Jedynym nowym komponentem jest tutaj *rejestr adresów pośrednich*. Potrzebujemy go z tego względu, że musimy gdzieś trzymać adresy pośrednie pobrane z pamięci. W podobny sposób rejestr rozkazów przechowuje pobrane z pamięci instrukcje.

Dla uproszczenia rysunek 4.22 pomija wejścia zegarowe podłączone do wszystkich rejestrów i do pamięci. W przypadku prostego rejestru można założyć, że pobiera on nowe dane z każdym następnym taktem zegara, o ile tylko jego zezwolenie jest włączone. Podobnie, licznik programu oraz pamięć robią to, co ich sygnały sterujące każą im robić przy każdym takcie zegara. Wszystkie inne komponenty, jak na przykład selektory, są czysto kombinatoryczne i jako takie nie używają zegara.

Sterowanie ruchem

Teraz, kiedy już zaznajomiliśmy się ze wszystkimi wejściami i wyjściami, czas zbudować jednostkę zarządzającą ruchem. Przypatrzmy się kilku przykładowym wskazówkom na temat tego, jak powinna się ona zachowywać.

We wszystkich instrukcjach pojawia się pobieranie. Sterują nim następujące sygnały:

- `adres źródłowy` musi być ustawiony tak, by wybierał licznik programu.
- Pamięć musi być włączona, a sygnał odczytu/zapisu `r/`$\overline{\texttt{w}}$ musi być ustawiony na odczyt (1).
- Rejestr instrukcji musi być włączony.

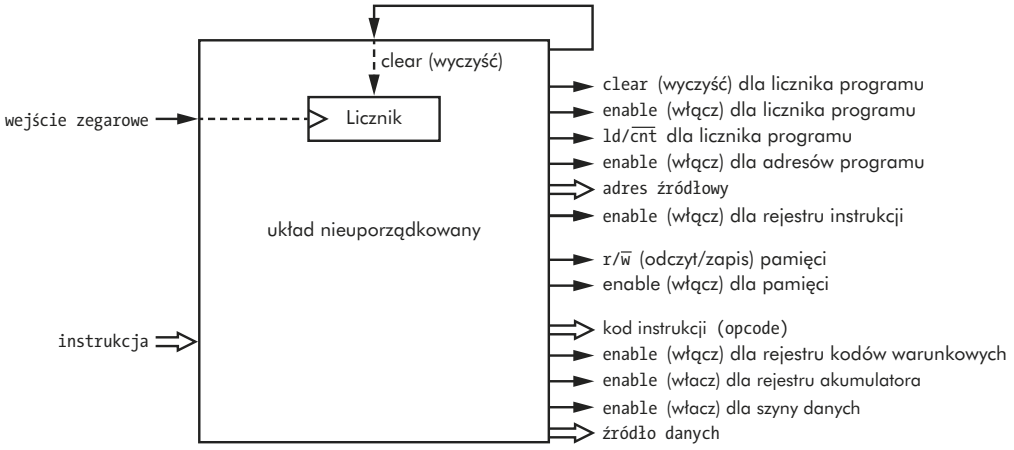
W drugim przykładzie chcemy zapisać zawartość akumulatora w lokacji pamięci wskazanej w adresie zawartym instrukcji – innymi słowy, z użyciem adresowania pośredniego. Pobieranie działa tak jak dotychczas.

- `adres źródłowy` musi być ustawiony tak, by wybierał rejestr rozkazów, co da nam adres zapisany w instrukcji.
- Pamięć jest włączona, a sygnał `r/w` jest ustawiony na odczyt (1).
- Rejestr adresów pośrednich jest włączony.

Zapisujemy zawartość akumulatora pod podanym adresem:

- `adres źródłowy` musi być ustawiony tak, by wybierał rejestr adresów pośrednich
- wyzwolenie na szynie danych musi być włączone
- Pamięć jest włączona, a `r/w` jest ustawione na zapis (0).
- Licznik programu podlega inkrementacji.

