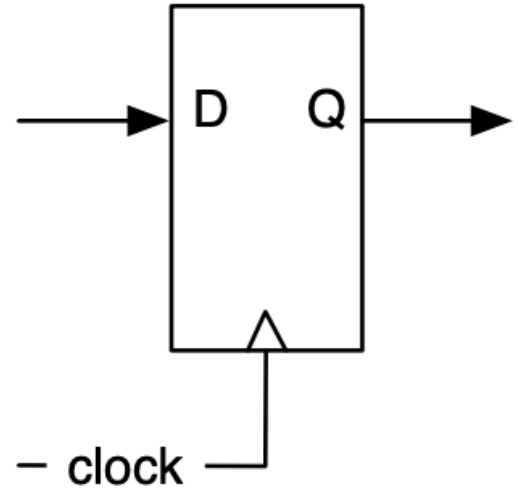
6 Sequential Circuit

Sequential Building Blocks

```
val q = RegNext(d)
```

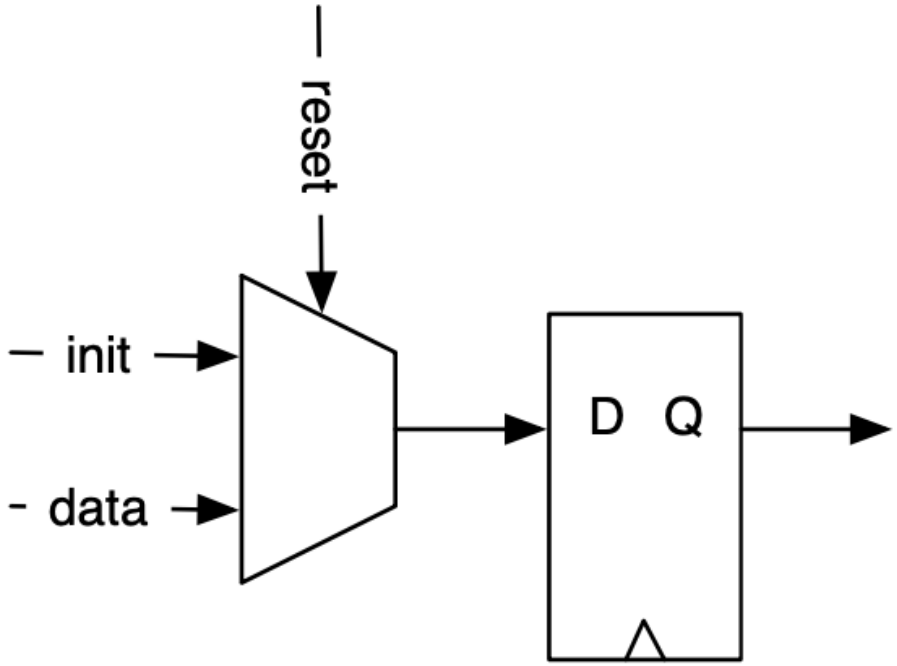
```
val delayReg = Reg(UInt(4.W))
delayReg := delayIn
```



