Ćwiczenie 1

Zaawansowane architektury komputerowe

Projektowanie mikrooperacji procesora RISC/DLX(MIPS)

1 Wprowadzenie

W trakcie bieżącego ćwiczenia student pozna podstawowe zasady kierujące pracą inżyniera projektanta mikroprocesorów. W szczególności zapozna się z mikrooperacjami procesora RISC/DLX. Głównym celem ćwiczenia jest zapoznanie się z niższą niż poziom asemblera warstwą wykonywania aplikacji oraz zwrócenie uwagi na technologie realizacji operacji z wykorzystaniem instrukcji typu RISC, ale na procesorach typu CISC (np. platformy Intel, chociaż w tym przypadku chodzi o inny prostszy architektonicznie procesor MIPS/ARM). Laboratorium obejmuje implementację prostego filtru FIR (SOI) na maszynie o prostej architekturze RISC. Filtr będzie analizowany, wdrażany i modyfikowany w celu usprawnienia i uproszczenia realizacji (architektury procesora).

.

1.1 Uruchomienie symulatora system mikroarchitektonicznego ESCAPE

- Rozpocznij od ściągnięcia ćwiczenia ze wskazanej strony
- Rozpakuj .zip w ustalonym wcześniej miejscu – katalogu roboczym.
- Uruchom symulator – znajduje się w pakiecie

2 Podstawowa Konfiguracja (założenia architektoniczne procesora)

Konfigurację symulatora najłatwiej przeprowadzić wczytując przykładową konfigurację dla zakładki Configurationi →Load simple.ecf, a następnie dokonać modyfikacji. Otwórz zakładkę ***instruction encoding*** (zak. 1). Istnieje tam 3 typy mikrooperacji, mikrooperacje te zostaną wykorzystane do implementacji mikroinstrukcji asemblerowych.

.

2.1 Parametry formalne mikrooperacji

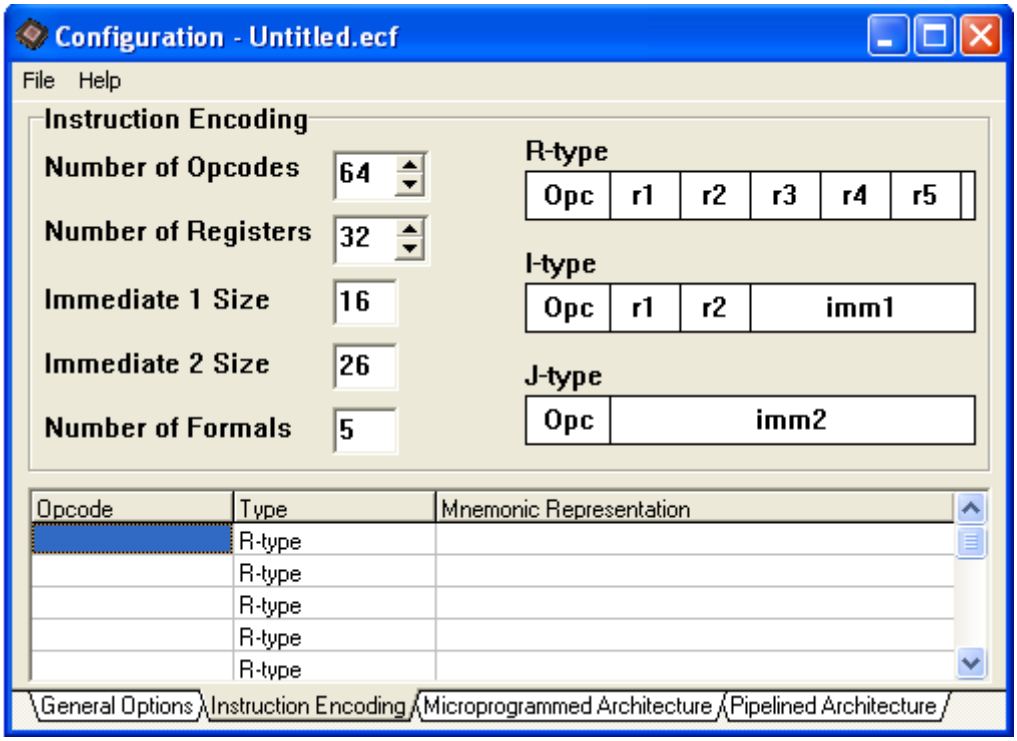
Instrukcja asemblerowa może posiadać 3 typy parametrów: rejestrowe, natychmiastowe oraz etykietowane. Parametry formalne wykorzystywane są przy deklarowaniu instrukcji, ale rzeczywiste wartości są przyporządkowywane rzeczywistym rejestrom zob. Tabela 2.

