Logika cyfrowa

Wykład 12: wprowadzenie do RISC V

Marek Materzok

29 maja 2023

Maszyna uniwersalna

Obwody cyfrowe jako implementacja algorytmów

Możemy rozumieć obwody jako algorytmy:

- Układy kombinacyjne – algorytmy działające w czasie stałym
- Automaty skończone – algorytmy działające w stałej pamięci
- Automaty skończone z (potencjalnie nieograniczoną) pamięcią – dowolne algorytmy

Obwody cyfrowe jako implementacja algorytmów

Możemy rozumieć obwody jako algorytmy:

- Układy kombinacyjne – algorytmy działające w czasie stałym
- Automaty skończone – algorytmy działające w stałej pamięci
- Automaty skończone z (potencjalnie nieograniczoną) pamięcią – dowolne algorytmy

Czy istnieje obwód, który może wykonać *dowolny algorytm*?

Algorytm uniwersalny

Algorytm uniwersalny – to algorytm, który jest w stanie wykonać **dowolny inny algorytm**, opisany w wejściu dla algorytmu uniwersalnego (np. w formie *programu*).

Algorytm uniwersalny

Algorytm uniwersalny – to algorytm, który jest w stanie wykonać **dowolny inny algorytm**, opisany w wejściu dla algorytmu uniwersalnego (np. w formie *programu*).

Programiści nazywają algorytmy uniwersalne **interpreterami**!

Automat uniwersalny

Automat uniwersalny – to automat, który jest w stanie symulować **dowolny inny automat**, opisany np. w pamięci tego automatu.

Automat uniwersalny

Automat uniwersalny – to automat, który jest w stanie symulować **dowolny inny automat**, opisany np. w pamięci tego automatu.

Interpreter zaimplementowany w formie automatu!

Automat uniwersalny

Automat uniwersalny – to automat, który jest w stanie symulować **dowolny inny automat**, opisany np. w pamięci tego automatu.

Interpreter zaimplementowany w formie automatu!

W teorii automatów – np. uniwersalna maszyna Turinga (UTM)

Automat uniwersalny

Automat uniwersalny – to automat, który jest w stanie symulować **dowolny inny automat**, opisany np. w pamięci tego automatu.

Interpreter zaimplementowany w formie automatu!

W teorii automatów – np. uniwersalna maszyna Turinga (UTM)

W układach cyfrowych – *procesor* (central processing unit, CPU)

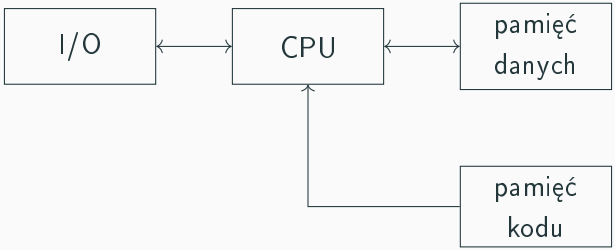
Architektura von Neumanna

Program zapisany w pamięci, razem z danymi:

Stosowana np. w dawnych procesorach 8-bitowych i 16-bitowych (np. Z80, 8086, 6809, 68000), niektórych mikrokontrolerach (np. ARM Cortex M0, M0+, M1).

Architektura harwardzka

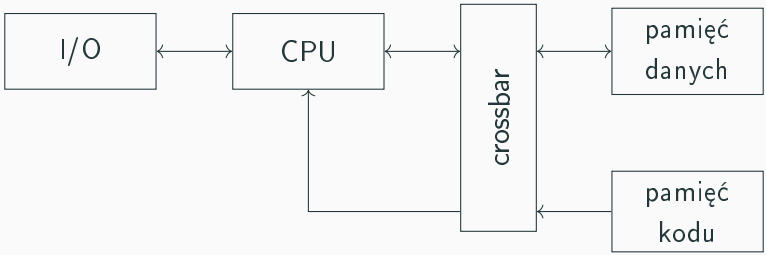
Program zapisany w pamięci rozdzielnej od pamięci danych:



Stosowana np. w niektórych mikrokontrolerach (np. AVR, PIC).

Zmodyfikowana architektura harwardzka

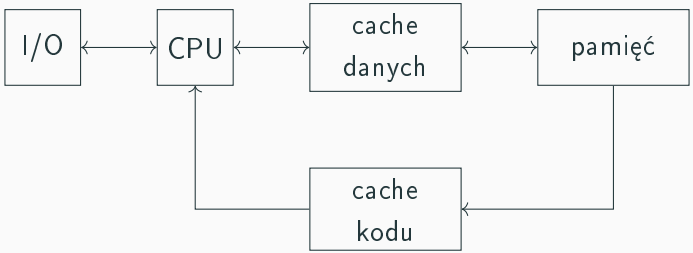
Rozdzielne pamięci programu i danych, wspólna przestrzeń adresowa:



Stosowana np. w niektórych mikrokontrolerach (np. ARM Cortex M3, M4, M7).

Zmodyfikowana architektura harwardzka

Rozdzielne pamięci podręczne, wspólna pamięć główna:



Stosowana np. we współczesnych komputerach osobistych i mobilnych.

Programy dla procesorów

Tak zwany *kod maszynowy*.

Program jest ciągiem ponumerowanych (adresowanych) *instrukcji*, zakodowanych binarnie. Instrukcje dzielą się na:

- *przetwarzania danych* – zlecają wykonanie operacji na danych (np. dodawania)
- *operacji na pamięci* – przenoszą dane pomiędzy szybką wewnętrzną pamięcią procesora (*rejestrami*) a mniej efektywną, ale obszerną pamięcią zewnętrzną (*operacyjną*)
- *przepływu sterowania* – zmieniają obecnie wykonywaną instrukcję na inną, być może w zależności od danych

Asembler

Kod maszynowy zapisany w sposób czytelny dla człowieka.

