POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

Kierunek: Informatyka (INF)
Specjalność: Inżynieria Systemów Informatycznych (INS)

ARCHITEKTURA KOMPUTERÓW
PROJEKT

Projekt procesora RISC (Verilog)

AUTORZY:
Piotr Szlenk 118330
Piotr Tyburski 118398

GRUPA:
CZW/N/11

PROWADZĄCY PRACĘ:
mgr inż. Ireneusz Tarnowski
Instytut Cybernetyki Technicznej

OCENA PRACY:

WROCŁAW, 27 stycznia 2004

Spis treści

Wstęp . . . 3

1. Filozofia RISC . . . 4

2. Architektura procesora Jane . . . 5

2.1. Blok rejestrów . . . 5
2.2. Jednostka wykonawcza . . . 6
2.3. Jednostka kontrolna . . . 6

3. Organizacja pamięci . . . 7

4. Przerwania i komunikacja z urządzeniami wejścia/wyjścia . . . 8

4.1. Przerwania . . . 8
4.2. Komunikacja z urządeniami I/O . . . 8

5. Instrukcje procesora Jane . . . 9

5.1. Instrukcje arytmetyczne . . . 9
5.2. Instrukcje logiczne . . . 10
5.3. Instrukcje skoków . . . 11
5.4. Instrukcje pozostałe . . . 12

6. Jane assembler . . . 14

Podsumowanie . . . 17

Dodatek A - Schematy . . . 18

Bibliografia . . . 21

Wstęp

Jako zadanie postawiliśmy sobie przygotowanie projektu procesora o architekturze RISC. Aby móc wywiązać się z tego zadania, musieliśmy dokładnie przemyśleć wszystkie logiczne cechy projektowanej przez nas jednostki, w szczególności skupiając się na liście rozkazów. Lista ta musi być możliwie najprostsza, zachowując jednak spójność i funkcjonalność z punktu widzenia programisty. Na rozwiązanie postawionego problemu składają się :

— ogólne założenia dotyczące architektury sprzętu (szerokość magistrali, rozmiar jednostki danych, etc.),
— dokładna specyfikacja architektury listy rozkazów,
— omówienie asemblera,
— symulacja procesora w języku Verilog,
— przykładowe programy napisane w asemblerze i wykazujące poprawność symulacji (czyli zgodność wyników otrzymanych z oczekiwanymi).

Ponieważ do zadania podeszliśmy z dużym zaangażowniem emocjonalnym (jako przedstawiciele nowopowstałej kultury zwanej *techie*) projektowany przez nas układ otrzymał imię, za pomocą którego będziemy się już teraz do niego odwoływać. Imię to brzmi *Jane*.

1. Filozofia RISC

Przez wiele początkowych lat projektowania mikroprocesorów wyraźnie kształtowała się tendencja do tworzenia coraz bardziej rozbudowanych list rozkazów oraz wyrafinowanych trybów adresowania. Celem tych wszystkich działań było ułatwienie programistom implementacji złożonych algorytmów. Miało to jednak niezwykle kosztowne konsekwencje. Wydłużały się słowa rozkazowe, komplikowały się jednostki sterujące, pojawiały się złożone rozkazy wymagające wielu taktów zegara aby otrzymać wynik. Te ostatnie były szczególnie niebezpieczne w rękach niewprawnych programistów, którzy z chęcią, lecz bez rozwagi korzystali z ułatwień projektantów. Budowali aplikacje, których wydajność pozostawiała wiele do życzenia i narażała użytkowników na ogromną frustrację. Architektura wspomnianych procesorów nosi miano architektury CISC(*ang. Complex Instruction Set Computer*).

Analiza statystyczna wielu tzw. typowych programów wykazała, że złożone rozkazy jak i wyszukane tryby adresowania wykorzystywane są tak naprawdę bardzo rzadko. Większość kodu to proste instrukcje arytmetyczne i logiczne. Wnioski z tej analizy były oczywiste, należało obrać nowy kierunek rozwoju architektury listy rozkazów (ISA, *ang. Instruction Set Architecture*) tym razem opierając się na `prostocie`, a może raczej `rozsądku`. W ten właśnie sposób narodził się RISC (*ang. Rational Instruction Set Computer*).

Najważniejsze cechy klasycznej architektury RISC :
— proste rozkazy i tryby adresowania, które pozwoliły na lepsze ujednolicenie struktury słowa rozkazowego oraz skróciły czas jego dekodowania i wykonania,
— rejestrowa architektura jednostki arytmetyczno-logicznej,
— duży plik rejestrowy,
— czteroetapowe przetwarzanie rozkazu : pobranie z pamięci, dekodowanie, wykonanie, zapisanie wyniku (*ang. Fetch, Decode, Execute, Write*)

Jane jest przedstawicielką klasycznej architektury RISC, więc posiada wszystkie w/w cechy. W kolejnych rozdziałach przyjrzymy się jej znacznie dokładniej.