Mikrokod wykorzystuje tę samą zasadę, która jest obecna podczas implementacji kodu asemblerowego, jego parametry formalne zastępowane są przez rzeczywiste rejestry. To jest odpowiednik sparametryzowanej funkcji– dlaczego?

Tabela 1: Formalne i rzeczywiste parametry - korelacja

	Formalne	Rzeczywiste
Rejestry		
przyklad znaczenie rejestr	ADDreg r1, r2, r3 Mikrokod instrukcja ADD r1	ADDreg R4, R3, R6 R6 = R4 + R3 R4

natychmiastowe		
przyklad znaczenie wartość operandu	PUTimm r1, i mikrokod instukcja PUT i	PUTimm R3, 200 R3 załaduj 200 200
Etykieta		
przyklad znaczenie etykieta	JMPrel j Skok JMP J	JMPrel 200 PC PC + 200 200



Rys. 1:Zakładka Encoding window

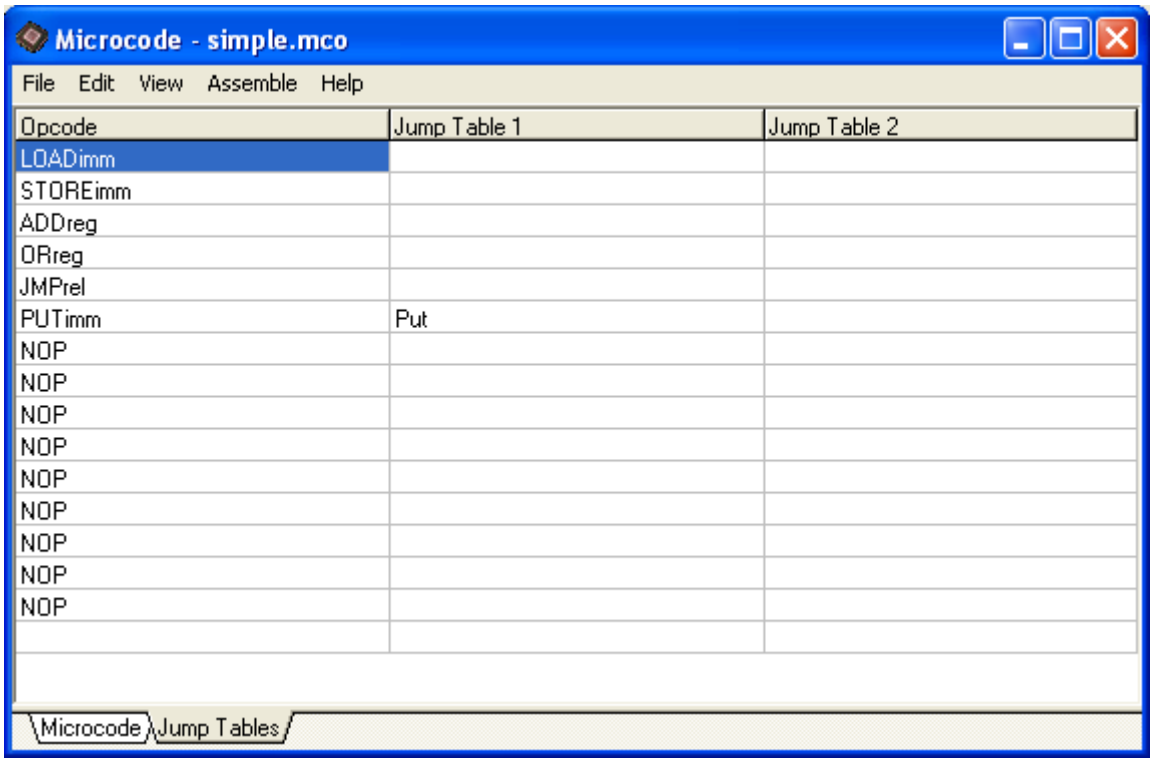
Tabela 1: Przyklad zestawu instrukcji

Opcode	Type		
LOADimm	I	r1, i	znaczenie kodu operacyjnego
STOREimm	I	r1, r2, r3	załaduj spod adresu i do rejestru r1
ADDreg	R	r3 = r1 + r2	zachowaj pod adresem i wartość rejestru r1
ORreg	R	r3 = r1 | r2	(operacja LUB)
JMPrel	J	j	skocz pod adres i etykietę j
PUTimm	I	r1, i	zachowaj wartość i w rejestrze r1
NOP	R		pusta instrukcja

Ponizszy program (pliki ***simple.****) wykonuje prostą operację sumowania. Nie można tych operacji wykonać dopóki nie zostanie zaimplementowany zestaw mikrooperacji.

PUTimm R1, 0x0000C8
PUTimm R2, 0x00012C
ADDreg R1, R2, R3
STOREimm R3, 0x000064
LOADimm R2, 0x000064
JMPrel l1

Zaimplementowana jest jedna mikrooperacja kodu operacyjnego PUTimm i dostępna jest w zadaniu.