Przykład programu w pseudoasemblerze (algorytm Euklidesa):

```
1 GDY a = b SKOCZ DO 9
2 GDY a < b SKOCZ DO 5
3 a <- a - b
4 SKOCZ DO 1
5 c <- a
6 a <- b
7 b <- c
8 SKOCZ DO 1
9 ZAPISZ a DO out
```

Adresy względne

Instrukcje skoku podają adres *względem* bieżącego:

```
1 GDY a = b SKOCZ DO +8
2 GDY a < b SKOCZ DO +3
3 a <- a - b
4 SKOCZ DO -3
5 c <- a
6 a <- b
7 b <- c
8 SKOCZ DO -7
9 ZAPISZ a DO out
```

Architektura RISC V

Architektura

- To, co widzi użytkownik (programista) procesora
  - „Język programowania” interpretowany przez procesor
- Rejestry, adresowanie pamięci, instrukcje, ...
- Więcej na przedmiocie „Architektury systemów komputerowych”
- *Mikroarchitektura* – sposób zaimplementowania architektury w sprzęcie

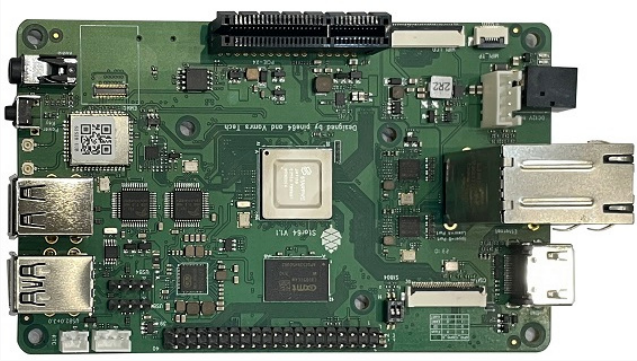
Znane architektury

- x86-64 (albo AMD64) – komputery PC
- ARM – smartfony, tablety, Raspberry Pi
- MIPS – routery, dekodery TV, PlayStation 2
- PowerPC – PlayStation 3, Xbox 360, kiedyś Apple

RISC V – wprowadzenie

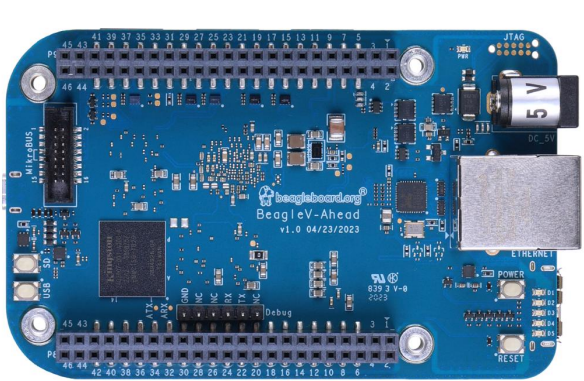
- Rozwijana od 2010 roku na Uniwersytecie Kalifornijskim w Berkeley
- Dokumentacja na licencji Creative Commons
- Inspirowana m.in. architekturą MIPS (1985)
- Odmiany 32, 64 i 128-bitowe (RV32I, RV64I, RV128I)
- Modularna, podzielona na rozszerzenia (M, A, F, D, C, N...)
- 32-bitowe instrukcje (podstawowe)
- Liczne otwarte implementacje:
  `https://github.com/riscv/riscv-cores-list`
- Wsparcie software'owe: Linux, GCC, LLVM, QEMU
- Procesory produkowane komercyjnie (np. SiFive)

Star64



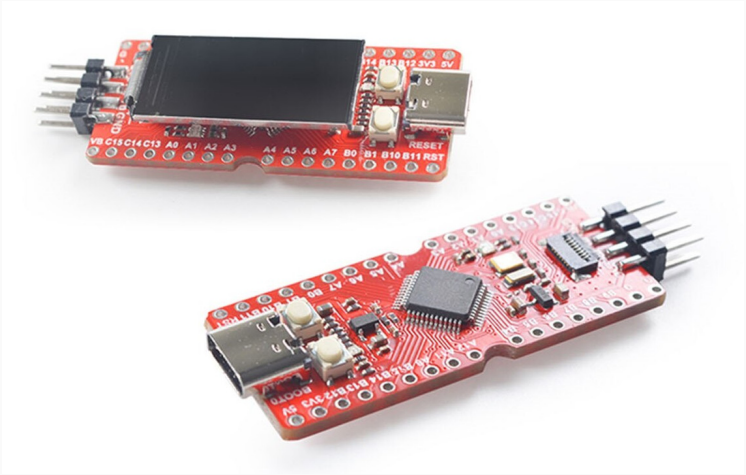
90$

BeagleV-Ahead



150$

Longan Nano – GD32VF103



6$

RISC V – otwarte implementacje

Możliwe do uruchomienia na FPGA:

- Rocket Chip (Chisel, BSD)
- BOOM: Berkeley Out-of-Order Machine (Chisel, BSD)
- VexRiscv (SpinalHDL, MIT)
- ORCA (VHDL, BSD)
- RI5CY (SystemVerilog, BSD)
- Ariane (SystemVerilog, SHL)
- PicoRV32 (Verilog, ISC)
- riscv_simple (SystemVerilog, BSD) – UWr
- Coreblocks (Amaranth, BSD) – UWr
- ...

RISC V – rozszerzenia

- M – mnożenie i dzielenie
- A – instrukcje atomowe
- F/D – instrukcje zmiennoprzecinkowe
- C – instrukcje skompresowane (16-bitowe)
- N – przerwania
- B – operacje bitowe
- V – operacje wektorowe

Wybrany wariant RISC V

Na wykładzie będziemy korzystać z podstawowego wariantu 32-bitowego RISC V, bez rozszerzeń (czyli RV32I).

W dalszej części wykładu, jeśli nie napisano inaczej, opisy dotyczą właśnie tego wariantu.

Rejestry w RISC V

- 32 ponumerowane rejestry x0...x31
  - 32-bitowe (RV32I)
  - 64-bitowe (RV64I)
  - 128-bitowe (RV128I)
- rejestr x0 specjalny – zawsze zawiera 0
- dodatkowy rejestr pc zawierający adres bieżącej instrukcji
  modyfikowany tylko przez instrukcje sterowania

Rodzaje instrukcji w RISC V

- arytmetyka między rejestrami
- arytmetyka między rejestrem i stałą
- skok warunkowy względny
- skok bezwarunkowy względny
- skok bezwarunkowy pod adres w rejestrze
- odczyt z pamięci
- zapis do pamięci
- ładowanie stałej do rejestru
- ładowanie adresu względnego do rejestru
- instrukcje systemowe

Instrukcje RISC V

Instrukcje arytmetyczno-logiczne

```
add x1, x2, x3
```

Zapisuje do `x1` sumę wartości rejestrów `x2` i `x3`.