2. Architektura procesora Jane

Jane jest 32 bitowym procesorem, na który składa się :
— 60 rejestrów ogólnego użytku w dwóch bankach po 30, oraz 3 rejestry specjalne,
— 32-bitowa jednostka arytmetyczno-logiczna,
— jednostka kontrolna.
Przestrzeń adresowa Jane jest 20-bitowa, co przy adresowaniu 32-bitowych słów pozwala na wykorzystanie łącznie $4 * 2^{20}$ bajtów. Część tej pamięci jest zarezerwowana do specjalnych celów i nie może być dowolnie wykorzystywana przez programistę.

2.1. Blok rejestrów

Rejestry ogólnego przeznaczenia ponumerowane są od R0 do R29, tak samo w banku 0 jak i 1. Numer aktywnego banku jest ustalany w rejestrze flag omówionym niżej. Każdy z tych rejestrów ma rozmiar 32 bitów.

IP (*instruction pointer*) to rejestr zawierający adres następnego rozkazu. Przy sekwencyjnym wykonaniu instrukcji po każdej z nich jest on inkrementowany o 1.

SP (*stack pointer*) to wzkaźnik stosu, wskazuje on pierwsze wolne miejsce na stosie.

Rejestr flag jest 32 bitowym rejestrem przechowującym 1-bitowe informacje. Ma on następującą strukturę:

ZF	—	CF	IA	IE	RB	—	...	—	IN	IN	IN

ZF (*zero flag*) – flaga ustawiana ilekroć wynikiem ostatniej operacji logicznej lub arytmetycznej było zero

CF (*carriage flag*) – flaga przeniesienia ustawiana kiedy w wyniku operacji arytmetycnej wystapilo przeniesienie na/z nieistniejącej po najstarszej pozycji

IA (*interrupt acknowledge*) – flaga nadejścia przerwania ustawiana w momencie zgłoszenia przez zewnętrzny sterownik wystąpienia przerwania

IE (*interrupt enable*) – flaga zezwalająca na obsługę przerwań, kiedy nie jest ustawiona procesor, będzie ignorował zgłoszenia kontrolera przerwań

RB (*register bank*) – numer aktywnego banku rejestrów

Następnie występuje 23 nieokreślonych bitów, które mogą być wykorzystane dowolnie przez programistę

Ostatnie 3 bity rejestru flag przeznaczone są do określenia numeru przerwania zgłoszonego przez zewnętrzny sterownik przerwań i są przez niego ustawiane. Więcej na ten temat w rozdziale dotyczącym przerwań procesora Jane.

2.2. Jednostka wykonawcza

Na jednostkę wykonawczą składa się 32-bitowa jednostka arytmetyczno-logiczna. Potrafi ona wykonać osiem różnych operacji arytmetycznych oraz logicznych, których zarówno operandy jak i wynik są 4-bajtowymi słowami.
Działania arytmetyczne wykonywane są tylko na liczbach całkowitych i są to: dodawanie, odejmowanie, mnożenie, dzielenie. W zakresie działań logicznych ALU *Jane*(*ang. arythemtical-logical unit*) wykonuje sumę, mnożenie, sumę modulo 2 oraz przesunięcie logiczne w dowolną stronę. Wynik działania przekazywany jest do odpowiedniego rejestru zdefiniowanego w rozkazie i jednocześnie modyfikowane są odpowiednie flagi w rejestrze FR.

2.3. Jednostka kontrolna

Jednostka kontrolna koordynuje i steruje wszystkimi czterema etapami przetwarzania rozkazu. Jego elementy składowe to główny kontroler oraz dekoder połączony z rejestrem instrukcji IR (*ang. Instruction Register*).

Główny kontroler. Ten układ jest odpowiedzialny za wysyłanie odpowiednich sygnałów sterujących przepływem danych oraz przetwarzaniem, oraz za dekodowanie instrukcji.
W fazie pobrania rozkazu wystawia on na magistrali adresowej wartość PC po czym wysyła do pamięci RAM sygnał odczytu. Jedno słowo zostaje pobrane i zapisane do rejestru IR, a licznik rozkazów jest inkrementowany.
W fazie dekodowania wysterowane są sygnały pobrania odpowiednich argumentów rozkazu z banku rejestrów. W szczególnym przypadku argument jest pobierany z zewnętrznej pamięci (instrukcja załadowania danych do rejestru). Jeśli do przetworzenia danego rozkazu jest potrzebna jednostka arytmetyczno-logiczna dostaje ona w tym momencie odpowiedni sygnał i wykonuje określoną operację na argumentach (lub tylko jednym z nich).
W fazie wykonania rozkazu aktywna jest jedynia ALU, natomiast w fazie następnej, czyli zapis wyniku, jednostka kontrolna inicjuje zapis wyniku do odpowienij komórki bloku rejestrów.

3. Organizacja pamięci

Jak już wcześniej zostało zaznaczone, przestrzeń adresowa Jane jest 20-bitowa. Pojedynczą komórką jest 32-bitowe słowo, co oznacza, że maksymalna adresowalna pamięć to $4 * 2^{20}$ bajtów. Procesor współpracuje z dowolną pamięcią RAM, której czas dostępu pozwala na odczyt/zapis w czasie jednego cyklu.
W pamięci tej przechowywane są kod wykonywalny oraz dane. Jane ma więc klasyczną architekturę von Neumanna bez podziału na pamięć instrukcji i pamięć danych.