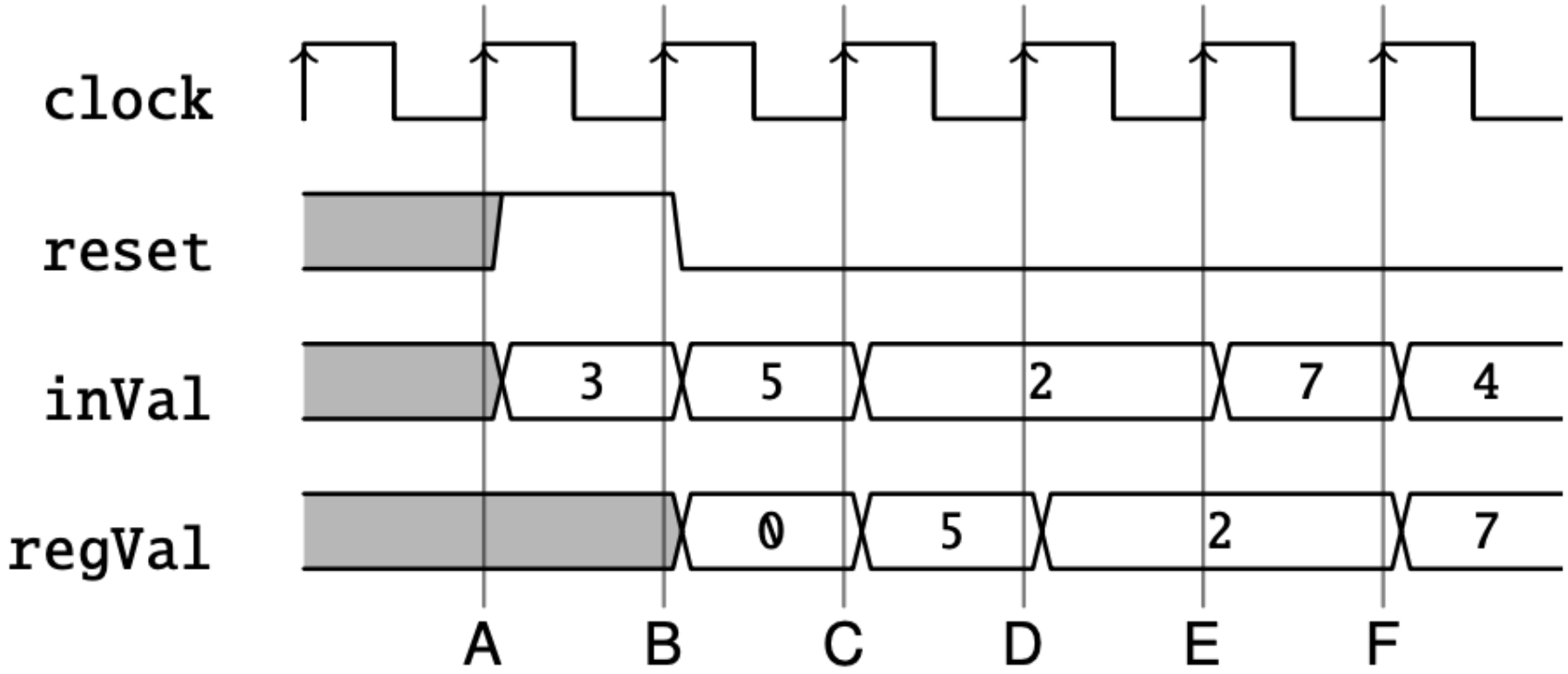
Register with Sync Reset

```
val valReg = RegInit(0.U(4.W))

valReg := inVal
```