Instrukcje arytmetyczno-logiczne

```
add x1, x2, x3
```

Zapisuje do `x1` sumę wartości rejestrów `x2` i `x3`.

Pozostałe instrukcje arytmetyczno-logiczne:

- `sub` – odejmowanie
- `and`, `or`, `xor` – operacje bitowe
- `slt`, `sltu` – porównanie ze znakiem/bez znaku (*set if less than*)
- `sll`, `srl` – przesunięcie w lewo/prawo (*shift left logical*)
- `sra` – przesunięcie arytmetyczne w prawo (*shift right arithmetic*)

Wspólny format instrukcji

Wszystkie wymienione instrukcje arytmetyczno-logiczne (`add`, `sub`, `and`, `or`, ...) mają wspólny format:

```
op x1, x2, x3
```

Wszystkie wykonują operację na wartościach dwóch rejestrów i zapisują wynik w trzecim.

Dzięki temu obwody je obsługujące współdzielą wiele części.

A co z wartością przeciwną?

- Co zrobić, jeśli chcemy zapisać do `x1` wartość przeciwną do `x2` (czyli $-$`x2`)?

A co z wartością przeciwną?

- Co zrobić, jeśli chcemy zapisać do `x1` wartość przeciwną do `x2` (czyli $-$`x2`)?

  ```
  sub x1, x0, x2
  ```

  Rejestr `x0` ma zawsze wartość 0!

A co z wartością przeciwną?

- Co zrobić, jeśli chcemy zapisać do `x1` wartość przeciwną do `x2` (czyli $-$`x2`)?

  `sub x1, x0, x2`

  Rejestr `x0` ma zawsze wartość 0!
- Można napisać w skrócie:

  `neg x1, x2`

  Taki `neg` to *pseudoinstrukcja* („lukier syntaktyczny"). Procesor widzi instrukcję `sub`.

Instrukcje arytmetyczno-logiczne ze stałą

`addi x1, x2, n`

Zapisuje do `x1` sumę wartości rejestru `x2` i stałej (*immediate*) `n`.

- Stała `n` (ze znakiem) jest częścią instrukcji, jest zakodowana na 12 bitach (z 32).

Instrukcje arytmetyczno-logiczne ze stałą

`addi x1, x2, n`

Zapisuje do `x1` sumę wartości rejestru `x2` i stałej (*immediate*) `n`.

- Stała `n` (ze znakiem) jest częścią instrukcji, jest zakodowana na 12 bitach (z 32).
- Pozostałe instrukcje arytmetyczno-logiczne ze stałą:
  - `andi`, `ori`, `xori` – operacje bitowe
  - `slti`, `sltiu` – porównanie ze znakiem/bez znaku
  - `slli`, `srli`, `srai` – przesunięcia bitowe

Instrukcje arytmetyczno-logiczne ze stałą

`addi x1, x2, n`

Zapisuje do `x1` sumę wartości rejestru `x2` i stałej (*immediate*) `n`.

- Stała `n` (ze znakiem) jest częścią instrukcji, jest zakodowana na 12 bitach (z 32).
- Pozostałe instrukcje arytmetyczno-logiczne ze stałą:
  - `andi`, `ori`, `xori` – operacje bitowe
  - `slti`, `sltiu` – porównanie ze znakiem/bez znaku
  - `slli`, `srli`, `srai` – przesunięcia bitowe
- Nie ma instrukcji `subi` (dlaczego?)

Inne pseudoinstrukcje

- Negacja logiczna

  `not x1, x2`

Inne pseudoinstrukcje

- Negacja logiczna

```
not x1, x2
--> xori x1, x2, -1
```

Inne pseudoinstrukcje

- Negacja logiczna

  ```
  not x1, x2
  --> xori x1, x2, -1
  ```

- Kopiowanie z rejestru do rejestru

  ```
  mv x1, x2
  ```

Inne pseudoinstrukcje

- Negacja logiczna

```
not x1, x2
--> xori x1, x2, -1
```

- Kopiowanie z rejestru do rejestru

```
mv x1, x2
--> addi x1, x2, 0
```

Skok bezwarunkowy względny

`j offset`

Zacznij wykonywać instrukcję przesuniętą o `offset` od bieżącej.

- *Offset* (przesunięcie) jest ze znakiem, 21-bitowy, musi być parzysty.

Skok bezwarunkowy względny

`j offset`

Zacznij wykonywać instrukcję przesuniętą o `offset` od bieżącej.

- *Offset* (przesunięcie) jest ze znakiem, 21-bitowy, musi być parzysty.
- Instrukcja `j` jest pseudoinstrukcją – rozwinięcie będzie podane w dalszej części wykładu.

Skok warunkowy względny

`beq x1, x2, offset`

Jeśli wartości rejestru `x1` i `x2` równe, zacznij wykonywać instrukcję przesuniętą o `offset` od bieżącej.

- *Offset* (przesunięcie) jest ze znakiem, zakodowany na 12 bitach, musi być parzysty.

Skok warunkowy względny

`beq x1, x2, offset`

Jeśli wartości rejestru `x1` i `x2` równe, zacznij wykonywać instrukcję przesuniętą o `offset` od bieżącej.

