjemy  
w kodzie programu)

wartość po  
wykonaniu  
zerowania

interesujący nas rejestr  
umożliwiający ustawienie wyjścia  
w stan wysoki

Jak widzimy, w tabeli nie zostały podane adresy fizyczne każdego rejestru, lecz jedynie „odległości” rejestrów od adresu bazowego.

Wspomniane „odległości” w tabeli nazwane zostały Offset, czyli „przesunięcie”.

W jednym mikrokontrolerze mogą być różne kontrolery PIO, mogą znajdować się one pod różnymi adresami.

Pod względem budowy i działania nie będą się one niczym między sobą różniły - będą jedynie umieszczone pod innymi adresami bazowymi

# Mikrokontrolery rodziny SAM7 – architektura

## Zastosowanie adresów bazowych

- Aby uniezależnić się od adresu bazowego układu peryferyjnego, który będziemy chcieli oprogramować, musimy jawnie podać adres bazowy modułu, o który nam chodzi, za pomocą stałej lub zmiennej odpowiedniego typu.
- W ramach realizacji przykładu, docelowo napiszemy funkcję ustawiającą 8 najmłodszych bitów zadanego kontrolera PIO.
- Funkcja ma działać z każdym kontrolerem PIO w każdym mikrokontrolerze typu SAM7.
- Korzystamy z gotowych definicji z pliku AT91SAM7XC256.h:

```
#define AT91C_BASE_PIOA      ((AT91PS_PIO) 0xFFFFF400)
#define AT91C_BASE_PIOB      ((AT91PS_PIO) 0xFFFFF600)
```

# Mikrokontrolery rodziny SAM7 – architektura

## Zastosowanie adresów bazowych

- Od adresu bazowego (np. *AT91C\_BASE\_PIOB*) musimy jakoś odliczyć kilka rejestrów aż do interesującego nas rejestru *PIO\_SODR*.
- Innymi słowy, musimy jakoś zadać offset, czyli przesunięcie danego rejestru od adresu bazowego.
- Pamiętamy, jak działa się to w przypadku kontrolera PIO: dysponowaliśmy adresem bazowym kontrolera PIOB (*0xFFFFF600*) oraz przesunięciem, jakie należało dodać do adresu bazowego, aby odnieść się do rejestru *PIO\_SODR* (przesunięcie to wynosiło *0x30*).
- Gdy zsumowaliśmy te wartości, otrzymaliśmy adres rejestru *PIO\_SODR* kontrolera PIOB.



# Mikrokontrolery rodziny SAM7 – architektura

## Zastosowanie adresów bazowych

Jak zautomatyzować proces odnajdowania interesującego rejestru, mając dany adres bazowy.

Pierwsze, co może przyjść na myśl, to zastosowanie wskaźnika ustawionego na adres bazowy, a następnie posługiwanie się nim jak tablicą (indeksowanie).

Można to zrealizować na przykład tak:

```
//definicja wskaźnika, który będzie adresem bazowym
```

```
uint32_t *pioBase;
```

```
//ustawienie wskaźnika na adresie bazowym PIOB
```

```
pioBase = (uint32_t*)AT91C_BASE_PIOB;
```

```
//ustawienie 8 najmłodszych bitów w stan wysoki
```

```
pioBase[0x0C] = 0xFF;
```

# Mikrokontrolery rodziny SAM7 – architektura

## Zastosowanie adresów bazowych

Adresy bazowe kontrolerów PIOA i PIOB rzutowane są w swoich definicjach do typu *AT91PS\_PIO*.

Od tego momentu, jeśli utworzymy zmienną lub stałą typu *AT91PS\_PIO*, to utworzymy wskaźnik do struktury opisującej kontroler PIO. Dysponując wskaźnikiem do struktury danych, odnosimy się do jej elementów za pomocą operatora `->`.