Register with Sync Reset



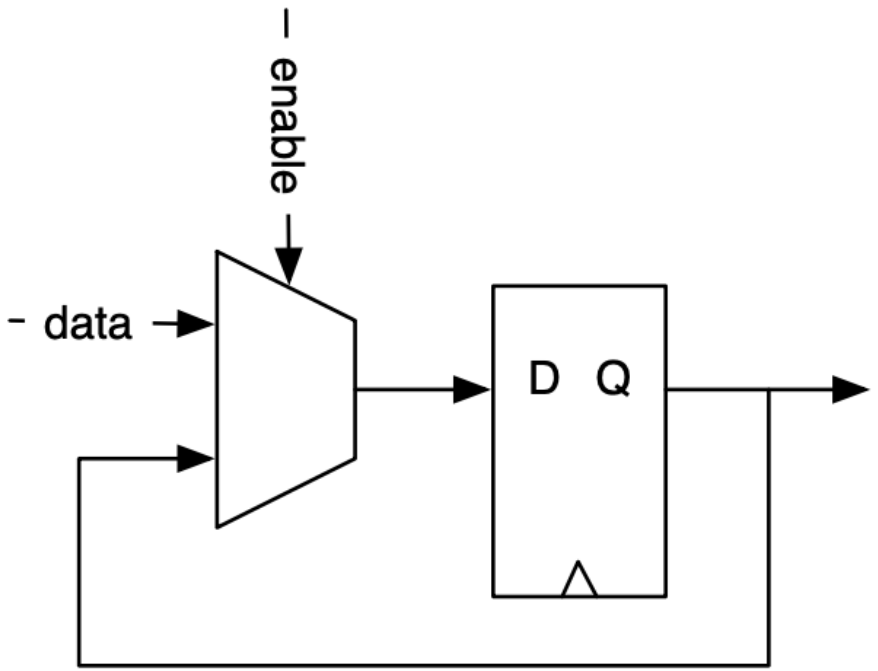
Register with Async Reset

Register with Async Reset

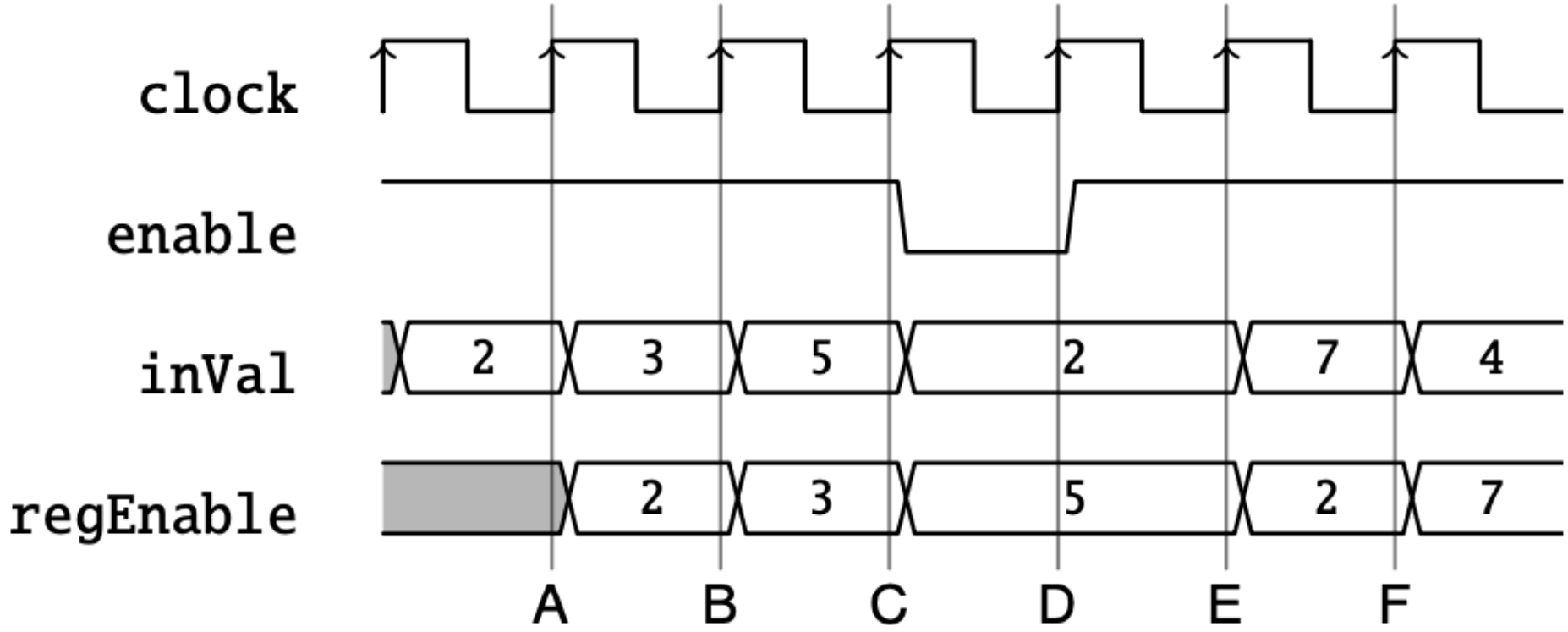
Register with an Enable Signal

```
val enableReg = Reg(UInt(4.W))

when (enable) {
  enableReg := inVal
}
```



Register with an Enable Signal



레지스터에 대해 활성화 신호를 사용하는 것은 매우 일반적이어서 ChiseI은 두 번째 매개변수가 활성화 신호인 RegEnable을 정의합니다.

```
val enableReg2 = RegEnable(inVal, enable)
```

활성화된 레지스터도 reset 할 수 있습니다.

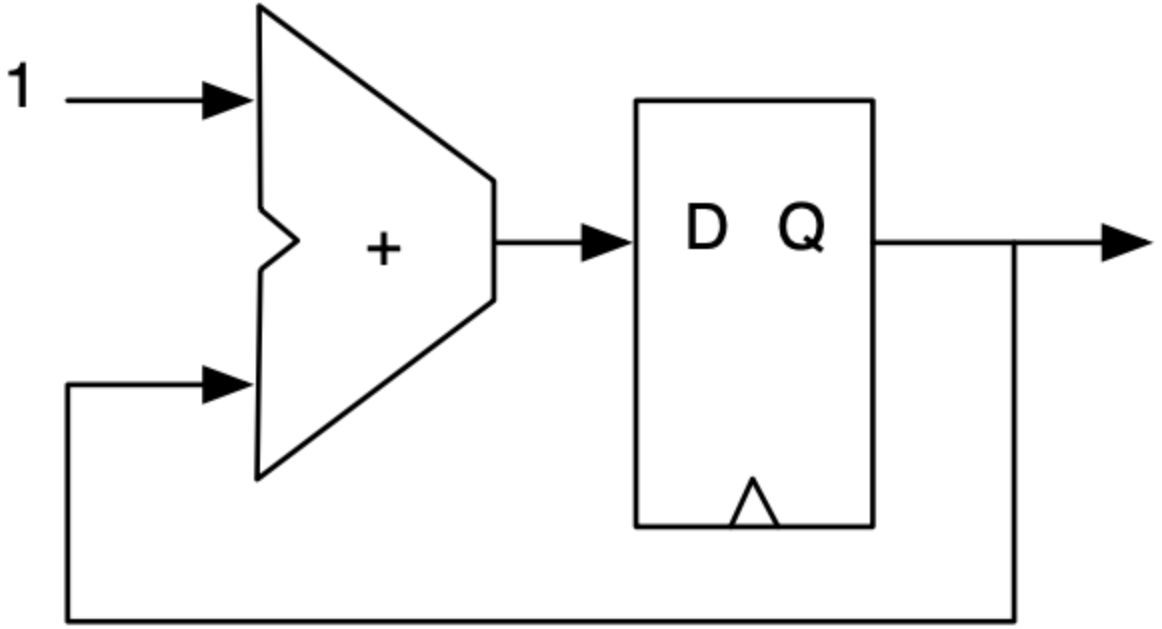
```
val resetEnableReg = RegInit(0.U(4.W))

when (enable) {
  resetEnableReg := inVal
}
```