Mapa tej pamięci prezentuje się następująco:

Tabela 1. Mapa pamięci

Adresy	Zawartość
`0x00000h - 0x00007h`	Obszar zarezerwowany na obsługę przerwań. Ponieważ dla każdego jest jedynie 32 bity, praktycznie znajdują się tam instrukcje skoków do własciwych procedur obsługi.
`0x00008h - 0x0000Fh`	Adresy zarezerwowane do komunikacji z urządzeniami wejścia/wyjścia.
`0x00010h`	Pierwsza instrukcja wykonywana po starcie procesora.
`0x00011h - 0xFFFFFh`	Pamięć dowolnie gospodarowalna przez programistę. W niej również pod odpowiednimi adresami muszą znaleźć się procedury obsługi przerwań.

Domyślnie, czyli po starcie procesora, stos programowy jest umieszczony na samym końcu pamięci (od adresu 0xFFFFFh). Wzrost wierzchołka stosu jest w kierunku adresów malejących, rośnie on więc z góry na dół. Jane nie posiada instrukcji specjalnych do obsługi stosu, więc programista sam musi dbać o dekrementację odpowiedniego wskaźnika przy odkładaniu na stos, oraz jego inkrementację przy ściąganiu.

Pozornie niedogodne może być operowanie jedynie na 4-bajtowych słowach (co wynika z organizacji pamięci oraz budowy jednostki arytmetycznzo-logicznej) gdyż intuicyjnie wydaje nam się, że pojedyncze bajty są wygodniejsze. Pozory jednak mylą, gdyż tak naprawdę obrane przez nas rozwiązanie pozwala dość elastycznie posługiwać się danymi jedynie przez dobranie odpowiednich masek. Nic nie stoi przecież na przeszkodzie aby jedną wartość przetwarzać w czterech cyklach w każdym z nich zajmując się tylko odseparowanym pojedynczym bajtem. A z drugiej strony, w porównaniu z rozdzielczością bajtową, mamy do dyspozycji cztery razy więcej pamięci. W szczególnych przypadkach tak duże rozmiary pozwalają na przetwarzanie pseudo-SIMD (*Single Instruction Multiple Data*).

4. Przerwania i komunikacja z urządzeniami wejścia/wyjścia

4.1. Przerwania

W kwestii obsługi przerwań zewnętrznych Jane współpracuje ze specjalnym, osobnym sterownikiem, który nie będzie tutaj omawiany gdyż nie mieści się w ramach projektu. Omówię jedynie rozwiązanie po stronie samego mikroprocesora.
Jane potrafi obsłużyć 8 różnych przerwań zewnętrznych. Numer aktualnie występującego przerwania jest rozpoznawany poprzez 3 najmłodsze bity rejestru FR, które są ustawiane przez zewnętrzny sterownik. Procesor nie zajmuje się również priorytetami, to też leży w kwestii układu dołączanego.

Pierwszych 8 słów pamięci to swoista tablica wektorów procedur obsługi przerwań. W każdym z nich znajduje się instrukcja skoku bezwarunkowego (JMP) pod adres zawierający początek odpowiedniej procedury.Adresy tych komórek (numery słów) odpowiadają numerom zgłoszonych przerwań.
Kiedy układ odpowiedzialny za przyjmowanie przerwań odbierze informację o wystąpieniu któregoś z nich, wysyła do Jane sygnały ustawiające flagę IA oraz numer przerwania na 3 ostatnich bitach w rejestrze flag. W takiej sytuacji możliwe są dwa scenariusze:
— jeśli flaga obsługi przerwań (IE) jest wyzerowana, to procesor zignoruje wystąpienie zgłoszenia,
— ustawiona flaga IE oznacza możliwość obsługi przerwań, w tej sytuacji jednostka kontrolna Jane wywołuje poprostu rozkaz CALL X, gdzie X jest numerem przerwania. Jednocześnie wyzerowany zostanie również bit IA w rejestrze flag.

Szczególnym przypadkiem przerwania jest sygnał RESET. Jest on do procesora podawany osobną linią i nie może być w żaden sposób zamaskowany. Jego wystąpienie powoduje ustawienie licznika rozkazów na wartość 0x10h, wyzerowanie wszystkich rejestrów oraz ustawienie bitu IE.

4.2. Komunikacja z urządeniami I/O

Obsługa urządzeń wejścia/wyjścia jest zorganizowana za pośrednictwem mechanizmu przerwań. Wymiana danych jest realizowana poprzez specjalne bufory znajdujące się w pamięci RAM począwszy od adresu 0x8h.

5. Instrukcje procesora Jane

Jak już wspomniano wcześniej Jane potrafi obsłużyć 16 różnych instrukcji. Jej lista opiewa cztery instrukcje arytmetyczne, 4 instrukcje logiczne, 5 instrukcji skoków (w tym skok ze śladem oraz powrót), instrukcję kopiującą dane oraz dwie instrukcje operujące na bankach rejestrów.