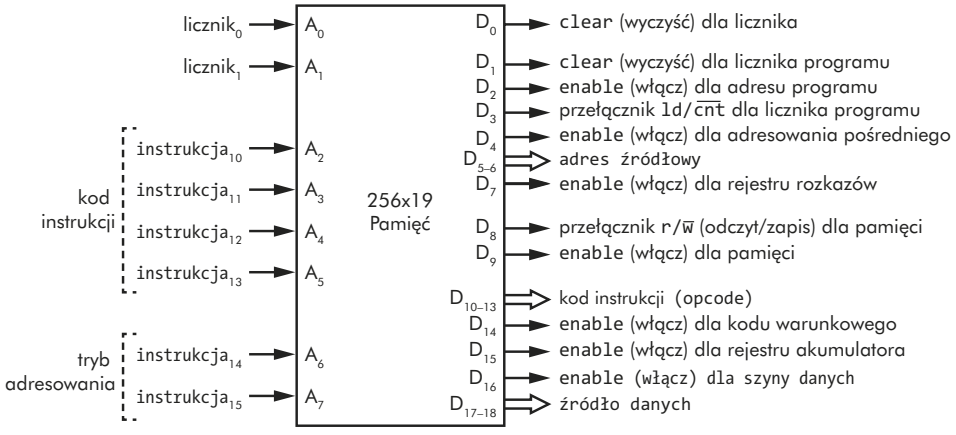
Ponieważ pobieranie i wykonywanie instrukcji zajmuje kilka kroków, potrzebujemy licznika, by wiedzieć, który z nich wykonujemy aktualnie. Zawartość licznika oraz kod instrukcji (*opcode*) i tryb adresowania zapisane w instrukcji są wszystkim, czego nam trzeba, by wygenerować niezbędne sygnały kontrolne. Potrzebujemy dwóch bitów licznika, ponieważ na wykonanie najbardziej skomplikowanych instrukcji potrzeba trzech kroków. Całość widoczna jest na rysunku 4.23.



Rysunek 4.23. Sterowanie ruchem za pomocą układu nieuporządkowanego (random logic)

To wielkie pudło pełne czegoś, co nazywamy *układem nieuporządkowanym* lub wręcz *przypadkowym* (niekiedy *logiką nieuporządkowaną* lub *logiką przypadku* – ang. *random logic*). Wszystkie układy logiczne, które widzieliśmy do tej pory mają rozpoznawalne, regularne wzory. Funkcjonalne bloki, takie jak selektory czy rejestry zbudowane są z mniejszych funkcjonalnych bloków w przejrzysty, dający się prześledzić sposób. Czasami jednak, na przykład teraz, kiedy trzeba zaprojektować naszą jednostkę „sterowania ruchem", dysponujemy zestawem wejść, który musi zostać odwzorowany na zestaw wyjść bez żadnej dającej się rozpoznać prawidłowości. Schemat elektroniczny urządzenia wygląda jak szczurze gniazdo połączeń – stąd nazwa *układ nieuporządkowany* (*random logic*).

Istnieje jednak jeszcze jeden sposób, na jaki może zaimplementować naszą jednostkę kontrolną. Zamiast plątaniny połączeń układu przypadkowego możemy użyć kawałka pamięci. Adresy zostaną utworzone z wyjść licznika oraz części kodu instrukcji (*opcode*) i trybu adresowania zapisanych w instrukcji. Całość pokazana jest na rysunku 4.24.



Rysunek 4.24. Sterowanie ruchem zbudowane na bazie pamięci

Każde 19-bitowe słowo pamięci ma układ taki, jak pokazano na rysunku 4.25.

Rysunek 4.25. Układ wewnętrzny słowa mikrokodu

To może wydawać się nieco dziwne. Z jednej strony to tylko następna maszyna stanowa zaimplementowana z użyciem pamięci zamiast plątaniny przewodów. Z drugiej strony to wygląda na rodzaj prostego komputera. Obie interpretacje są tutaj poprawne. Jest to maszyna stanowa, ponieważ komputery też są maszynami stanowymi. Jest również komputerem, ponieważ można to zaprogramować.

Przyjrzyjmy się fragmentowi *mikroinstrukcji* pokazanej na rysunku 4.26. Implementuje ona omawiane przez nas przykłady.

	DS_1	DS_0	DBE	ARE	CCRE	OP_3	OP_2	OP_1	OP_0	ME	R/$\overline{W}$	IRE	LD/$\overline{CNT}$	AS_1	AS_0	IAE	PCE	PCC	CC
Store	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	1
Indirect	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	1	0	0	0
Fetch	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	1	0	0	0
	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Rysunek 4.26. Przykład mikrokodu

Jak można się spodziewać, trudno uniknąć nadużywania dobrych pomysłów. Istnieją maszyny, które mają bloki *nanokodu* wykonujące bloki *mikrokodu* realizującego zbiór instrukcji.