Rys. 2 Okno projektowania mikrooperacji

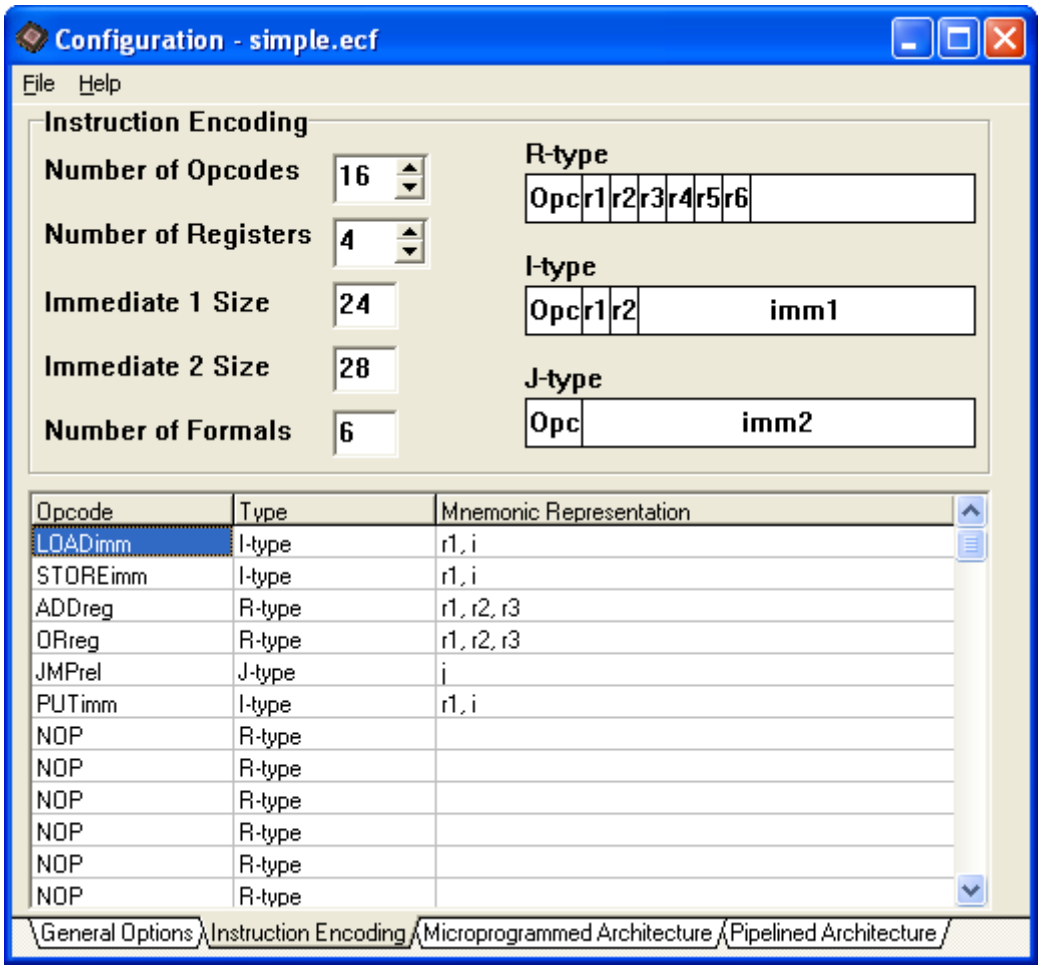
Pierwsze instrukcje zostały zaimplementowane, zob. Rys. 4. *Start* polega na wczytaniu z pamięci MEM słowa RW o adresie MAdr PC, pod rejestr MDest IR. W istocie jest to sekwencja pobierająca rozkaz z pamięci. W kolejnej mikrooperacji inkrementowany jest licznik rozkazów i skok do obsługi mikroinstrukcji. Po tych instrukcjach na pozycji 0004pojawia mikrooperacja *add*. Całość mikrooperacji implementujących kod asemblerowy, jest realizowana poprzez pobieranie instrukcji, a następnie wykonywanie skoku do kodu w tablicy Jump1. Tablica asocjacyjna Jump Tables (od jej implementacji należy zacząć) w efekcie mówi o tym, jaki powinien być następny adres uAR, ponieważ kojarzy ona adres mikroinstrukcji asemblerowej z etykietą zapisaną w swojej kolumnie np. PUTI (rozpoczynające program, zob. wyżej) będzie wykonane przez Put , uAR 0002.

Listing gotowego rozwiązania, dla powyższego przykładu, znajduje się poniżej (można go wprowadzić do programu zob. Rys.4 i uruchomić):