Ogólna struktura słowa rozkazowego wygląda następująco :

`IIIIAARR RRRRRRRR RRRRRRRR RRRRRRRR`

Najstarsze cztery bity zawierają kod jednego z 16 rozkazów, kolejne 2 przeznaczone są na określenie trybu adresowania. Dostępne są cztery (a właściwie 3) tryby :
— `00` : natychmiastowy,
— `01` : rejestrowy pośredni (kierunek *do rejestru*),
— `10` : rejestrowy bezpośredni,
— `11` : rejestrowy pośredni (kierunek *do pamięci*).
Pozostała część do argumentu jest zależna od rodzaju adresowania.Poniżej znajduje się lista wszystkich operacji wykonywanych przez procesor wraz z opisem słowa rozkazowego.

Oznaczenia używane na liście rozkazów:
— Ra, Rb, . . . : dowolny rejestr roboczy,
— # : stała wartość całkowita z określonego zakresu lub pojedynczy bit tej wartości,
— A,B, . . . : pojedynczy bit wskaźnika odpowiedniego rejestru roboczego,
— X : bity nieużywane.

5.1. Instrukcje arytmetyczne

mnemonik: ADD

format: ADD Ra,Rb,Rc

działanie: Suma. Rozkaz dodaje wartości zawarte w rejestrach Ra, Rb i umieszcza wynik w Rc. Jeśli wynik dodawania będzie zerem, zostanie ustawiona flaga ZF, w przypadku wystąpienia przeniesienia z najstarszej pozycji flaga CF.

struktura: `000010AA AAABBBBB CCCCCXXX XXXXXXXX`

mnemonik: SUB

format: SUB Ra,Rb,Rc

działanie: Różnica. Rozkaz odejmuje wartość rejestru Rb od wartości Ra i wynik umieszcza w Rc. Wpływ na flagi taki sam jak instrukcji ADD.

struktura: `000110AA AAABBBBB CCCCCXXX XXXXXXXX`

mnemonik: MUL

format: MUL Ra,Rb,Rc

działanie: Iloczyn. Rozkaz mnoży wartości zawarte w rejestrach Ra, Rb i umieszcza wynik w Rc. Jeśli wynik mnożenia będzie zerem, ustawiona zostanie flaga ZF.

struktura: `001010AA AAABBBBB CCCCCXXX XXXXXXXX`

mnemonik: DIV

format: DIV Ra,Rb,Rc,Rd

działanie: Iloraz. Rozkaz dzieli wartość zawartą w rejestrze Ra przez wartość w Rb po czym w rejestrze Rc umieszcza wynik dzielenia, a w Rd resztę. Jeśli reszta z dzielenia wyniesie 0, to ustawiona zostanie flaga ZF.

struktura: `001110AA AAABBBBB CCCCCDDD DDXXXXXX`

5.2. Instrukcje logiczne

mnemonik: AND

format: AND Ra,Rb,Rc

działanie: Iloczyn bitowy. Kolejne bity rejestru Ra są mnożone przez bity Rb, a wynik zostaje umieszczony w Rc. Jeśli wynikiem będzie zero, ustawiona zostanie flaga ZF.

struktura: `010010AA AAABBBBB CCCCCXXX XXXXXXXX`

mnemonik: OR

format: OR Ra,Rb,Rc

działanie: Suma bitowa. Kolejne bity rejestru Ra są dodawane do bitów Rb, a wynik zostaje umieszczony w Rc. Wpływ na flagi jak wyżej.

struktura: `010110AA AAABBBBB CCCCCXXX XXXXXXXX`

mnemonik: XOR

format: XOR Ra,Rb,Rc

działanie: Suma wykluczająca (suma modulo 2). Kolejne bity rejestru są Ra xorowane z bitami Rb, a wynik zostaje umieszczony w Rc. Wpływ na flagi jak wyżej.

struktura: `011010AA AAABBBBB CCCCCXXX XXXXXXXX`

mnemonik: SHIFT

format: SHIFT Ra,0/1

działanie: Instrukcja powoduje logiczne przesunięcie wybranego rejestru o jeden bit. Kierunek przesunięcia jest zależny od wartości drugiego parametru. Jeśli jest to 0 procesor wykonuje przesunięcię w prawo, a jeśli 1 to w lewo. Za każdym razem bit "wysunięty" zmienia odpowiednio wartość flagi CF.

struktura: `010010AA AAAXXXXX XXXXXXX# XXXXXXXX`

5.3. Instrukcje skoków

mnemonik: JMP

format: JMP #

działanie: Skok bezwarunkowy. Powoduje wpisanie argumentu do PC. Żadne flagi nie są modyfikowane.

struktura: `100000XX XXXX#### ######## ########`

mnemonik: JZ

format: JZ #

działanie: Skok warunkowy. Powoduje wpisanie argumentu do PC pod warunkiem, że flaga ZF jest ustawiona. Żadne flagi nie są modyfikowane.

struktura: `100100XX XXXX#### ######## ########`

mnemonik: JC

format: JC #

działanie: Skok warunkowy. Powoduje wpisanie argumentu do PC pod warunkiem, że flaga CF jest ustawiona. Żadne flagi nie są modyfikowane.

