Wydział Informatyki AGH

Prosty system mikroprocesorowy (przygotowanie do ćwiczenia 1)

Ada Brzoza-Zajęcka, abrzoza@agh.edu.pl

Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
AGH University of Krakow

Update 20251011

Plan realizacji ćwiczenia

1) Najpierw testowo spróbujemy poznać działanie mikroprocesora w systemie SCS bazując na jego symulacji w języku Python
2) Napiszemy programy w języku maszynowym i sprawdzimy je na symulowanym CPU w Pythonie
3) Dodamy możliwość wykrywania nieobsługiwanych instrukcji używając CPU w implementowanego w Pythonie
4) Dodamy nową instrukcję do mikroprocesora, np. dodawanie z wynikiem do R1, odejmowanie, mnożenie
5) Zaimplementujemy (prawie) identyczny system mikroprocesorowy, tylko w rzeczywistym sprzęcie, w układzie FPGA, używając środowiska Vivado
6) Przetestujemy nasze programy z pkt 2 na zaimplementowanym rzeczywistym sprzęcie
7) Dodamy nową instrukcję i przetestujemy ją na sprzętowej wersji mikroprocesora - analogicznie jak w punkcie 4.

System komputerowy SCS z CPU realizowanym całkowicie programowo

- Potrzebujemy pliki źródłowe SCS-CPU scs.py i cpu.py dostępne na platformie UPEL (*bardzo* zalecane jest pobranie "świeżych" plików, a nie używanie znalezionych na dysku - dotyczy absolutnie wszystkich ćwiczeń ;))
- Umieszczamy pliki w wybranym katalogu
- uruchamiamy:
  `python3 scs.py`
- Impulsy sygnału zegarowego podajemy przy pomocy spacji
- Na ekranie obserwujemy stan wewnętrznych rejestrów i pamięci przy realizacji programu widocznego tutaj z prawej
- Kod programu (a właściwie zawartość całej pamięci) możemy zmieniać, edytując listę **`base_program`** podawaną jako argument **`mem`** do konstruktora klasy **`Cpu`**.
- Dokumentacja sprzętowej wersji mikroprocesora dla systemu SCS dostępna jest również na UPEL - implementacja w Pythonie działa bardzo podobnie i można powiedzieć, że model programowy obu wersji jest taki sam.
- Objaśnienie linii wyświetlanej na terminalu znajduje się na następnym slajdzie: →→→

Adres	Instrukcja	Kod bin.	Kod hex.
00	mov r0, #0	1000 0000	80
01	mov r1, #0	1001 0000	90
02	mov r0, #1	1000 0001	81
03	mov r1, #2	1001 0010	92
04	mov r0, r1	1010 0000	A0
05	jmp #0	0000 0000	00

```
base_program = [\
  0x80, \
  0x90, \
  0x81, \
  0x92, \
  0xA0, \
  0x00, \
  0x00, \
  0x00, \
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00]
```

Obserwacja wewnętrznego stanu procesora

- Z każdym cyklem sygnału zegarowego wyświetla się linia zawierająca informację o wewnętrznym stanie procesora SCS CPU, np.
  c=1 r0=0 r1=0 pc=0 A=0 R=80 W=00 M: 80 90 81 92 A0 00 00 00 00 00 00 00 00 00 00 00
- Objaśnienia pól:
  - c - nr stanu wewnętrznego w cyklu instrukcji (0..3)
  - r0, r1 - stany wewnętrznych rejestrów ogólnego przeznaczenia
  - pc - stan licznika programu PC
  - A - dane na magistrali adresowej pamięci operacyjnej
  - R - dane na magistrali odczytu z pamięci operacyjnej
  - W - dane na magistrali zapisu do pamięci operacyjnej
  - M - zawartość pamięci operacyjnej (program i ew. dane)

Adres	Instrukcja	Kod bin.	Kod hex.
00	mov r0, #0	1000 0000	80
01	mov r1, #0	1001 0000	90
02	mov r0, #1	1000 0001	81
03	mov r1, #2	1001 0010	92
04	mov r0, r1	1010 0000	A0
05	jmp #0	0000 0000	00