RegEnable의 3개 매개변수 버전을 사용할 때 재설정 시 활성화 및 초기화 기능을 결합할 수 있습니다. 첫 번째 매개변수는 입력 신호, 두 번째 매개변수는 초기화 값,

6.2 Counters

가장 기본적인 순차 회로 중 하나는 카운터입니다. 가장 간단한 형태의 카운터는 출력이 가산기에 연결되고 가산기의 출력이 레지스터의 입력에 연결되는 레지스터입니다. 그림 6.6은 이러한 자유 실행 카운터를 보여줍니다.



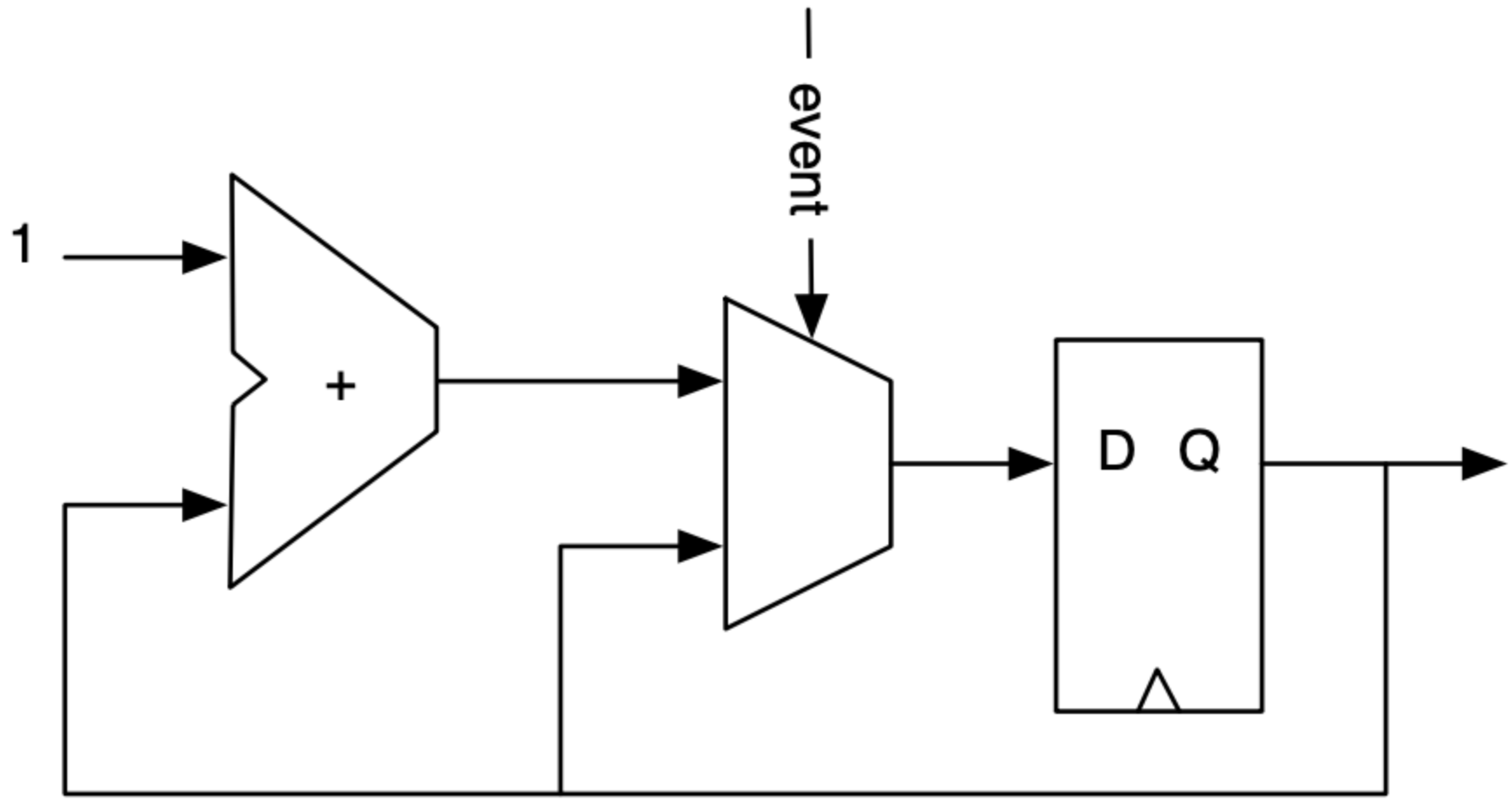


Figure 6.7: Counting events.