struktura: `101000XX XXXX#### ######## ########`

mnemonik: CALL

format: CALL #

działanie: Skok ze śladem. Odkłada na stos PC po czym wpisuje wartość argumentu do PC. Żadne flagi nie są modyfikowane.

struktura: `101100XX XXXX#### ######## ########`

mnemonik: RET

format: RET

działanie: Powrót ze skoku ze śladem. Powoduje wpisanie argumentu do PC wartości ściągniętej ze stosu. Żadne flagi nie są modyfikowane.

struktura: `110000XX XXXXXXXX XXXXXXXX XXXXXXXX`

5.4. Instrukcje pozostałe

mnemonik: MOVE

format: MOVE Ra, Rb

działanie: Dokonuje transferu danych z rejestru Ra do rejestru Rb. Żadne flagi nie są modyfikowane.

struktura: `110110AA AAABBBBB XXXXXXXX XXXXXXXX`

format: MOVE [Ra], Rb

działanie: Dokonuje transferu danych z komórki pamięci o adresie w Ra do rejestru Rb. Żadne flagi nie są modyfikowane.

struktura: `110101AA AAABBBBB XXXXXXXX XXXXXXXX`

format: MOVE Ra, [Rb]

działanie: Dokonuje transferu danych z rejstru Ra do komórki pamięci o adresie w Rb. Żadne flagi nie są modyfikowane.

struktura: `110111AA AAABBBBB XXXXXXXX XXXXXXXX`

format: MOVE #, Ra

działanie: Ładuje do rejestru Ra 20-bitową stałą. Żadne flagi nie są modyfikowane.

struktura: `110100AA AAAX#### ######## ########`

mnemonik: CRB

format: CRB

działanie: Przełącza aktywny bank rejestrów. Żadne flagi nie są modyfikowane.

struktura: `111000XX XXXXXXXX XXXXXXXX XXXXXXXX`

mnemonik: ZR

format: ZR

działanie: Zeruje wszystkie rejestry ogólnego przeznaczenia. Żadne flagi nie są modyfikowane.

struktura: `100000XX XXXXXXXX XXXXXXXX XXXXXXXX`

6. Jane assembler

Asembler Jane nie różni się znacznie składnią od znanego powszechnie asemblera procesorów 8086.

Dyrektywy kompilatora rozpoczynają sie od '.' i pierwszą zawsze jest `.jane`, która oznacza, że program został napisany dla naszego procesora. Kolejne dwie dyrektywy do `.begin` oraz `.end` Na początku każdej linii wewnątrz programu, znajduje się opcjonalna etykieta zakończona dwukropkiem (':'). Po niej następuje mnemonik operacji (dokładny ich opis znajduje się w poprzednim rozdziale).
Komentarz zaczyna sie od '//' i kończy wraz ze znakiem końca linii. Użycie jako parametrów samych nazw rejestrów oznacza tryb adresowania rejestrowy bezpośredni, z nawiadami kwadratowymi ('[',']') mamy do czynienia z adresowaniem rejestrowym pośrednim. Dla adresowania natychmiastowego parametrem są stałe wartości liczbowe.

A oto przykładowy program :

```
.jane // -> to jest komentarz

ORG 0x10h //kod umieszczony w pamięci począwszy od adresu 0x10h

.begin // <- poczatek programu

 JMP start // tam zaczyna sie wykonywalny kod program

 tablica : "To jest otwarty tekst." //tekst do zaszyfrowania,
      //22 bajty zajmujace 6 komorek pamieci
 klucz : "wrak" //klucz do szyfrowania/deszyfrowania tekstu, 1 komorka
 tablica_zaszyfrowana : "012301230123012301" //rezerwuje sobie miejsce
    //na tekst zaszyfrowany, 6 komorek

start:

 MOVE 1,R1 //potrzebne do inkrementacji/dekrementacji
 MOVE 6,R2 //ilosc wykonan petli

 MOVE klucz,R4 //adres klucza szyfrujacego
 MOVE [R4],R7 //ladujemy caly klucz do rejestru
```

```
 MOVE tablica,R5  //adres poczatku tablicy tekstu otwartego
 MOVE tablica_zaszyfrowana,R6 //adres poczatku tablicy tekstu zaszyfrowanego

petla:

 MOVE [R5],R3 //ladujemy 4 bajty tekstu
 XOR R3,R7,R3 //szyfrujemy XORujac z kluczem
 MOVE R3,[R6] //kopiujemy w zarezerwowane miejsce

 ADD R5,R1,R5 //inkrementacja adresu w tablicy wejściowej
 ADD R6,R1,R6 //inkrementacja adresu w tablicy wyjściowej

 SUB R2,R1,R2 //dekrementacja licznika pętli
 JZ koniec //kiedy R2=R2-1 bedzie rowne 0, koniec petli
 JMP petla

koniec:
 JMP start //zapętlenie, żeby procesor nie zaczął wykonywać przypadkowych operacji
.end // <- koniec
```