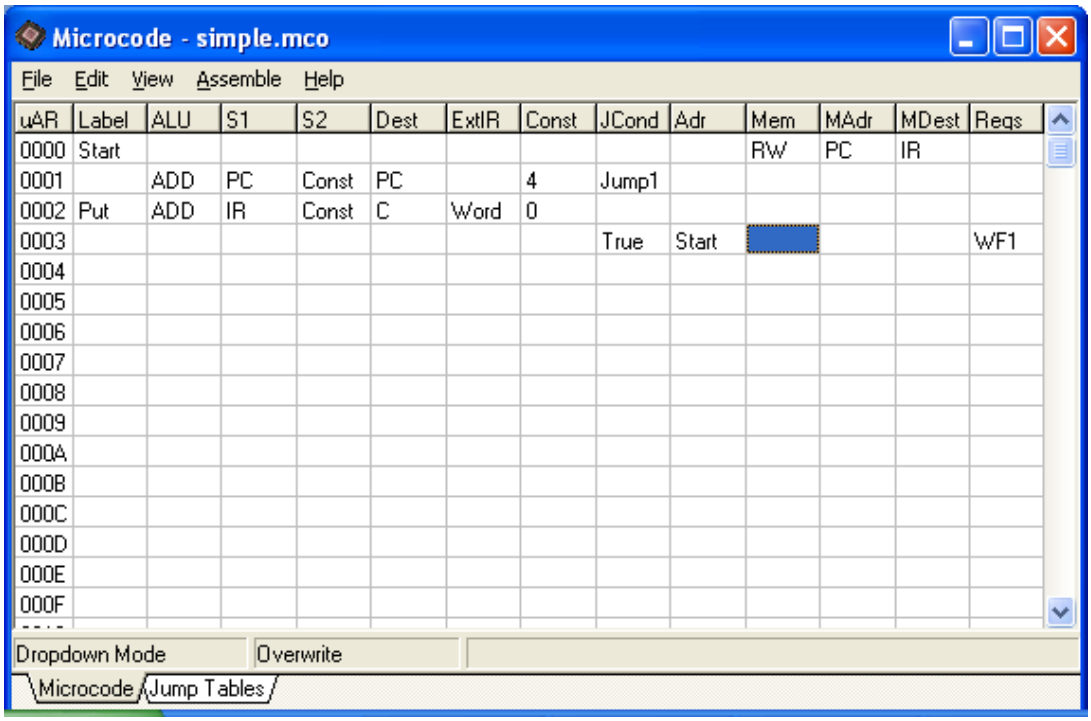
```
0000 |Start |     |     |     |     |     |     |     |RW   |PC   |IR   |
0001 |     |ADD  |PC   |Const |PC   |     |4    |Jump1 |     |     |     |     |
0002 |Put  |ADD  |IR   |Const |C    |Word  |0    |     |     |     |     |     |
0003 |     |     |     |     |     |     |     |True  |Start |     |     |     |WF1
0004 |add  |     |     |     |     |     |     |     |     |     |     |     |RR
0005 |     |ADD  |A    |B    |C    |     |     |     |     |     |     |     |
0006 |     |     |     |     |     |     |     |True  |Start |     |     |     |WF3
0007 |store |     |     |     |     |     |     |     |     |     |     |     |RAF1
0008 |     |S1   |A    |     |MDR  |     |     |     |     |     |     |     |
0009 |     |S1   |IR   |     |MAR  |Word  |     |     |     |     |     |     |
000A |     |     |     |     |     |     |     |True  |Start |WW   |MAR  |MDR  |
000B |load  |S1   |IR   |     |MAR  |Word  |     |     |     |     |     |     |
000C |     |     |     |     |     |     |     |     |     |RW   |MAR  |MDR  |
000D |     |S1   |MDR  |     |C    |     |     |     |     |     |     |     |
000E |     |     |     |     |     |     |     |True  |Start |     |     |     |WF1
000F |jmp  |     |     |     |     |     |     |     |     |     |     |     |
0010 |     |ADD  |PC   |IR   |PC   |Word  |     |True  |Start |     |     |     |
```

Sprawdzenie poprawnie wykonanej mikrooperacji polega na testowaniu pamięci po wykonaniu całego programu.
Operacje typu Write Back (WB) są realizowane w ramach *Reqs*, czyli operacja WF3 zapisze nam wartość z rejestru C do R3. To znaczy gdyby w wierszu mikrooperacji umieścić tylko taką operację, otrzymalibyśmy w wyniku przepisanie C do R3. Należy jednak zauważyć, że operacje ALU i operacje Mem wykonywane są równolegle.

Po każdej zmianie mikrooperacji należy je zachować File/Save.