4비트 레지스터가 있는 free running 카운터는 0에서 15까지 계산한 다음 다시

6.2.1 Counting Up and Down

값을 세고 0으로 다시 시작하려면 카운터 값을 최대 상수(예: when 조건문)와 비교해야 합니다.

```
val cntReg = RegInit(0.U(8.W))

cntReg := cntReg + 1.U
when(cntReg === N) {
  cntReg := 0.U
}
```

카운터에 멀티플렉서를 사용할 수도 있습니다.

```
val cntReg = RegInit(0.U(8.W))

cntReg := Mux(cntReg === N, 0.U, cntReg + 1.U)
```

카운트다운을 하고 싶다면 카운터 레지스터를 최대값으로 재설정하는 것으로 시작하고 0에 도달하면 카운터를 해당 값으로 재설정합니다.

```
val cntReg = RegInit(N)

cntReg := cntReg - 1.U
when(cntReg === 0.U) {
  cntReg := N
}
```

더 많은 카운터를 코딩하고 사용함에 따라 카운터를 생성하는 매개변수가 있는 함수를 정의할 수 있습니다.

```
// This function returns a counter
def genCounter(n: Int) = {
  val cntReg = RegInit(0.U(8.W))
  cntReg := Mux(cntReg === n.U, 0.U, cntReg + 1.U)
  cntReg
}

---

// now we can easily create many counters
val count10 = genCounter (10)
val count99 = genCounter (99)
```

genCounter 함수의 마지막 명령문은 함수의 반환 값이며, 이 예에서는 계수 레지

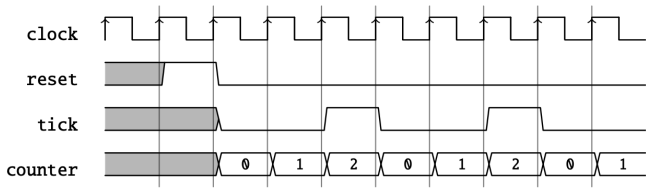
6.2.2 Generating Timing with Counters

이벤트를 계산하는 것 외에도 카운터는 종종 시간 개념을 생성하는 데 사용됩니다 (벽시계의 시간과 같은 시간). 동기 회로는 고정 주파수의 클록으로 실행됩니다. 회로는 해당 클록 틱에서 진행됩니다. 클록 틱을 계산하는 것 외에 디지털 회로에는 시간 개념이 없습니다. 클럭 주파수를 알면 치즐 "Hello World" 예제에서 볼 수 있듯이 특정 주파수에서 LED를 깜박이는 것과 같은 시간 이벤트를 생성하는 회로를 생성할 수 있습니다.

일반적인 관행은 회로에 필요한 주파수 ftick으로 단일 주기 틱을 생성하는 것입니다. 이 틱은 n 클록 사이클마다 발생합니다. 여기서 n = fclock/ftick이고 틱은 정확히 한 클록 사이클 길이입니다. 이 틱은 파생 클럭으로 사용되지 않고 주파수 ftick에서 논리적으로 작동해야 하는 회로의 레지스터에 대한 활성화 신호로 사용됩니다. 그림 6.8은 매 3 클럭 사이클마다 생성되는 틱의 예를 보여줍니다.

Figure 6.8: A waveform diagram for the generation of a slow frequency tick.

다음 회로에서는 0에서 최대값 N - 1까지 카운트하는 카운터를 설명합니다. 최대값에 도달하면 틱은 단일 주기에 대해 true이고 카운터는 0으로 재설정됩니다. 0에서 N - 1까지, 우리는 N 클록 사이클마다 하나의 논리적 틱을 생성합니다.




```
val tickCounterReg = RegInit(0.U(32.W))
val tick = tickCounterReg === (N-1).U

tickCounterReg := tickCounterReg + 1.U
when (tick) {
  tickCounterReg := 0.U
}
```

Figure 6.9: Using the slow frequency tick.

n 클록 주기마다 한 틱의 이 논리적 타이밍은 이 더 느리고 논리적 클록으로 회로의 다른 부분을 진행하는 데 사용할 수 있습니다. 다음 코드에서는 n 클럭 주기마다 1씩 증가하는 또 다른 카운터를 사용합니다.

```
val lowFrequCntReg = RegInit(0.U(4.W))
when (tick) {
  lowFrequCntReg := lowFrequCntReg + 1.U
}
```