Oto fragment pamięci na początku programu :

```
 mem[17] 01010100011011110010000001101010 //tablica : "To j"
 mem[18] 01100101011100110111010000100000 //"est "
 mem[19] 01101111011101000111011101100001 //"otwa"
 mem[20] 01110010011101000111100100100000 //"rty "
 mem[21] 01110100011001010110101101110011 //"teks"
 mem[22] 01110100001011100000000000000000 //"t."

 mem[24] 00110000001100010011001000110011 //"0123"
 mem[25] 00110000001100010011001000110011 //"0123"
 mem[26] 00110000001100010011001000110011 //"0123"
 mem[27] 00110000001100010011001000110011 //"0123"
 mem[28] 00110000001100010011001000110011 //"0123"
 mem[29] 00110000001100010000000000000000 //"01"
```

Po zakończeniu prezentował się on następująco :

```
 mem[24] 00100011000111010100000100000001
 mem[25] 00010010000000010001010101001011
 mem[26] 00011000000001100001011000001010
 mem[27] 00000101000001100001100001001011
 mem[28] 00000011000101110000101000011000
```

```
mem[29] 00000011010111000110000101101011
```

Tekst został więc z powodzeniem zaszyfrowany.

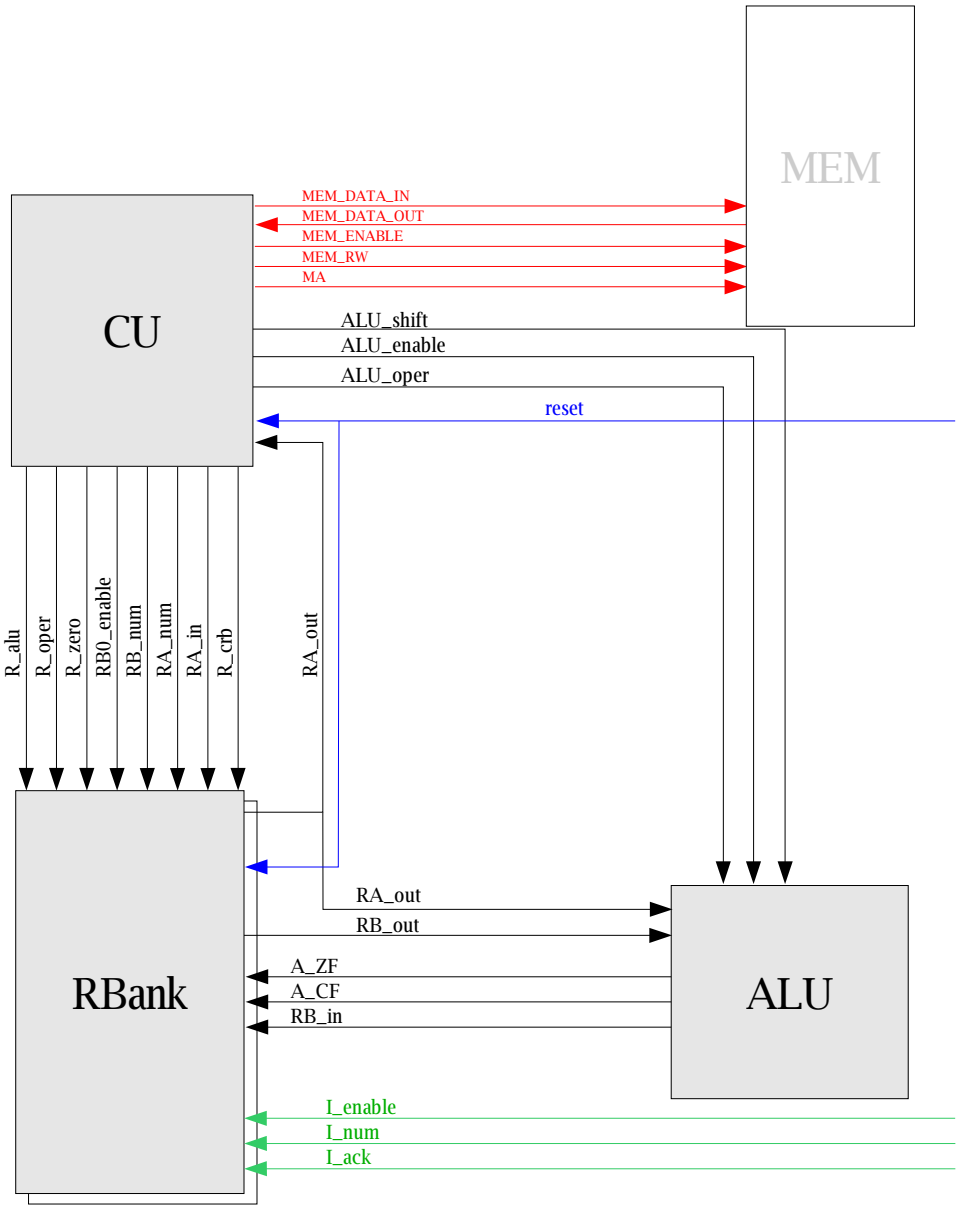
W powyższym przykładzie widać, że w związku z przyjętą konwencją pamięci operacyjnej pojawia się czasem problem jej fragmentacji. Projektując Jane postawiliśmy na jej prostotę i to jest jedna z konsekwencji. Biorąc jednak pod uwagę bardzo duży obszar dostępnej pamięci nie powinno to stanowić ograniczenia. Wymaga jedynie od programisty nieco większej troski o odpowiednią alokację obszarów danych. Problem oczywiście nie istnieje wogóle jeśli operujemy na danych 32 bitowych.