Definicja adresu bazowego kontrolera PIO

```
#define AT91C_BASE_PIOA      ((AT91PS_PIO) 0xFFFFF400)
```

jest typu *AT91PS\_PIO*, czyli jest wskaźnikiem do struktury danych opisującej rejestry PIO, możemy zastosować następujący zapis: `//ustawiamy 8 najmłodszych bitów na "1"`

```
AT91C_BASE_PIOA->PIO_SODR = 0xFF;
```

# Mikrokontrolery rodziny SAM7 – architektura

## Zastosowanie adresów bazowych

- **Możemy odnosić się do rejestrów za pomocą kombinacji adresu bazowego i offsetu.** Teraz wystarczy, zamiast stałego adresu bazowego `AT91C_BASE_PIOA`, podstawić zmienną - konkretnie wskaźnik o takim samym typie jak `AT91C_BASE_PIOA`, czyli typu `AT91PS_PIO`:

**//utworzenie zmiennej pPio będącej wskaźnikiem do dowolnego PIO AT91PSPIO pPio;**

**//ustawienie pPio na adres bazowy PIOA**

**t\_pPio = AT91C\_BASE\_PIOA;**

**//ustawienie 8 najmłodszych bitów w PIOA**

**t\_pPio->PIO\_SODR = 0xFF;**

**//ustawienie pPio tym razem na adresie bazowym PIOB**

**t\_pPio = AT91C\_BASE\_PIOB;**

**//8 najmłodszych bitów ustawiliśmy w PIOB**

**t\_pPio->PIO\_SODR = 0xFF;**

- W powyższym fragmencie programu użyliśmy zmiennej `t_pPio` raz, by za jej pomocą dokonać operacji na kontrolerze PIOA, i raz, by zrobić to samo na kontrolerze PIOB.

## Rdzeń ARMv4T (ARM7)

## Kontroler PIO – przykład: konfiguracja, tryby adresowania

// GPIO init

```
t_pPioA->PIO_ODR  = 0xffffffff;    // All as input
t_pPioB->PIO_ODR  = 0xffffffff;    // All as input
t_pSys->PIOA_PPUDR = 0xffffffff;    // Disable Pull-up resistor
t_pSys->PIOB_PPUDR = 0xffffffff;    // Disable Pull-up resistor
```

// all as GPIO

```
t_pPioA->PIO_PER = 0xffffffff; //Disables the PIO from controlling the corresponding pin (enables
                                peripheral control of the pin)

t_pPioA->PIO_ASR = 0;           //Assigns the I/O line to the peripheral B function
t_pPioA->PIO_BSR = 0;

t_pPioB->PIO_PER = 0xffffffff; //Disables the PIO from controlling the corresponding pin (enables
                                peripheral control of the pin)

t_pPioB->PIO_ASR = 0;           //Assigns the I/O line to the peripheral B function
t_pPioB->PIO_BSR = 0;
```

## Rdzeń ARMv4T (ARM7)

```
// all as input
```

```
t_pPioA->PIO_ODR = 0xffffffff;
```

```
t_pPioB->PIO_ODR = 0xffffffff;
```

```
// pull up
```

```
t_pPioA->PIO_SODR = BIT3;
```

```
t_pPioA->PIO_OER = BIT3;
```

```
// check for low on port A
```

```
if((t_pPioA->PIO_PDSR | (~mask_port_a)) != 0xFFFFFFFF) {
```

```
....
```

```
}
```

```
// check for low on port b
```

```
if((t_pPioB->PIO_PDSR | (~mask_port_b)) != 0xFFFFFFFF) {
```

```
...
```

```
}
```

## Kontroler PIO – przykład: konfiguracja, tryby adresowania

```
// All as input
```

```
// All as input
```

```
// high
```

```
// output
```

# Mikrokontrolery rodziny SAM7 – architektura

## Bezpośrednie adresy rejestrów

- Atmel udostępnia na swojej stronie internetowej specjalne **pliki nagłówkowe** zawierające informacje o wszystkich układach peryferyjnych danego mikrokontrolera.
- Wystarczy tylko, że umieścimy w naszym programie taki plik nagłówkowy i nie musimy ciągle poszukiwać wartości adresów w nocie katalogowej (dodatkowo unikniemy ewentualnych pomyłek).
- Oprócz adresów bazowych, w dostarczonych plikach nagłówkowych znajdziemy także definicje ułatwiające dostęp do wszystkich rejestrów mikrokontrolera (czasem nawet tych, których nie znajdziemy w nocie katalogowej) oraz definicje masek bitowych.
- Dzięki maskom będziemy mogli stosować nazwy bitów lub pól bitowych w rejestrach zazwyczaj takie, jakie odczytamy z noty katalogowej.
- Sporadycznie zdarzają się rozbieżności w nazwach rejestrów z pliku nagłówkowego oraz z noty.
- Teraz prześledzimy, jak można zapisać kod programu ustawiający „nóżkę” PB23 mikrokontrolera w stan wysoki **za pomocą gotowej makrodefinicji** rejestru PIO\_SODR.
- Zaczniemy od otwarcia pliku nagłówkowego *AT91SAM7XC256.h*