Pierwszy własny program - przygotowanie

- Za chwilę utworzymy własny program zapisany jako kod maszynowy w 16-elementowej liście, którą podamy jako argument mem konstruktora klasy **`Cpu`**: Tutaj zalecane jest "kopiuj, wklej", a następnie zmiana nazwy i edycja listy base_program. **Nazwę base_program zmieniamy na: `load_store_program`**.
- Zanim w ogóle zaczniemy kodować właściwe instrukcje, należy wpisać liczbę **`0x05`** do komórki o adresie (tu indeksie) **13**, licząc od 0. Ta liczba będzie stanowiła naszą daną wejściową:

Ustalamy taką nazwę (dla ujednolicenia rozwiązań)

Tutaj wpiszemy nasz program

A tutaj pod adresem 13 mamy przygotowaną daną wejściową 0x05

```
program load_store_program = [\
    0x??, \
    0x??, \
    0x??, \
    0x??, \
    0x??, \
    0x??, \
    0x??, \
    0x??, \
    0x00, 0x00, 0x00, 0x00, 0x00, 0x05, 0x00, 0x00]
```

Pierwszy własny program - asemblacja

- Program powinien być napisany jako ciąg liczb w systemie szesnastkowym i powinien wykonywać następujące czynności:
  1) Wyczyścić zawartość rejestrów R0 i R1 wpisując do nich wartość 0,
  2) Wyczyścić zawartość komórki pamięci o adresie 14 wpisując do niej wartość 0,
  3) Do rejestru R0 wpisać liczbę 0x6
  4) Do aktualnej wartości przechowywanej w rejestrze R0 dodać wartość przechowywaną w pamięci RAM pod adresem 13 (tak, aby w R0 znalazła się suma zawartości R0 i komórki pamięci o adresie 13)
  5) Umieścić zawartość rejestru R0 w pamięci RAM w komórce o adresie 14,
  6) Powrócić do adresu 0, aby zapętlić się.

Sprawozdanie:
a) Przedstawiamy osobie prowadzącej, co się dzieje w programie, w tym: **potrafimy odpowiedzieć na pytanie "po czym poznajemy, że program zapętlił się?"**
b) Na UPEL umieszczamy screenshot z ekranu wyświetlającego przejście przez cały program wraz z widocznym zapętleniem się.

```
program load_store_program = [\
  0x??, \
  0x??, \
  0x??, \
  0x??, \
  0x??, \
  0x??, \
  0x??, \
  0x??, \
  0x00, 0x00, 0x00, 0x00, 0x00, 0x05, 0x00, 0x00]
```

Tutaj wpiszemy nasz program

Dodanie obsługi niezdefiniowanej ("nielegalnej") instrukcji

- W tym etapie należy dopisać do programu procesora instrukcję, której dostarczony kod bazowy klasy Cpu nie rozpoznaje, np. **`0x30`**.
- Sytuacja wystąpienia nieznanej instrukcji powinna zostać wykryta w kodzie Cpu.
- Wystąpienie wyjątku powinno generować wywołanie funkcji callback, do której referencję podajemy przy wywoływaniu konstruktora klasy Cpu.

- szczegóły na kolejnym slajdzie →→→

Dodanie obsługi niezdefiniowanej ("nielegalnej") instrukcji

- Jak to zrobić:
  - W **cpu.py** należy zmodyfikować klasę Cpu w taki sposób, aby można było do jej konstruktora podać opcjonalny argument z funkcją callback (wewnętrzna funkcja może, ale nie musi być prywatna), np:

```
def __init__(self, mem, undefined_cb=None):
  self.undefined_cb = undefined_cb
```

  - W cpu.py należy dodać kod, który będzie wychwytywał wszystkie nieobsługiwane kody instrukcji
  - Jeśli nieznana instrukcja zostanie wykryta, należy wywołać funkcję przekazaną jako argument do konstruktora klasy, najlepiej, jeśli funkcja będzie informowała o adresie w pamięci, w którym znajduje się nielegalna instrukcja np.

```
self.undefined_cb(self.mem_address)
```