Podsumowanie

Postawione sobie zadanie realizowaliśmy z entuzjazmem i przyjemnością. Efektem wielu godzin naszej pracy stał się 32-bitowy procesor o klasycznej architekturze RISC, który nazwaliśmy Jane.
Jednostka została zaprojektowana w taki sposób, aby połączyć maksymalną prostotę z użytecznością. Jej zastosowanie ogranicza sie oczywiście jedynie do przetwarzania danych całkowitych, jednak bardzo duża przestrzeń adresowa i długie słowa danych pozwalają na swobodną ich obróbkę. Jak pokazał przykładowy program możliwe są też działania pseudo-SIMD.

Początkowo nasze plany opiewały znacznie więcej cech tego procesora. Miał obsługiwać adresowanie natychamistowe przy operacjach arytmetycznyh i logicznych, miało być więcej flag (między innymi przeniesienia cząstkowe co 8 bitów), pojawiały się wręcz pomysły implementacji przetwarzania potokowego i pamięci podręcznej (o architekturze Harvardzkiej). Oczywiście wszystkie te pomysły zostały odrzucone ze względu na brak czasu. Konsekwencją tego są między innymi "dziury" w rejestrze flag, które trzymają miejsce na przyszłe rozszerzenia. To co zostało opisane w niniejszym raporcie to podstawowa wersja Jane, która może być w każdej chwili rozszerzona o wymienione wyżej, jak również zupełnie nowe, cechy.

Dodatek A - Schematy



Schemat 1. Linie sygnałowe

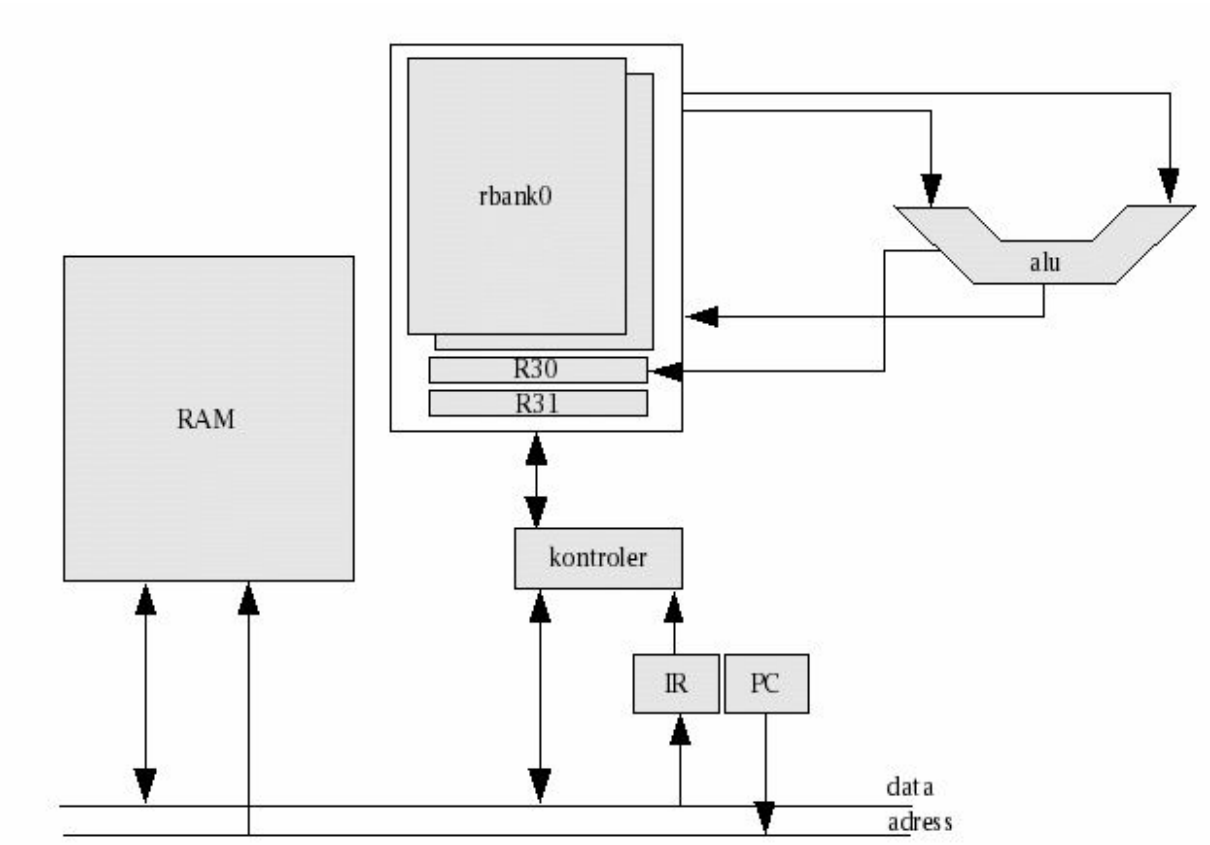
Poniżej omówione są oznaczenia zastosowane na schemacie.