# Mikrokontrolery rodziny SAM7 – architektura

## Bezpośrednie adresy rejestrów

- Najprostszym sposobem ustawienia stanu wysokiego na pinie PB23 może być zastosowanie gotowego, ustawionego już i stałego wskaźnika do rejestru PIO\_SODR kontrolera PIOB.
- Definicja tego wskaźnika znajduje się w omawianym pliku nagłówkowym w linii 1967. Jak widać, plik nagłówkowy *AT91SAM7XC256.h* składa się z bardzo wielu linii - w końcu każdy układ peryferyjny, każdy rejestr i nawet każdy bit ma swoją, odpowiadającą mu definicję.
- Znaleziona definicja rejestru PIO\_SODR to  

```
#define AT91C_PIOB_SODR ((AT91_REG *) 0xFFFFF630)
```
- *AT91C\_PIOB\_SODR* jest wskaźnikiem typu *AT91\_REG* ustawionym od razu na wartość *0xFFFFF630*, czyli dokładnie tyle, ile wyliczyliśmy w poprzednim podpunkcie, opierając się na mapie pamięci SAM7XC oraz tabeli rejestrów kontrolera PIO.
- Korzystamy ze wskaźnika typu *AT91\_REG\**, a nie napiszemy po prostu *uint32\_t\** przede wszystkim dla porządku i przejrzystości kodu. Jeśli popatrzymy na linię 49 pliku nagłówkowego, zobaczymy definicję typu *AT91\_REG*;

```
typedef volatile unsigned int AT91_REG;
```

# Mikrokontrolery rodziny SAM7 – architektura

## Bezpośrednie adresy rejestrów

- Ponieważ posługujemy się także plikiem nagłówkowym *inttypes.h*, moglibyśmy tę definicję zapisać po swojemu jako:

```
typedef volatile uint32_t AT91_REG;
```

- czyli „od teraz volatile uint32\_t nazywamy AT91\_REG”, i wyszłoby praktycznie na to samo, ponieważ unsigned int odpowiada w przypadku mikrokontrolerów ARM dokładnie jednemu 32-bitowemu słowu maszynowemu bez znaku tak samo jak uint32\_t.
- Oczywiście typu AT91\_REG możemy używać we własnym programie identycznie jak każdego innego typu, np. do przekazywania rejestrów jako argumentów do funkcji.
- **Dzięki temu typowi możemy uniknąć pomyłki polegającej choćby na nieświadomym wywołaniu takiej funkcji z argumentem niebędącym rejestrem zadany makrodefinicją z pliku nagłówkowego, lecz liczbą o przypadkowej wartości.**



# Mikrokontrolery rodziny SAM7 – architektura

## Bezpośrednie adresy rejestrów

- Do rejestru PIO\_SODR układu PIOB można odnieść się za pomocą nowo poznanej definicji:

```
*AT91C_PIOB_SODR = (1<<23);
```

- W pliku *AT91SAM7XC256.h* znajdują się także definicje poszczególnych bitów w rejestrach.
- Zatem zamiast pisać wartość w postaci  $(1 \ll 23)$ , możemy użyć odpowiedniej stałej, zapewniającej dalsze ułatwienie w czytaniu kodu:

```
*AT91C_PIOB_SODR = AT91C_PIO_PB23;
```

# Mikrokontrolery rodziny SAM7 – architektura

## Bezpośrednie adresy rejestrów

Treść definicji bitu PB23 dla rejestrów PIO jest dość oczywista (linijka 2585 pliku nagłówkowego):

```
#define AT91C_PIO_PB23 ((unsigned int) 1 << 23)
```

Niektórzy mogą zwrócić teraz uwagę na fakt, że stosowanie dłuższej nazwy definicji (*AT91C\_PIO\_PB23*) niż liczby, którą ta definicja stanowi ( $(1 \ll 23)$ ), może mijać się z celem.

Konsekwentne **stosowanie makrodefinicji przy dokonywaniu wpisów do rejestrów mikrokontrolera** wbrew pozorom nie jest pozbawione sensu, ponieważ stosując powyższy zapis, od razu wiemy, jaki bit ustawiamy.

Materiały zostały opracowane w ramach projektu  
*„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”*,  
umowa nr **POWR.03.05.00-00-Z060/18-00**  
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020  
współfinansowanego ze środków Europejskiego Funduszu Społecznego