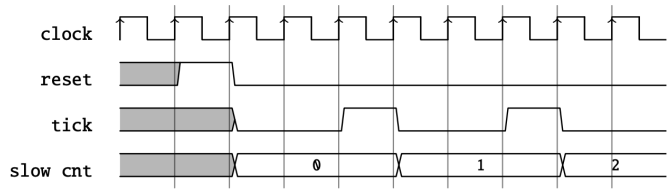


그림 6.9는 틱의 파형과 모든 틱(n 클럭 사이클)을 증가시키는 느린 카운터를 보여줍니다.

6.2.3 The Nerd Counter

우리 중 많은 사람들이 때때로 괴상한 사람이 된 것처럼 느낍니다. 예를 들어, 카운터/틱 생성의 고도로 최적화된 버전을 설계하려고 합니다. 표준 카운터는 하나의 레지스터, 하나의 가산기(또는 감산기), 비교기 등의 리소스가 필요합니다. 우리는 레지스터나 가산기에 대해 많은 것을 할 수 없습니다. 세어보면 비트 문자열인 숫자와 비교해야 합니다. 비교기는 비트 스트링의 0과 큰 AND 게이트에 대한 인버터로 구성될 수 있습니다. 0으로 카운트다운할 때 비교기는 큰 NOR 게이트로 ASIC의 상수에 대해 비교기보다 약간 저렴할 수 있습니다. 로직이 룩업 테이블에서 구축되는 FPGA에서는 0 또는 1과 비교하는 것 사이에 차이가 없습니다. 리소스 요구 사항은 업 카운터와 다운 카운터에서 동일합니다.

그러나 영리한 하드웨어 디자이너가 해낼 수 있는 트릭이 한 가지 더 있습니다. 카운트 업 또는 다운은 지금까지 모든 카운팅 비트에 대한 비교가 필요했습니다. N-2

6.2.4 A Timer

우리가 만들 수 있는 또 다른 형태의 타이머는 원샷 타이머입니다. 원샷 타이머는 주방 타이머와 같습니다. 분 수를 설정하고 시작을 누릅니다. 지정된 시간이 경과하면 알람이 울립니다. 디지털 타이머는 클록 사이클의 시간과 함께 로드됩니다. 그런 다음 0에 도달할 때까지 카운트다운합니다. 0에서 타이머가 완료되었다고 주장합니다.

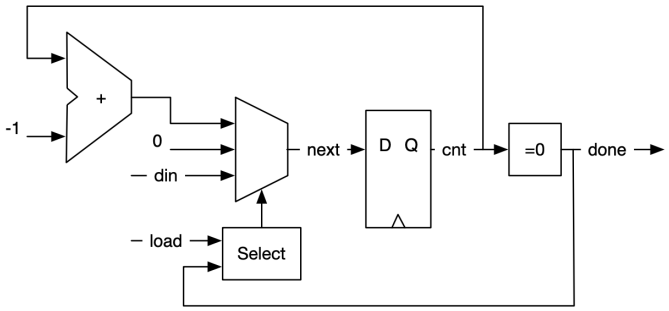


그림 6.10은 타이머의 블록도를 보여줍니다. load를 선언하여 레지스터에 din 값을 로드할 수 있습니다. 로드 신호가 선언 해제되면 카운트다운이 선택됩니다 (cntReg - 1을 레지스터 입력으로 선택). 카운터가 0

Listing 6.1은 타이머의 치즐 코드를 보여준다. 0으로 재설정된 8비트 레지스터 reg를 사용합니다. boolean 값은 reg를 0과 비교한 결과입니다. 입력 멀티플렉서의 경우 기본값 0을 사용하여 다음 와이어를 소개합니다. when/elsewhen 블록은 다음을 소개합니다. 선택 기능이 있는 다른 두 입력. 신호 부하는 감소 선택보다 우선합니다. 마지막 줄은 다음으로 표시되는 멀티플렉서를 레지스터 reg의 입력에 연결합니다.

```
val cntReg = RegInit(0.U(8.W))
val done = cntReg === 0.U

val next = WireDefault(0.U)
when (load) {
  next := din
} .elsewhen (!done) {
  next := cntReg - 1.U
}
cntReg := next
```

6.2.5 Pulse-Width Modulation

펄스 폭 변조(PWM)는 일정한 주기를 갖는 신호이며 해당 주기 내에서 신호가 높은 시간의 변조입니다.