`CU` – jednostka kontrolna, `RBank` – bank rejestrów, `MEM` – pamięć zewnętrzna, `ALU` – jednostka arytmetyczno-logiczna.

— `MEM_DATA_IN` – 32-bitowa magistrala zapisu pamięci,
— `MEM_DATA_OUT` – 32-bitowa magistrala odczytu z pamięci,
— `MEM_RW` – 1 = odczyt z pamięci; 0 = zapis do pamici,
— `MEM_ENABLE` – odblokowanie komunikacji z pamięcią,
— `MA` – 20-bitowa magistrala adresowa.

— `R_alu` – 1 = dane do zapisu w banku rejestrów pobierane są z magistrali `RB_out` (połączenie `ALU - RBank`) i numer rejestru z magistrali `RB_num`; 0 = dane do zapisu w banku rejestrów pobierane są z magistrali `RA_out`(połączenie `CU - RBank`) i numer rejestru z magistrali `RA_num`,
— `R_oper` – Sygnał sterujący rejestrami, 00 = odczyt, na linię `RA_out` wystawiany `Rx[RA_num]`, na linię `RB_out` wystawiany `Rx[RB_num]`.; 01 = odczyt, na linię `RA_out` wystawiany `R[RA_num]`; 10 = zapis, z linii `RA_in` zapis do `Rx[RA_num]`, gdy `R_alu`=1 zapis z linii `RB_in` do `Rx[RB_num]`; 11 = zapis, `Rx[RA_num]->Rx[RB_num]`
x - numer aktywnego banku rejestrów,
— `R_zero` – gdy pojawi się taki sygnał, rejestry są zerowane,
— `RB0_enable` – sygnał na tej linii odblokowuje operacje na rejestrach,
— `R_crb` – sygnał ustawiający aktywny bank rejestrów ogólnego przeznaczenia (0 lub 1),
— `RA_num, RB_num` – 5-bitowe magistrale przesyłające numer rejestru,
— `RA_in` – 32-bitowa magistrala przesyłająca dane do rejestrów (połączenie `CU - RBank`),
— `RB_in` – 32-bitowa magistrala przesyłająca dane do rejestrów (połączenie `ALU - RBank`),
— `RA_out` – 32-bitowa magistrala, którą wysyłane są dane do `CU` i `ALU` (połączenie `CU - RBank` i `CU - ALU`),
— `RB_out` – 32-bitowa magistrala, która służy do wysyłania danych do `ALU` (połączenie `ALU - RBank`),
— `I_enable` – sygnał informujący bank rejestrów iż zostanie ustawiona flaga IA (jeśli IE = 1) oraz wpisany numer przerwania w rejestrze R30,
— `I_ack` – przyjęcie przerwania przez kontroler,
— `I_num` – 3-bitowy numer przerwania wpisywany do R30.

— `ALU_shift` – sygnał sterujący kierunkiem przesunięcia logicznego (1 = lewo, 0 = prawo),
— `ALU_enable` – sygnał sterujący alu (1 = uruchom, 0 = wyłącz),
— `ALU_oper` – sygnały sterujące operacją wykonywaną przez alu, 000 = ADD, 001 = SUB, 010 = MUL, 011 = DIV, 100 = AND, 101 = OR, 110 = XOR, 111 = SHIFT
— `A_ZF` – sygnał ustawiający lub gaszący flagę ZF w R30.
— `A_CF` – sygnał ustawiający lub gaszący flagę CF w R30.



Schemat 2. Przepływ danych

`RAM` – pamięć zewnętrzna, `alu` – jednostka arytmetyczno logiczna, `IR` – *Instruction Register*, `PC` – *Program Counter*, `kontroler` – jednostka kontrolna, `data` – magistrala danych, `adress` – magistrala adresowa, `rbank0` – zerowy bank rejestrów ogólnego przeznaczenia, pod nim znajduje się `rbank1`, `R30` – rejestr flag (FR), `R31` – wskaźnik stosu (SP).

Bibliografia

[1] J. Biernat, „Architektura komputerów", Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław, 2001.

[2] W. Stallings, „Organizacja i architektura systemu komputerowego", Wydawnictwa Naukowo-Techniczne, Warszawa, 2000.

[3] Z. Pogoda, „Mikroprocesory RISC rodziny PowerPC", Wydawnictwo Pracowni Komputerowej Jacka Skalmierskiego, Gliwice, 1995.

[4] A. Dev, „CPU Design HOW-TO", Dokumentacja systemu Linux, 2002.

[5] D. Hyde, „CSCI 320 Computer Architecture Handbook on Verilog HDL", Lewisburg, 1997

[6] D. Hyde, „Using Verilog HDL to Teach Computer Architecture Concepts", Lewisburg, 1998

[7] S. Sutherland, „On-line Verilog HDL Quick Reference Guide", Portland, Oregon, USA