- *Offset* (przesunięcie) jest ze znakiem, zakodowany na 12 bitach, musi być parzysty.
- Inne instrukcje skoku warunkowego:
  - `bne` – skok gdy różne
  - `blt`, `bltu` – skok gdy mniejszy, ze znakiem/bez znaku
  - `bge`, `bgeu` – skok gdy większy lub równy, ze znakiem/bez

Etykiety

- W asemblerze RISC V można stosować *etykiety* instrukcji.
  Zamiast podawać offset liczbowo, można wpisać etykietę – offset zostanie wyliczony przy translacji do kodu binarnego.
- Przykład:

```
forever:
    addi x1, x1, 1
    j forever
```

Przykładowy program

```
strt: beq x1, x2, end  # 1 GDY a = b SKOCZ DO 9
      blt x1, x2, swap # 2 GDY a < b SKOCZ DO 5
      sub x1, x1, x2   # 3 a <- a - b
      j strt           # 4 SKOCZ DO 1
swap: mv x3, x1        # 5 c <- a
      mv x1, x2        # 6 a <- b
      mv x2, x3        # 7 b <- c
      j strt           # 8 SKOCZ DO 1
end:  ???              # 9 ZAPISZ a DO out
```

Pamięć

Pamięć jest adresowana bajtami, ale możliwy jest też odczyt lub zapis czterech bajtów (*słowa maszynowego*) lub dwóch bajtów (*połówki słowa*) jednocześnie.

adres	dane			
⋮	⋮			
0000000c	40	f3	07	88
00000008	01	ee	28	42
00000004	f2	f1	ac	07
00000000	ab	cd	ef	78
	+3	+2	+1	+0

Konwencja *little endian* – bajt najmniej znaczący w słowie ma najmniejszy adres.

Pamięć

adres	dane			
⋮	⋮			
0000000c	40	f3	07	88
00000008	01	ee	28	42
00000004	f2	f1	ac	07
00000000	ab	cd	ef	78
	+3	+2	+1	+0

- Adres 00000000: słowo abcdef78$_h$, połówka ef78$_h$, bajt 78$_h$
- Adres 00000002: słowo ac07abcd$_h$, połówka abcd$_h$, bajt cd$_h$
- Adres 00000003: słowo f1ac07ab$_h$, połówka 07ab$_h$, bajt ab$_h$

Wyrównanie dostępów do pamięci

Dostęp do pamięci jest *wyrównany*, jeśli adres jest wielokrotnością rozmiaru odczytywanej lub zapisywanej porcji danych:

- 1 dla bajtu (zawsze wyrównane),
- 2 dla połówki słowa,
- 4 dla słowa.

Nie wyrównane dostępy w RISC V są dozwolone, ale mogą być wykonywane **dużo wolniej**. Implementacje dla systemów wbudowanych mogą nie wspierać nie wyrównanych dostępów.

Odczyt z pamięci

`lw x1, offset(x2)`

Load word – ładuje słowo (4 bajty) zaczynające się od adresu `x2` + `offset` i zapisuje je w rejestrze `x1`. Offset jest 12-bitowy.

Pozostałe instrukcje odczytu:

- `lh` – odczyt połowy słowa, ze znakiem (*load half*)
- `lb` – odczyt bajtu, ze znakiem (*load byte*)
- `lhu`, `lbu` – odczyt połowy słowa/bajtu bez znaku

Odczyty z pamięci – przykłady

adres	dane			
⋮	⋮			
0000000c	40	f3	07	88
00000008	01	ee	28	42
00000004	f2	f1	ac	07
00000000	ab	cd	ef	78
	+3	+2	+1	+0

```
lw x1, 4(x0) –
```

Odczyty z pamięci – przykłady

adres	dane			
⋮	⋮			
0000000c	40	f3	07	88
00000008	01	ee	28	42
00000004	f2	f1	ac	07
00000000	ab	cd	ef	78
	+3	+2	+1	+0

`lw x1, 4(x0)` – $\texttt{f2f1ac07}_h$

`lh x1, 8(x0)` –

Odczyty z pamięci – przykłady

adres	dane			
⋮	⋮			
0000000c	40	f3	07	88
00000008	01	ee	28	42
00000004	f2	f1	ac	07
00000000	ab	cd	ef	78
	+3	+2	+1	+0

`lw x1, 4(x0)` – $\texttt{f2f1ac07}_h$

`lh x1, 8(x0)` – $\texttt{00002842}_h$

`lh x1, 6(x0)` –

Odczyty z pamięci – przykłady

adres	dane			
⋮	⋮			
0000000c	40	f3	07	88
00000008	01	ee	28	42
00000004	f2	f1	ac	07
00000000	ab	cd	ef	78
	+3	+2	+1	+0

`lw x1, 4(x0)` – $\texttt{f2f1ac07}_h$

`lh x1, 8(x0)` – $\texttt{00002842}_h$

`lh x1, 6(x0)` – $\texttt{fffff2f1}_h$

`lhu x1, 6(x0)` –

Odczyty z pamięci – przykłady

adres	dane			
⋮	⋮			
0000000c	40	f3	07	88
00000008	01	ee	28	42
00000004	f2	f1	ac	07
00000000	ab	cd	ef	78
	+3	+2	+1	+0

`lw x1, 4(x0)` – $\texttt{f2f1ac07}_h$

`lh x1, 8(x0)` – $\texttt{00002842}_h$

`lh x1, 6(x0)` – $\texttt{ffff2f1}_h$

`lhu x1, 6(x0)` – $\texttt{0000f2f1}_h$

`lb x1, 4(x0)` –

Odczyty z pamięci – przykłady

adres	dane			
⋮	⋮			
0000000c	40	f3	07	88
00000008	01	ee	28	42
00000004	f2	f1	ac	07
00000000	ab	cd	ef	78
	+3	+2	+1	+0

`lw x1, 4(x0)` – $\texttt{f2f1ac07}_h$

`lh x1, 8(x0)` – $\texttt{00002842}_h$

`lh x1, 6(x0)` – $\texttt{ffff2f1}_h$

`lhu x1, 6(x0)` – $\texttt{0000f2f1}_h$

`lb x1, 4(x0)` – $\texttt{00000007}_h$

`lb x1, 5(x0)` –

Odczyty z pamięci – przykłady

adres	dane			
⋮	⋮			
0000000c	40	f3	07	88
00000008	01	ee	28	42
00000004	f2	f1	ac	07
00000000	ab	cd	ef	78
	+3	+2	+1	+0