그림 6.11은 PWM 신호를 보여줍니다. 화살표는 신호 기간의 시작을 가리킵니다. 신호가 높은 시간의 백분율을 듀티 사이클이라고도 합니다. 처음 두 주기에서 듀티 사이클은 25%, 다음 두 주기에서는 50%, 마지막 두 주기에서는 75%입니다. 펄스 폭은 25%와 75% 사이에서 변조됩니다.

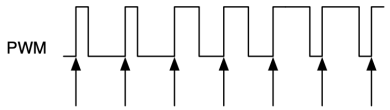


Figure 6.11: Pulse-width modulation.

```
def pwm(nrCycles: Int, din: UInt) = {
  val cntReg = RegInit(0.U(unsignedBitLength(nrCycles-1).W))
  cntReg := Mux(cntReg === (nrCycles-1).U, 0.U,
  cntReg + 1.U) din > cntReg
}

val din = 3.U
val dout = pwm(10, din)
```

재사용 가능한 경량 구성 요소를 제공하기 위해 PWM 생성기용 기능을 사용합니다. 이 기능에는 두 개의 매개변수가 있습니다. 하나는 클럭 주기(nrCycles)로 PWM을 구성하는 정수이고, 다른 하나는 PWM 출력 신호에 대한 듀티 주기(펄스 폭)를 제공하는 끌 와이어(din)입니다. 이 예에서는 멀티플렉서를 사용하여 카운터를 표현합니다. 함수의 마지막 줄은 카운터 값을 입력 값 din과 비교하여 PWM 신호를 반환합니다. Chisel 함수의 마지막 표현식은 반환 값입니다. 이 경우 비교 함수에 연결된 와이어입니다

또 다른 애플리케이션은 PWM을 사용하여 LED를 어둡게 하는 것입니다. 이 경우 눈은 저역 통과 필터 역할을 합니다. 위의 예를 확장하여 삼각 함수로 PWM 생성을 구동합니다. 그 결과 지속적으로 변화하는 강도의 LED가 탄생했습니다.

```
val FREQ = 100000000 // a 100 MHz clock input
val MAX = FREQ/1000 // 1 kHz

val modulationReg = RegInit(0.U(32.W))

val upReg = RegInit(true.B)

when (modulationReg < FREQ.U && upReg) {
  modulationReg := modulationReg + 1.U
} .elsewhen (modulationReg === FREQ.U && upReg) {
  upReg := false.B
} .elsewhen (modulationReg > 0.U && !upReg) {
  modulationReg := modulationReg - 1.U
} .otherwise { // 0
  upReg := true.B
}
```

6.3 Shift Registers

시프트 레지스터는 시퀀스로 연결된 플립플롭의 모음입니다. 레지스터의 각 출력(플립플롭)은 다음 레지스터의 입력에 연결됩니다. 그림 6.12는 4단계 시프트 레지스터를 보여줍니다. 회로는 각 클록 틱에서 데이터를 왼쪽에서 오른쪽으로 이동합니다. 이 간단한 형식에서 회로는 din에서 dout으로 4탭 지연을 구현합니다.

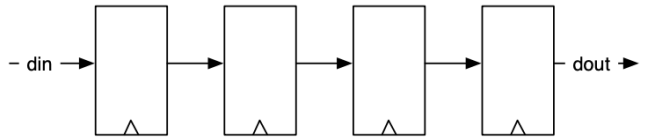


Figure 6.12: A 4 stage shift register.

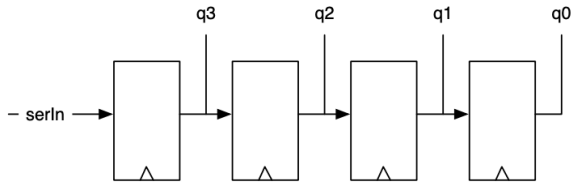
이 간단한 시프트 레지스터에 대한 치즐 코드는 (1) 4비트 레지스터 shiftReg를 생성하고, (2) 시프트 레지스터의 하위 3비트를 레지스터에 대한 다음 입력을 위한 입력 din과 연결하고, (3) 다음을 사용합니다. 레지스터의 최상위 비트(MSB)를 출력 출력으로 사용합니다.

```
val shiftReg = Reg(UInt(4.W))
shiftReg := Cat(shiftReg(2, 0), din)
val dout = shiftReg (3)
```

시프트 레지스터는 직렬 데이터에서 병렬 데이터로 또는 병렬 데이터에서 직렬 데이터로 변환하는 데 자주 사용됩니다. 섹션 11.2는 수신 및 송신 기능에 시프트 레지스터를 사용하는 직렬 포트를 보여줍니다.