  - W kodzie nadrzędnym **scs.py** tworzymy funkcję callback i podajemy ją jako argument przy wywoływaniu konstruktora obiektu cpu klasy Cpu:

```
def undefined(address):
  print(f'Undefined instruction at address 0x{address:02X}')

cpu = Cpu(program_2_load_store,undefined)
```

Sprawozdanie:
a) Przedstawiamy osobie prowadzącej, jak działa program i co się dzieje w programie.
b) Na UPEL umieszczamy screenshot z ekranu wyświetlającego przejście przez cały program wraz z widocznym zapętleniem się.
c) na UPEL umieszczamy zmodyfikowane pliki scs.py, cpu.py

Drugi program - dodanie własnej instrukcji

- W tym etapie należy dopisać kod procesora, który będzie realizował jakąś instrukcję, której dostarczony kod bazowy nie realizuje. Instrukcji przypisujemy kod operacji **`0x30`** (przy prawidłowej implementacji powinien przestać pojawiać się wyjątek nielegalnej instrukcji).
- Przykłady realizowanych działań są następujące:
  - Dodawanie, jednak z wynikiem w R1
  - Mnożenie R1 * R0 z wynikiem umieszczonym w R1 (bardziej znaczące bity) i R0 (mniej znaczące bity)
  - Odejmowanie
- Następnie należy napisać program, który pozwoli na zaprezentowanie działania utworzonej instrukcji

Sprawozdanie:
a) Przedstawiamy osobie prowadzącej, jak działa program i co się dzieje w programie.
b) Na UPEL umieszczamy screenshot z ekranu wyświetlającego przejście przez cały program wraz z widocznym zapętleniem się.

Realizacja sprzętowa

- Po tym wstępie przechodzimy do implementacji sprzętowej procesora wg głównej instrukcji do części sprzętowej dostępnej na UPEL.
- Programy napisane tutaj można przepisać do uruchomienia na wersji sprzętowej.

Wydział Informatyki AGH

Prosty system mikroprocesorowy (przygotowanie do ćwiczenia 1)

Ada Brzoza-Zajęcka, abrzoza@agh.edu.pl

Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
AGH University of Krakow

Update 20251011

Plan realizacji ćwiczenia

1) Najpierw testowo spróbujemy poznać działanie mikroprocesora w systemie SCS bazując na jego symulacji w języku Python
2) Napiszemy programy w języku maszynowym i sprawdzimy je na symulowanym CPU w Pythonie
3) Dodamy możliwość wykrywania nieobsługiwanych instrukcji używając CPU w implementowanego w Pythonie
4) Dodamy nową instrukcję do mikroprocesora, np. dodawanie z wynikiem do R1, odejmowanie, mnożenie
5) Zaimplementujemy (prawie) identyczny system mikroprocesorowy, tylko w rzeczywistym sprzęcie, w układzie FPGA, używając środowiska Vivado
6) Przetestujemy nasze programy z pkt 2 na zaimplementowanym rzeczywistym sprzęcie
7) Dodamy nową instrukcję i przetestujemy ją na sprzętowej wersji mikroprocesora - analogicznie jak w punkcie 4.

System komputerowy SCS z CPU realizowanym całkowicie programowo

- Potrzebujemy pliki źródłowe SCS-CPU scs.py i cpu.py dostępne na platformie UPEL (*bardzo* zalecane jest pobranie "świeżych" plików, a nie używanie znalezionych na dysku - dotyczy absolutnie wszystkich ćwiczeń ;))
- Umieszczamy pliki w wybranym katalogu
- uruchamiamy:
  `python3 scs.py`
- Impulsy sygnału zegarowego podajemy przy pomocy spacji
- Na ekranie obserwujemy stan wewnętrznych rejestrów i pamięci przy realizacji programu widocznego tutaj z prawej
- Kod programu (a właściwie zawartość całej pamięci) możemy zmieniać, edytując listę **`base_program`** podawaną jako argument **`mem`** do konstruktora klasy **`Cpu`**.
- Dokumentacja sprzętowej wersji mikroprocesora dla systemu SCS dostępna jest również na UPEL - implementacja w Pythonie działa bardzo podobnie i można powiedzieć, że model programowy obu wersji jest taki sam.
- Objaśnienie linii wyświetlanej na terminalu znajduje się na następnym slajdzie: →→→