`lw x1, 4(x0)` – $\texttt{f2f1ac07}_h$

`lh x1, 8(x0)` – $\texttt{00002842}_h$

`lh x1, 6(x0)` – $\texttt{ffff2f1}_h$

`lhu x1, 6(x0)` – $\texttt{0000f2f1}_h$

`lb x1, 4(x0)` – $\texttt{00000007}_h$

`lb x1, 5(x0)` – $\texttt{ffffffac}_h$

`lbu x1, 5(x0)` –

Odczyty z pamięci – przykłady

adres	dane			
⋮	⋮			
0000000c	40	f3	07	88
00000008	01	ee	28	42
00000004	f2	f1	ac	07
00000000	ab	cd	ef	78
	+3	+2	+1	+0

`lw x1, 4(x0)` – $\texttt{f2f1ac07}_h$

`lh x1, 8(x0)` – $\texttt{00002842}_h$

`lh x1, 6(x0)` – $\texttt{ffff2f1}_h$

`lhu x1, 6(x0)` – $\texttt{0000f2f1}_h$

`lb x1, 4(x0)` – $\texttt{00000007}_h$

`lb x1, 5(x0)` – $\texttt{ffffffac}_h$

`lbu x1, 5(x0)` – $\texttt{000000ac}_h$

Zapis do pamięci

`sw x1, offset(x2)`

Store word – zapisuje słowo będące wartością rejestru `x1` pod adres `x2` + `offset`. Offset jest 12-bitowy.

Pozostałe instrukcje zapisu:

- `sh` – zapis połowy słowa (z młodszych dwóch bajtów rejestru) (*store half*)
- `sb` – zapis bajtu (z najmłodszego bajtu rejestru) (*store byte*)

Zapis do pamięci

`sw x1, offset(x2)`

Store word – zapisuje słowo będące wartością rejestru `x1` pod adres `x2` + `offset`. Offset jest 12-bitowy.

Pozostałe instrukcje zapisu:

- `sh` – zapis połowy słowa (z młodszych dwóch bajtów rejestru) (*store half*)
- `sb` – zapis bajtu (z najmłodszego bajtu rejestru) (*store byte*)

Dlaczego nie ma `shu`/`sbu`?

Ładowanie stałej

`li x1, n`

Load immediate – ładuje wartość `n` do rejestru `x1`.

- Pseudoinstrukcja z dużą liczbą rozwinięć, dla wartości 12-bitowych ze znakiem rozwija się do:

  `addi x1, x0, n`

Ładowanie stałej

`li x1, n`

Load immediate – ładuje wartość `n` do rejestru `x1`.

- Pseudoinstrukcja z dużą liczbą rozwinięć, dla wartości 12-bitowych ze znakiem rozwija się do:

  `addi x1, x0, n`
- A co, jeśli chcemy załadować stałą większą niż 12-bitową?

Ładowanie dużej stałej

`lui x1, n`

Load upper immediate – ładuje (20-bitową) wartość `n` do górnych 20 bitów rejestru `x1`, pozostałe 12 wypełniając zerami.

- Innymi słowy, ładuje $\texttt{n} \ll 12$.

Ładowanie dużej stałej

`lui x1, n`

Load upper immediate – ładuje (20-bitową) wartość `n` do górnych 20 bitów rejestru `x1`, pozostałe 12 wypełniając zerami.

- Innymi słowy, ładuje $\texttt{n} \ll 12$.
- `lui x1, 0xcafed` ładuje do `x1` wartość $\texttt{cafed}000_h$.

Ładowanie dużej stałej

`lui x1, n`

Load upper immediate – ładuje (20-bitową) wartość `n` do górnych 20 bitów rejestru `x1`, pozostałe 12 wypełniając zerami.

- Innymi słowy, ładuje $\texttt{n} \ll 12$.
- `lui x1, 0xcafed` ładuje do `x1` wartość cafed000_h.
- Jak załadować wartość 32-bitową (np. cafed00d_h)?

Ładowanie dużej stałej

`lui x1, n`

Load upper immediate – ładuje (20-bitową) wartość `n` do górnych 20 bitów rejestru `x1`, pozostałe 12 wypełniając zerami.

- Innymi słowy, ładuje $\texttt{n} \ll 12$.
- `lui x1, 0xcafed` ładuje do `x1` wartość $\texttt{cafed000}_h$.
- Jak załadować wartość 32-bitową (np. $\texttt{cafed00d}_h$)?

```
lui x1, 0xcafed    # w skrócie:
ori x1, x1, 0xd    # li x1, 0xcafed00d
```

Ładowanie adresu względnego

- Adresy w RISC V są liczone względem bieżąco wykonywanej instrukcji.
- Motywacja – kod niezależny od bezwzględnego adresu, w którym jest załadowany (*position independent code*, PIC).

Ładowanie adresu względnego

```
la x1, symbol
```

Load address, ładuje adres symbolu `symbol` do rejestru `x1`.

Ładowanie adresu względnego

`la x1, symbol`

Load address, ładuje adres symbolu `symbol` do rejestru `x1`.

- Pseudoinstrukcja, podobnie jak `li` dla 32-bitowych stałych rozwija się do dwóch instrukcji, ładujących osobno górne 20 bitów i pozostałe 12:

  ```
  auipc x1, symbol[31:12]
  addi x1, x1, symbol[11:0]
  ```

Ładowanie górnych 20 bitów adresu

`auipc x1, offset`

Add upper immediate to program counter, ładuje do rejestru `x1` adres bieżąco wykonywanej instrukcji, przesunięty o `offset` $\ll 12$.

- Na przykład, instrukcja pod adresem 12345678_h:
  `auipc x1, 0x87654`
  załaduje do rejestru `x1` adres 99999678_h.

Ładowanie górnych 20 bitów adresu

`auipc x1, offset`

Add upper immediate to program counter, ładuje do rejestru `x1` adres bieżąco wykonywanej instrukcji, przesunięty o `offset` $\ll 12$.

- Na przykład, instrukcja pod adresem 12345678_h:
  `auipc x1, 0x87654`
  załaduje do rejestru `x1` adres 99999678_h.
- Jak załadować 99999999_h?