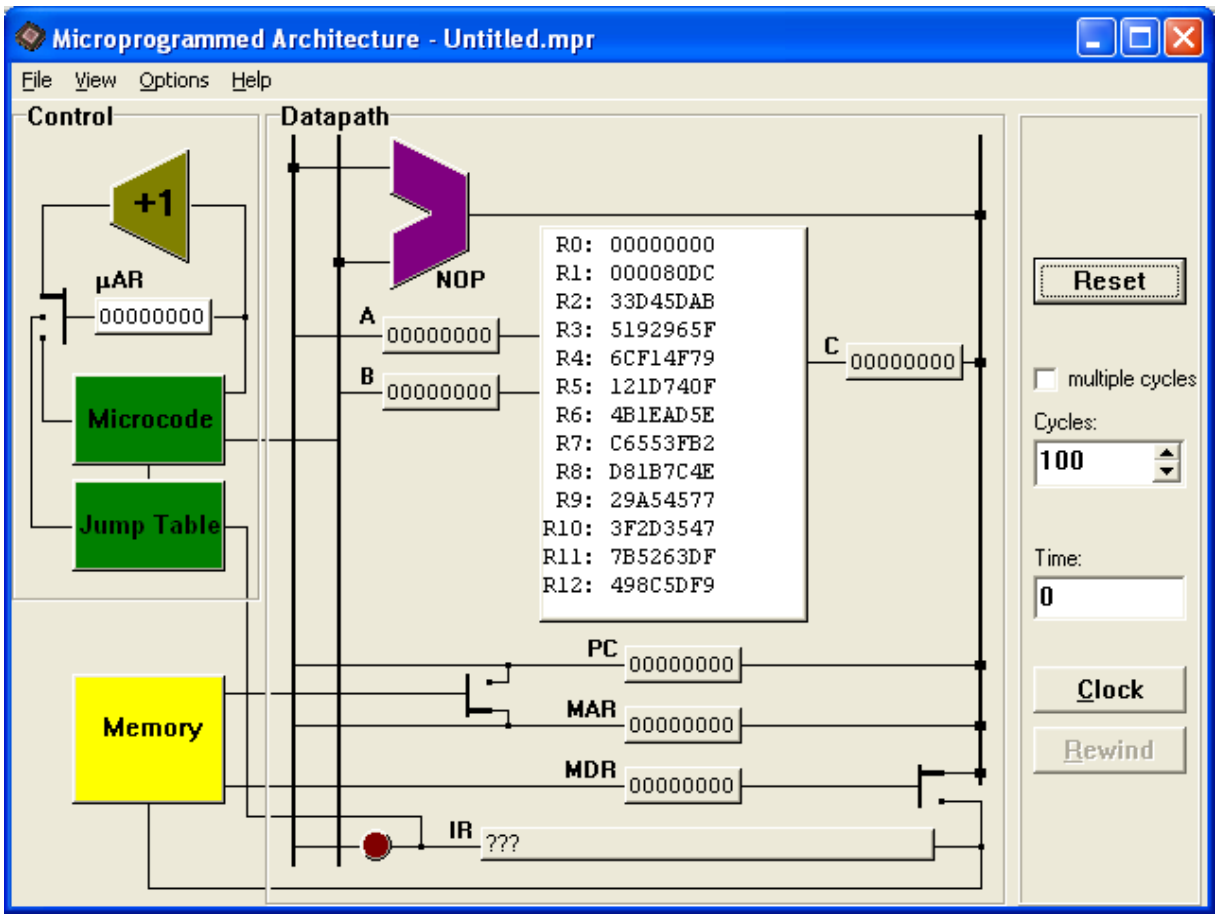


Rys. 3 Zestaw mikroinstrukcji do oprogramowania z wykorzystaniem mikrooperacji



Rys. 4 Przykładowy zestaw mikrooperacji dla danego zestawu mikroinstukcji

Pytanie 1 Co trzeba zmienić, jeśli zwiększy się czas dostępu do pamięci do kilku cykli zegarowych.