Adres	Instrukcja	Kod bin.	Kod hex.
00	mov r0, #0	1000 0000	80
01	mov r1, #0	1001 0000	90
02	mov r0, #1	1000 0001	81
03	mov r1, #2	1001 0010	92
04	mov r0, r1	1010 0000	A0
05	jmp #0	0000 0000	00

```
base_program = [\
  0x80, \
  0x90, \
  0x81, \
  0x92, \
  0xA0, \
  0x00, \
  0x00, \
  0x00, \
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00]
```

Obserwacja wewnętrznego stanu procesora

- Z każdym cyklem sygnału zegarowego wyświetla się linia zawierająca informację o wewnętrznym stanie procesora SCS CPU, np.
  c=1 r0=0 r1=0 pc=0 A=0 R=80 W=00 M: 80 90 81 92 A0 00 00 00 00 00 00 00 00 00 00 00
- Objaśnienia pól:
  - c - nr stanu wewnętrznego w cyklu instrukcji (0..3)
  - r0, r1 - stany wewnętrznych rejestrów ogólnego przeznaczenia
  - pc - stan licznika programu PC
  - A - dane na magistrali adresowej pamięci operacyjnej
  - R - dane na magistrali odczytu z pamięci operacyjnej
  - W - dane na magistrali zapisu do pamięci operacyjnej
  - M - zawartość pamięci operacyjnej (program i ew. dane)

Adres	Instrukcja	Kod bin.	Kod hex.
00	mov r0, #0	1000 0000	80
01	mov r1, #0	1001 0000	90
02	mov r0, #1	1000 0001	81
03	mov r1, #2	1001 0010	92
04	mov r0, r1	1010 0000	A0
05	jmp #0	0000 0000	00

Pierwszy własny program - przygotowanie

- Za chwilę utworzymy własny program zapisany jako kod maszynowy w 16-elementowej liście, którą podamy jako argument mem konstruktora klasy **`Cpu`**: Tutaj zalecane jest "kopiuj, wklej", a następnie zmiana nazwy i edycja listy base_program. **Nazwę base_program zmieniamy na: `load_store_program`**.
- Zanim w ogóle zaczniemy kodować właściwe instrukcje, należy wpisać liczbę **`0x05`** do komórki o adresie (tu indeksie) **13**, licząc od 0. Ta liczba będzie stanowiła naszą daną wejściową:

Ustalamy taką nazwę (dla ujednolicenia rozwiązań)

Tutaj wpiszemy nasz program

A tutaj pod adresem 13 mamy przygotowaną daną wejściową 0x05

```
program load_store_program = [\
    0x??, \
    0x??, \
    0x??, \
    0x??, \
    0x??, \
    0x??, \
    0x??, \
    0x??, \
    0x00, 0x00, 0x00, 0x00, 0x00, 0x05, 0x00, 0x00]
```

Pierwszy własny program - asemblacja

- Program powinien być napisany jako ciąg liczb w systemie szesnastkowym i powinien wykonywać następujące czynności:
  1) Wyczyścić zawartość rejestrów R0 i R1 wpisując do nich wartość 0,
  2) Wyczyścić zawartość komórki pamięci o adresie 14 wpisując do niej wartość 0,
  3) Do rejestru R0 wpisać liczbę 0x6
  4) Do aktualnej wartości przechowywanej w rejestrze R0 dodać wartość przechowywaną w pamięci RAM pod adresem 13 (tak, aby w R0 znalazła się suma zawartości R0 i komórki pamięci o adresie 13)
  5) Umieścić zawartość rejestru R0 w pamięci RAM w komórce o adresie 14,
  6) Powrócić do adresu 0, aby zapętlić się.