Ładowanie górnych 20 bitów adresu

`auipc x1, offset`

Add upper immediate to program counter, ładuje do rejestru `x1` adres bieżąco wykonywanej instrukcji, przesunięty o `offset` $\ll 12$.

- Na przykład, instrukcja pod adresem 12345678_h:

  `auipc x1, 0x87654`

  załaduje do rejestru `x1` adres 99999678_h.
- Jak załadować 99999999_h?

  `auipc x1, 0x87654`
  `addi x1, x1, 0x321`

Dostęp do pamięci pod adresem względnym

W poniższym kodzie instrukcja `auipc` jest pod adresem 12345678_h, chcemy odczytać bajt pod adresem 99999999_h do rejestru `x1`.

```
auipc x2, 0x87654
addi x2, x2, 0x321
lb x1, 0(x2)
```

Trzy instrukcje, jak krócej?

Dostęp do pamięci pod adresem względnym

W poniższym kodzie instrukcja `auipc` jest pod adresem 12345678_h, chcemy odczytać bajt pod adresem 99999999_h do rejestru `x1`.

```
auipc x2, 0x87654
addi x2, x2, 0x321
lb x1, 0(x2)
```

Trzy instrukcje, jak krócej?

```
auipc x2, 0x87654
lb x1, 0x321(x2)
```

Dostęp do pamięci pod adresem względnym

`lw x1, symbol, x2`

Pseudoinstrukcja, ładuje do rejestru `x1` słowo pod adresem symbolu `symbol`, używając `x2` do obliczenia adresu.

- Rozwija się do:

  ```
  auipc x2, symbol[31:12]
  lw x1, symbol[11:0](x2)
  ```

- Podobne pseudoinstrukcje są dla `lh`, `lb`, `sw`, `sh`, `sb`.

Przykładowy program – dostęp do pamięci

```
strt: beq x1, x2, end  # 1 GDY a = b SKOCZ DO 9
      blt x1, x2, swap # 2 GDY a < b SKOCZ DO 5
      sub x1, x1, x2   # 3 a <- a - b
      j strt           # 4 SKOCZ DO 1
swap: mv x3, x1        # 5 c <- a
      mv x1, x2        # 6 a <- b
      mv x2, x3        # 7 b <- c
      j strt           # 8 SKOCZ DO 1
end:  sw x1, out, x3   # 9 ZAPISZ a DO out
```

Procedury

Fragmenty kodu, które są używane w wielu różnych miejscach przez inny kod.

- Zamiast być kopiowana w każde miejsce użycia, procedura jest zapisana w jednym miejscu w pamięci, gdzie jest *wywoływana*.
- Po zakończeniu wykonania, procedura *powraca* do kodu, który go wywołał.

ABI

Interfejs binarny aplikacji (ang. *Application Binary Interface*) opisuje m.in. *konwencję wywołań* – sposób wykorzystywania rejestrów i pamięci (np. *stosu*) przy wywoływaniu procedur.

- Gdzie mają być zapisane *argumenty*
- Gdzie *zwracany* jest wynik
- Gdzie zapisywany jest *adres powrotu*
- Które rejestry zapisuje na stos kod wywołujący, a które kod wywoływany
- Który rejestr zawiera adres szczytu stosu (*stack pointer*)

Więcej o ABI na przedmiocie ASK.

Konwencje użycia rejestrów w RISC V

rejestr	nazwa ABI	opis	zapisuje
x0	zero	Zawsze zero	—
x1	ra	Adres powrotu	wołający
x2	sp	Adres szczytu stosu	wołany
x3	gp	Adres danych globalnych	—
x4	tp	Adres danych wątku (TLS)	—
x5-7	t0-2	Tymczasowe	wołający
x8-9	s0-1	Zapisywane	wołany
x10-11	a0-1	Argumenty/wyniki procedur	wołający
x12-17	a2-7	Argumenty	wołający
x18-27	s2-11	Zapisywane	wołany
x28-31	t3-6	Tymczasowe	wołający

Wywołanie procedury

```
jal x1, offset
```

Jump and link, skacze `offset` od bieżącej instrukcji i zapisuje adres następnej instrukcji w `x1`.

- Offset ze znakiem, 21-bitowy, musi być parzysty.
- Pseudoinstrukcja `j offset` rozwija się do:

  ```
  jal x0, offset
  ```
- Pseudoinstrukcja `jal offset` rozwija się do:

  ```
  jal ra, offset    # ra = x1
  ```

Wywołanie procedury z adresem w rejestrze

`jalr x1, x2, offset`

Jump and link register, skacze pod adres zapisany w rejestrze `x2`, przesunięty o `offset`, i zapisuje adres następnej instrukcji w `x1`.

- Offset ze znakiem, 12-bitowy.
- Pseudoinstrukcja `jr x1` (*jump register*) rozwija się do:

  `jalr x0, x1, 0`
- Pseudoinstrukcja `jalr x2` rozwija sie do:

  ```
  jalr ra, x2, 0    # ra = x1
  ```

Powrót z procedury

`ret`

Return, pseudoinstrukcja, skacze pod adres zapisany w rejestrze `ra` (czyli `x1`).

Rozwija się do:

```
jalr x0, ra, 0
```

Przykładowy program – w formie procedury

```
gcd:  beq a0, a1, end  # 1 GDY a = b SKOCZ DO 9
      blt a0, a1, swap # 2 GDY a < b SKOCZ DO 5
      sub a0, a0, a1   # 3 a <- a - b
      j gcd            # 4 SKOCZ DO 1
swap: mv t0, a0        # 5 c <- a
      mv a0, a1        # 6 a <- b
      mv a1, t0        # 7 b <- c
      j gcd            # 8 SKOCZ DO 1
end:  ret              # 9 ZWRÓĆ a
```

Podsumowanie instrukcji

rodzaj	instrukcje	pseudoinstr.
OP	`add`, `sub`, `and`, ...	`neg`
OP-IMM	`addi`, `andi`, ...	`mv`, `not`
BRANCH	`beq`, `bne`, `blt`, ...	`bgt`, `ble`, ...
JAL	`jal`	`j`
JALR	`jalr`	`jr`, `ret`
LOAD	`lw`, `lh`, `lb`, ...	+ symbol
STORE	`sw`, `sh`, `sb`	+ symbol
LUI	`lui`	`li`
AUIPC	`auipc`	`la`