Rys. 5 Główne okno testowania mikrooperacji dla wybranej architektury RISC

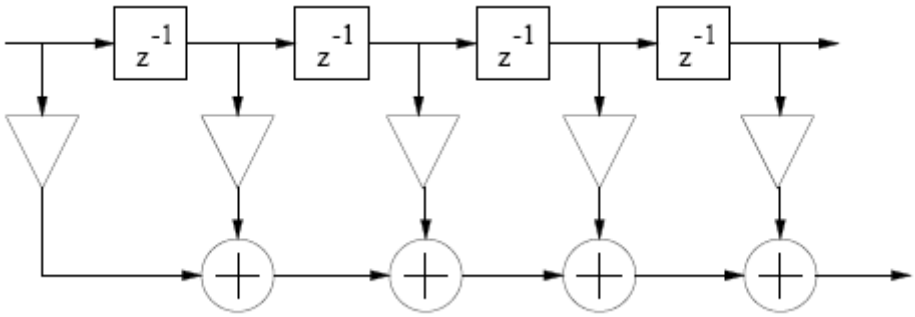
3. Omówienie programu do implementacji

3.1 Struktura filtru SOI

Dla każdej próbki wejściowej x(n) (zapisanej w pamięci), obliczany jest splot. W rezultacie otrzymujemy n próbek na wyjściu:

$$y(n) = \sum_{k=0}^{L-1} h(k) \times (n-k)$$

3.2 Implementacja



Rysunek 1: Linia opóźniająca filtru SOI

W naszym przypadku N (= 32) próbki wejściowe są przechowywane w pamięci danych (= 0x200 - 0x27F). Kopiowane są one do bufora cyklicznego reprezentującego linię opóźniającą (linia opóźniająca Rys. 1) (L=16, 0x280 - 0x2BF). Splot jest obliczany za pomocą współczynników przechowywanych w pamięci pod adresami 0x2c0 - 0x2FF. 0x2c0 - 0x2FF. Skumulowane wyniki są przechowywane w tablicy wyników (0x300 - 0x37F). Pamięć jest adresowana bajtowo a wszystkie próbki posiadają 32-bity.

Tabela 1

Rejestry	Wykorzystanie
R1	licznik próbek
R2	licznik pętli splotu
R3	akumulator
R4	adres aktualnej próbki do zapisy
R5	adres aktualnej próbki do odczytu
R6-R11	rejestry tymczasowe

Tabela 2. Symbole wykorzystane w pseudokodzie

Nazwa	wartość	Znaczenie
N	32	Liczba próbek wejściowych
L	16	Długość linii opóźniającej, rząd filtru
INPUT	0x200	Adres bazowy próbek wejściowych
DELAY	0x280	Ciąg próbek opóźnionych. Adres bazowy elementów linii opóźniającej
COEFF	0x2C0	Adres bazowy współczynników filtru

OUTPUT	0x300	Adres bazowy pamięci próbek na wyjściu

3.3 Pseudokod

Zobacz rejestry z Tab. 1 oraz zobacz tablicę symboli: Tab. 2

```
R4 = 0
        FOR R1 = 0 TO 4*(N-1) krok o 4
        R7 = MEM[INPUT + R1]
        MEM[DELAY + R4] = R7
        R3 = 0
        R5 = R4
        FOR R2 = 0 TO 4*(L-1) krok o 4
                R9 = MEM[DELAY + R5]
                R10 = MEM[COEFF + R2]
                R3 = R3 + R9*R10
                R5 = R5 + 4
                IF ( R5 > 4*(L-1) )
                THEN R5 = 0
        END FOR
        MEM[OUTPUT + R1] = R3
        R4 = R4 - 4
        IF ( R4 < 0 )
        THEN R4 = 4*(L-1)
END FOR
```

4 Przebieg ćwiczenia

4.1 Pobieranie i uruchamianie materiałów oraz symulatora

• Pobierz .zip
• Rozpakuj .zip w przygotowanym katalogu roboczym
• Uruchom program

4.2 Podstawowa konfiguracja architektury

Załaduj plik soi.ecf z formularza konfiguracji. Definiuje on dość prostą architekturę, zapoznaj się z nią. Popatrz na dostępne instrukcje w karcie kodowania rozkazów. Załaduj plik simple.mpr z okna Microarchitecure Window. (Menu File—Open).