Sprawozdanie:
a) Przedstawiamy osobie prowadzącej, co się dzieje w programie, w tym: **potrafimy odpowiedzieć na pytanie "po czym poznajemy, że program zapętlił się?"**
b) Na UPEL umieszczamy screenshot z ekranu wyświetlającego przejście przez cały program wraz z widocznym zapętleniem się.

```
program load_store_program = [\
  0x??, \
  0x??, \
  0x??, \
  0x??, \
  0x??, \
  0x??, \
  0x??, \
  0x??, \
  0x00, 0x00, 0x00, 0x00, 0x00, 0x05, 0x00, 0x00]
```

Tutaj wpiszemy nasz program

Dodanie obsługi niezdefiniowanej ("nielegalnej") instrukcji

- W tym etapie należy dopisać do programu procesora instrukcję, której dostarczony kod bazowy klasy Cpu nie rozpoznaje, np. **`0x30`**.
- Sytuacja wystąpienia nieznanej instrukcji powinna zostać wykryta w kodzie Cpu.
- Wystąpienie wyjątku powinno generować wywołanie funkcji callback, do której referencję podajemy przy wywoływaniu konstruktora klasy Cpu.

- szczegóły na kolejnym slajdzie →→→

Dodanie obsługi niezdefiniowanej ("nielegalnej") instrukcji

- Jak to zrobić:
  - W **cpu.py** należy zmodyfikować klasę Cpu w taki sposób, aby można było do jej konstruktora podać opcjonalny argument z funkcją callback (wewnętrzna funkcja może, ale nie musi być prywatna), np:

```
def __init__(self, mem, undefined_cb=None):
  self.undefined_cb = undefined_cb
```

  - W cpu.py należy dodać kod, który będzie wychwytywał wszystkie nieobsługiwane kody instrukcji
  - Jeśli nieznana instrukcja zostanie wykryta, należy wywołać funkcję przekazaną jako argument do konstruktora klasy, najlepiej, jeśli funkcja będzie informowała o adresie w pamięci, w którym znajduje się nielegalna instrukcja np.

```
self.undefined_cb(self.mem_address)
```

  - W kodzie nadrzędnym **scs.py** tworzymy funkcję callback i podajemy ją jako argument przy wywoływaniu konstruktora obiektu cpu klasy Cpu:

```
def undefined(address):
  print(f'Undefined instruction at address 0x{address:02X}')

cpu = Cpu(program_2_load_store,undefined)
```

Sprawozdanie:
a) Przedstawiamy osobie prowadzącej, jak działa program i co się dzieje w programie.
b) Na UPEL umieszczamy screenshot z ekranu wyświetlającego przejście przez cały program wraz z widocznym zapętleniem się.
c) na UPEL umieszczamy zmodyfikowane pliki scs.py, cpu.py

Drugi program - dodanie własnej instrukcji

- W tym etapie należy dopisać kod procesora, który będzie realizował jakąś instrukcję, której dostarczony kod bazowy nie realizuje. Instrukcji przypisujemy kod operacji **`0x30`** (przy prawidłowej implementacji powinien przestać pojawiać się wyjątek nielegalnej instrukcji).
- Przykłady realizowanych działań są następujące:
  - Dodawanie, jednak z wynikiem w R1
  - Mnożenie R1 * R0 z wynikiem umieszczonym w R1 (bardziej znaczące bity) i R0 (mniej znaczące bity)
  - Odejmowanie
- Następnie należy napisać program, który pozwoli na zaprezentowanie działania utworzonej instrukcji

Sprawozdanie:
a) Przedstawiamy osobie prowadzącej, jak działa program i co się dzieje w programie.
b) Na UPEL umieszczamy screenshot z ekranu wyświetlającego przejście przez cały program wraz z widocznym zapętleniem się.

Realizacja sprzętowa

- Po tym wstępie przechodzimy do implementacji sprzętowej procesora wg głównej instrukcji do części sprzętowej dostępnej na UPEL.
- Programy napisane tutaj można przepisać do uruchomienia na wersji sprzętowej.