Kodowanie instrukcji

Kodowanie instrukcji w RISC V

- Wszystkie podstawowe instrukcje kodowane w 32 bitach (zarówno w RV32I, RV64I i RV128I)
  - Skrócone 16-bitowe instrukcje w rozszerzeniu C
  - W przyszłości: rozszerzenia z długimi instrukcjami (długości podzielne przez 16 bit, nawet do 192)
- Kodowanie dobrane w taki sposób, żeby maksymalnie uprościć *dekodowanie instrukcji*
  - Niewielka liczba *formatów instrukcji*
  - Stałe pozycje bitowe pól pomiędzy formatami

Format instrukcji 32-bitowych

	xxxxxxxxxxxxxxxx		xxxxxxxxxxbbb11	
nr bitu	31	16	15	0
adres		a+2		a

- Najmłodsze dwa bity równe 1
- Kolejne trzy bity bbb różne od 111
- Najmłodsze siedem bitów tworzy *kod operacji* (*opkod*)

Podstawowe formaty instrukcji

rodzaj	r. źródł.	r.wynik.	b. stałej	format
OP	2	1	0	
OP-IMM	1	1	12	
BRANCH	2	0	12	
JAL	0	1	20	
JALR	1	1	12	
LOAD	1	1	12	
STORE	2	0	12	
LUI	0	1	20	
AUIPC	0	1	20	

Podstawowe formaty instrukcji

rodzaj	r. źródł.	r.wynik.	b. stałej	format
OP	2	1	0	R
OP-IMM	1	1	12	
BRANCH	2	0	12	
JAL	0	1	20	
JALR	1	1	12	
LOAD	1	1	12	
STORE	2	0	12	
LUI	0	1	20	
AUIPC	0	1	20	

Podstawowe formaty instrukcji

rodzaj	r. źródł.	r.wynik.	b. stałej	format
OP	2	1	0	R
OP-IMM	1	1	12	I
BRANCH	2	0	12	
JAL	0	1	20	
JALR	1	1	12	I
LOAD	1	1	12	I
STORE	2	0	12	
LUI	0	1	20	
AUIPC	0	1	20	

Podstawowe formaty instrukcji

rodzaj	r. źródł.	r.wynik.	b. stałej	format
OP	2	1	0	R
OP-IMM	1	1	12	I
BRANCH	2	0	12	S (B)
JAL	0	1	20	
JALR	1	1	12	I
LOAD	1	1	12	I
STORE	2	0	12	S
LUI	0	1	20	
AUIPC	0	1	20	

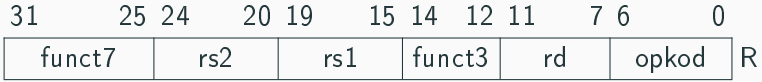
Podstawowe formaty instrukcji

rodzaj	r. źródł.	r.wynik.	b. stałej	format
OP	2	1	0	R
OP-IMM	1	1	12	I
BRANCH	2	0	12	S (B)
JAL	0	1	20	U (J)
JALR	1	1	12	I
LOAD	1	1	12	I
STORE	2	0	12	S
LUI	0	1	20	U
AUIPC	0	1	20	U

Opkody podstawowych instrukcji

rodzaj	opkod	format
OP	0110011	R
OP-IMM	0010011	I
BRANCH	1100011	S (B)
JAL	1101111	U (J)
JALR	1100111	I
LOAD	0000011	I
STORE	0100011	S
LUI	0110111	U
AUIPC	0010111	U

Format R (instrukcje OP)



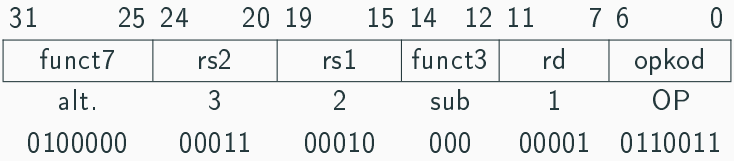
- Na przykład `add rd, rs1, rs2`
- `rs` – *register source*
- `rd` – *register destination*
- `funct3` i `funct7` kodują operację

Kody operacji arytmetyczno-logicznych

instrukcja	funct3	funct7
`add`	000	0000000
`sub`	000	0100000
`sll`	001	0000000
`slt`	010	0000000
`sltu`	011	0000000
`xor`	100	0000000
`srl`	101	0000000
`sra`	101	0100000
`or`	110	0000000
`and`	111	0000000

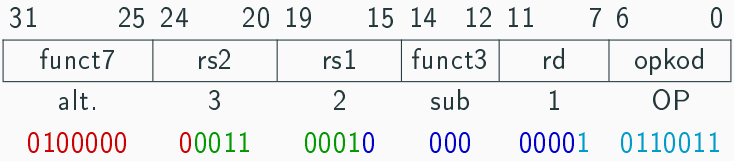
Przykładowe kodowanie instrukcji

```
sub x1, x2, x3
```



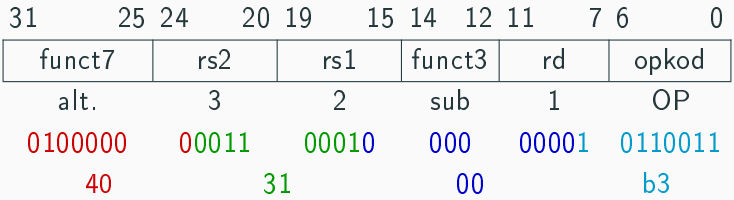
Przykładowe kodowanie instrukcji

```
sub x1, x2, x3
```



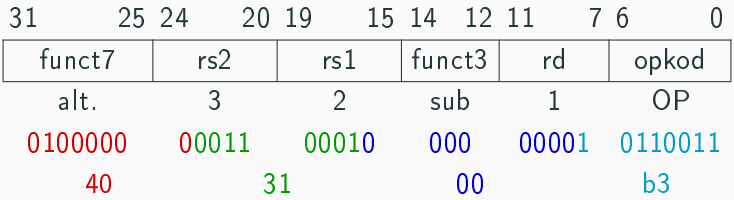
Przykładowe kodowanie instrukcji

```
sub x1, x2, x3
```