4.3 Zadania do wykonania

Z1 Napisz program asemblerowy dla pseudokodu z sekcji 3.3
Z2 Sprawdź gdzie można poprawić kod poprzez grupowanie bloków instrukcji w większe bloki wykorzystując na przykład kombinację Multiply-Accumulate, która znajduje się w prawie każdym procesorze DSP. Inne możliwości optymalizacji można wprowadzić przy dostępie do pamięci.
Z3 Porównaj prędkość poprawionej wersji z oryginałem – wykonaj to dla kilku wartości czasu dostępu do pamięci (menu Opcje-Memory Access Time).

5 Wskazówki

- Zwróć uwagę na Jump Table
- zestaw instrukcji

- WFn zapisuje do parametru formalnego zawartość rejestru C

Instrukcja	Operacja
`NOP`	–
`ADD R1, R2, R3`	R3 := R1 + R2
`SUB R1, R2, R3`	R3 := R1 − R2
`MUL R1, R2, R3`	R3 := R1 × R2
`DIV R1, R2, R3`	R3 := R1 / R2
`AND R1, R2, R3`	R3 := R1 AND R2
`OR R1, R2, R3`	R3 := R1 OR R2
`XOR R1, R2, R3`	R3 := R1 XOR R2
`SLL R1, R2, R3`	R3 := R1 << R2
`SRL R1, R2, R3`	R3 := R1 >> R2
`SRA R1, R2, R3`	R3 := R1 $>>_a$ R2
`ADDI R1, imm, R3`	R3 := R1 + imm
`SUBI R1, imm, R3`	R3 := R1 − imm
`MULI R1, imm, R3`	R3 := R1 × imm
`DIVI R1, imm, R3`	R3 := R1 / imm
`ANDI R1, imm, R3`	R3 := R1 AND imm
`ORI R1, imm, R3`	R3 := R1 OR imm
`XORI R1, imm, R3`	R3 := R1 XOR imm
`SLLI R1, imm, R3`	R3 := R1 << imm
`SRLI R1, imm, R3`	R3 := R1 >> imm
`SRAI R1, imm, R3`	R3 := R1 $>>_a$ imm
`LDB R1, offset(R2)`	R1 := 0x0 + mem8[R2 + offset]
`LDH R1, offset(R2)`	R1 := 0x0 + mem16[R2 + offset]
`LDW R1, offset(R2)`	R1 := 0x0 + mem32[R2 + offset]
`STB R1, offset(R2)`	mem8[R2 + offset] := R1(7:0)
`STH R1, offset(R2)`	mem16[R2 + offset] := R1(15:0)
`STW R1, offset(R2)`	mem32[R2 + offset] := R1
`LIH R1, immediate`	R1 := (immediate << 16) OR (R1 AND 0XFFFF)
`BRZ R1, label`	if R1 = 0 then PC := adr(label) else PC := PC+4
`BRNZ R1, label`	if R1 <> 0 then PC := adr(label) else PC := PC+4
`BRGT R1, label`	if R1 > 0 then PC := adr(label) else PC := PC+4
`BRGE R1, label`	if R1 >= 0 then PC := adr(label) else PC := PC+4
`BRLT R1, label`	if R1 < 0 then PC := adr(label) else PC := PC+4
`BRLE R1, label`	if R1 <= 0 then PC := adr(label) else PC := PC+4

Uwaga:
• Rejestr R0 zawiera zawsze wartość zero.
• Pamięć adresowana jest bajtowo, wszystkie wartości są 32-bitowe co może spowodować wygenerowanie błędu np. przy korzystaniu z adresu 0x63.
• Przy wprowadzaniu nowych instrukcji Wykorzystaj tymczasowe wewnętrzne rejestry (TMP1, TMP2).
• Wyczyść pamięć danych po symulacji.
• Poproś prowadzącego o wskazówki.
• Wstawianie nowej linii w oknie Microcode odbywa się za pomocą klawiszy Insert(zmiana trybu na INSERT), Enter (Enter wstawia od tego momentu nową linie).
Memory Unstable – zwróć uwagę na czas dostępu do pamięci, jeśli jest większy od 1, należy ten przypadek oprogramować.

5. Usuń katalog roboczy po zakończonym laboratorium.