Użycie ROM-u dla mikrokodu ma jakiś sens – inaczej trzeba by trzymać kopię mikrokodu gdzie indziej i mieć dodatkowy sprzęt, żeby załadować mikrokod. Są jednak sytuacje, kiedy usprawiedliwione jest użycie RAM-u – albo ROM-u i RAM-u razem. Niektóre procesory Intela mają programowalny mikrokod, do którego można pisać poprawki w celu naprawy błędów. Niektóre maszyny, takie jak te z serii HP-2100 miały *programowalną pamięć sterującą* (*writable control store*), która była mikrokodem RAM. Tych funkcjonalności można było użyć do rozszerzenia domyślnego zbioru instrukcji.

Maszyny, które dziś mają programowalny mikrokod, rzadko pozwalają użytkownikom na jego zmianę, a to z kilku przyczyn. Producenci chcą uniknąć sytuacji, w której użytkownicy polegają na własnym mikrokodzie, ponieważ gdy staną się od niego zależni, utrudni to znacznie wprowadzanie zmian przez producentów. Ponadto, wadliwy mikrokod może fizycznie uszkodzić maszynę. Mogłoby na przykład dojść do włączenia `enable` równocześnie na szynie pamięci i szynie danych. W rezultacie doszłoby do połączenia dwóch wyjść typu totem-pole i spalenia tranzystorów.

Zbiory instrukcji RISC i CISC

Projektanci tworzyli dawniej instrukcje dla komputerów, które wydawały się przydatne, ale powodowały, że całość maszyny stawała się skomplikowana. W latach 80. XX wieku amerykańscy informatycy David Patterson z Uniwersytetu w Berkeley i John Hennessey z Uniwersytetu w Stanford przeprowadzili analizę statystyczną programów komputerowych i odkryli, że większość skomplikowanych instrukcji była bardzo rzadko używana. Stali się pionierami paradygmatu projektowania maszyn w ten sposób, by zawierały tylko najczęściej używane instrukcje. Te rzadziej używane zostały zastąpione sekwencjami instrukcji prostszych. Maszyny tego typu nazwano RISC, od angielskiego *reduced instruction set computers*, tzn. *komputery o zredukowanym zbiorze instrukcji*. Starsze architektury nazwane zostały CISC, czyli *complicated instruction set computers*, tzn. *komputery o skomplikowanym zbiorze instrukcji*.

Jedną z cech charakterystycznych maszyn RISC jest ich *architektura ładowania-zapisu* (*load-store architecture*). To oznacza, że istnieją dwie kategorie instrukcji: jedna do dostępów do pamięci i druga do całej reszty.

Oczywiście wykorzystanie komputerów zmieniało się z czasem. Petterson i Hennesey przeprowadzili swoje statystyki, zanim jeszcze komputery były masowo używane np. do odtwarzania filmów czy dźwięku. Wciąż przeprowadza się podobne badania, dzięki czemu nowe instrukcje dodawane są do maszyn RISC – zawsze na podstawie ich realnego użycia. Dzisiejsze maszyny RISC są bardziej skomplikowane niż dawne maszyny CISC.

Komputer PDP-11 opracowany przez Digital Equipment Corporation jest przykładem bardzo wpływowej maszyny CISC. Dysponowała ona ośmioma rejestrami ogólnego przeznaczenia w miejsce pojedynczego akumulatora używanego przez nas w przykładach. Rejestry te mogły być używane również do adresowania pośredniego. Dodano również obsługę trybów *autoinkrementacji* i *autodekrementacji*. Pozwalały one na inkrementację i dekrementację rejestrów przed użyciem albo po nim. Dzięki temu można było budować bardzo wydajne programy. Powiedzmy, że chcemy na przykład skopiować *n* bajtów pamięci, od adresu źródłowego począwszy, do innego miejsca w pamięci, od lokacji podanej jako adres docelowy. Możemy umieścić adres źródłowy w rejestrze 0, adres docelowy w rejestrze 1 oraz licznik w rejestrze 2. Nie będziemy w tym miejscu śledzili konkretnych bitów, ponieważ nie ma potrzeby uczenia się zbioru instrukcji PDP-11. Tabela 4.5 pokazuje nam, jakie działania wykonywane za pomocą tych instrukcji.

Tabela 4.5. Program kopiowania pamięci dla PDP-11

Adres	Opis
0	Skopiuj zawartość tych lokacji pamięci, których adres zawarty jest w rejestrze 0, do lokacji pamięci o adresie podanej w rejestrze 1, a potem dodaj 1 do każdego rejestru
1	Odejmij 1 od zawartości rejestru 2 i porównaj wynik z liczbą 0
2	Jeśli rezultat nie równa się 0, skok do lokacji 0