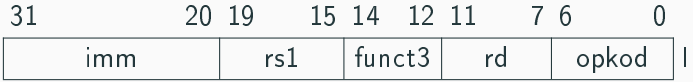
Przykładowe kodowanie instrukcji

```
sub x1, x2, x3
```



Kod instrukcji: 40c400b3

Format I (instrukcje OP-IMM, JALR, LOAD)



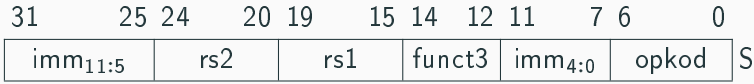
- Na przykład `addi rd, rs1, 42`, lub `lw rd, 42(rs1)`
- `imm` – wartość 12-bitowej stałej lub offsetu
- `funct3` koduje:
  - operację (OP-IMM)
  - rodzaj dostępu do pamięci (LOAD)
  - nic, zawsze 0 (JALR)

Rodzaje dostępów do pamięci

instrukcja	funct3
`lb`	000
`lh`	001
`lw`	010
`lbu`	100
`lhu`	101

- Dwa najmłodsze bity – $\log_2$ rozmiaru
- Najstarszy bit – ze znakiem (0) lub bez znaku (1)

Format S (instrukcje BRANCH i STORE)



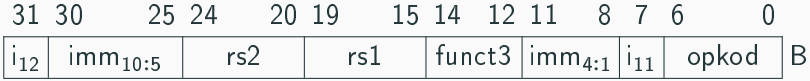
- Na przykład beq `rs1, rs2, imm`, lub `sw rs2, imm(rs1)`
- `imm` – wartość 12-bitowej stałej lub offsetu, rozbita na młodsze 5 i starsze 7 bitów
- `funct3` koduje:
  - rodzaj porównania (BRANCH)
  - rodzaj dostępu do pamięci (STORE)

Rodzaje porównań przy skoku warunkowym

instrukcja	funct3
`beq`	000
`bne`	001
`blt`	100
`bge`	101
`bltu`	110
`bgeu`	111

- Najstarszy bit – równość (0) lub nierówność (1)
- Środkowy bit – ze znakiem (0) lub bez znaku (1)
- Najmłodszy bit – negacja wyniku

Format B (odmiana S dla BRANCH)

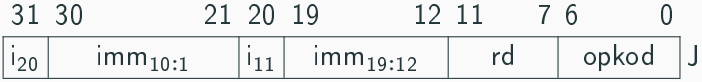


- Zmienione kodowanie stałej
- Zamiast 12-bitowej ($imm_{11:0}$) 13-bitowa ($imm_{12:1}0$)
- Podział wykonany aby:
  - pierwszy bit (zawsze 0) nie był kodowany,
  - zmaksymalizować część wspólną z S,
  - bit znaku był najstarszym bitem instrukcji.

Format U (instrukcje JAL, LUI, AUIPC)

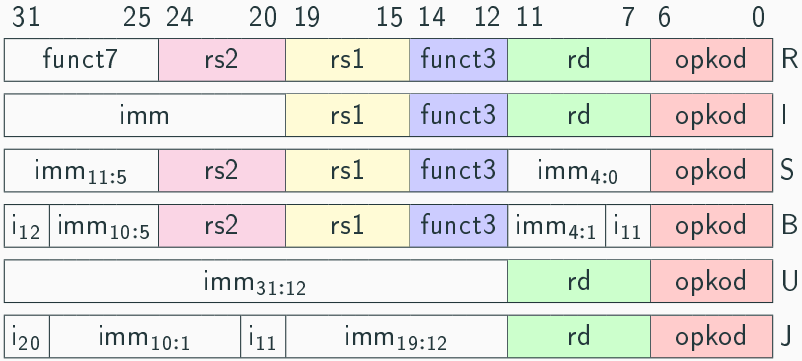
- Na przykład `jal rd, imm`, lub `lui rd, imm`
- `imm` koduje 20-bitową stałą lub offset
  - LUI i AUIPC – starsze 20 bitów
  - JAL – bity [20:1] (odmiana J)

Format J (odmiana U dla JAL)

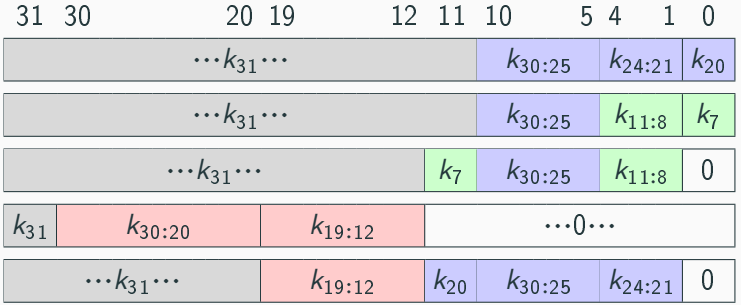


- Zmienione kodowanie stałej
- Zamiast 20-bitowej ($imm_{31:12}$) 21-bitowa ($imm_{20:1}0$)
- Podział wykonany aby:
  - pierwszy bit (zawsze 0) nie był kodowany,
  - zmaksymalizować część wspólną z I oraz U,
  - bit znaku był najstarszym bitem instrukcji.

Kodowanie instrukcji – podsumowanie



Kodowanie stałych i offsetów – podsumowanie



Oznaczenie k_i oraz $k_{j:i}$ oznaczają odpowiednie bity kodu instrukcji.

Przykładowy program – zakodowane instrukcje

```
gcd:  beq a0, a1, end  #  0: 02b50063
      blt a0, a1, swap #  4: 00b54663
      sub a0, a0, a1   #  8: 40b50533
      j gcd            #  c: ff5ff06f
swap: mv t0, a0        # 10: 00050293
      mv a0, a1        # 14: 00058513
      mv a1, t0        # 18: 00028593
      j gcd            # 1c: fe5ff06f
end:  ret              # 20: 00008067
```