6.3.1 Shift Register with Parallel Output

시프트 레지스터의 직렬 입력 병렬 출력 구성은 직렬 입력 스트림을 병렬 워드로 변환합니다. 이것은 수신 기능을 위한 직렬 포트(UART)에서 사용될 수 있습니다. 그림 6.13은 각 플립플롭 출력이 하나의 출력 비트에 연결된 4비트 시프트 레지스터를 보여줍니다. 4 클럭 사이클 후에 이 회로는 4비트 직렬 데이터 워드를 q에서 사용할 수 있는 4비트 병렬 데이터 워드로 변환합니다. 이 예에서는 비트 0(최하위 비트)이 먼저 전송되어 전체 단어를 읽고자 할 때 마지막 단계에 도착한다고 가정합니다.



6.3.2 Shift Register with Parallel Load

시프트 레지스터의 병렬 입력 직렬 출력 구성은 단어(바이트)의 병렬 입력 스트림을 직렬 출력 스트림으로 변환합니다. 이것은 전송 기능을 위한 직렬 포트(UART)에서 사용될 수 있습니다.

그림 6.14는 병렬 로드 기능이 있는 4비트 시프트 레지스터를 보여줍니다. 해당 기능에 대한 chisel 기술은 비교적 간단합니다.

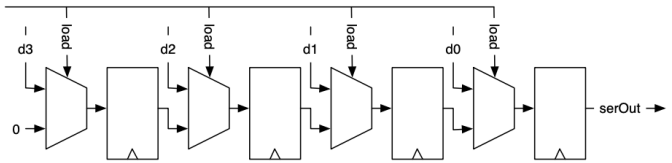
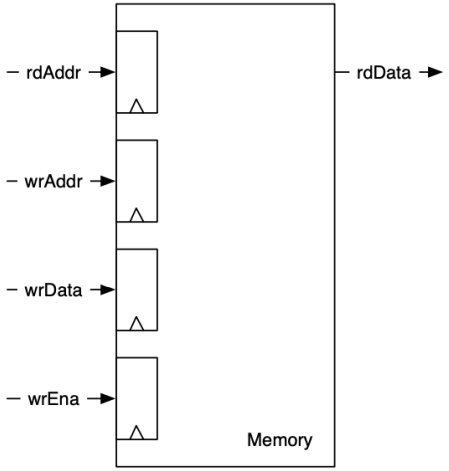


Figure 6.14: A 4-bit shift register with parallel load

6.4 Memory

Chisel a Reg of of Vec에서 메모리는 레지스터 모음에서 구축할 수 있습니다. 그러나 이것은 하드웨어 비용이 많이 들고 더 큰 메모리 구조가 SRAM으로 구축됩니다. ASIC의 경우 메모리 컴파일러는 메모리를 구성합니다. FPGA에는 블록 RAM이라고도 하는 온칩 메모리 블록이 포함되어 있습니다. 이러한 온칩 메모리 블록은 더 큰 메모리를 위해 결합될 수 있습니다. FPGA의 메모리에는 일반적으로 하나의 읽기 포트와 하나의 쓰기 포트가 있거나 런타임에 읽기와 쓰기 간에 전환할 수 있는 두 개의 포트가 있습니다.



온칩 메모리를 지원하기 위해 Chisel은 메모리 생성자 `SyncReadMem`을 제공합니다. 목록 6.2는 바이트 단위 입력 및 출력 데이터와 쓰기 가능으로 1KiB의 메모리를 구현하는 메모리 구성요소를 보여줍니다.
흥미로운 질문은 동일한 클럭 주기에서 읽은 동일한 주소에 새 값이 기록될 때 읽기에서 어떤 값이 반환되는지입니다. 우리는 메모리의 읽기 중 쓰기 동작에 관심이 있습니다. 세 가지 가능성이 있습니다. 새로 작성된 값, 이전 값 또는 정의되지 않음(이전 값의 일부 비트와 새로 작성된 데이터의 일부가 혼합될 수 있음). FPGA에서 사용할 수 있는 가능성은 FPGA 유형에 따라 다르며 때로는 지정할 수 있습니다. 치즐은 읽은 데이터가 정의되지 않았음을 문서화합니다.

```
class Memory() extends Module {
  val io = IO(new Bundle {
    val rdAddr = Input(UInt(10.W))
    val rdData = Output(UInt(8.W))
    val wrEna = Input(Bool())
```

새로 쓰여진 값을 읽고 싶다면 주소가 같다는 것을 감지하고 쓰기 데이터를 전달하는 전달 회로를 만들 수 있습니다. 그림 6.16은 전달 회로가 있는 메모리를 보여줍니다. 읽기 및 쓰기 주소는 쓰기 데이터의 전달 경로 또는 메모리 읽기 데이터 사이에서 선택하기 위해 쓰기 인에이블과 비교되고 게이트됩니다. 쓰기 데이터는 레지스터를 사용하여 1클록 주기만큼 지연됩니다.

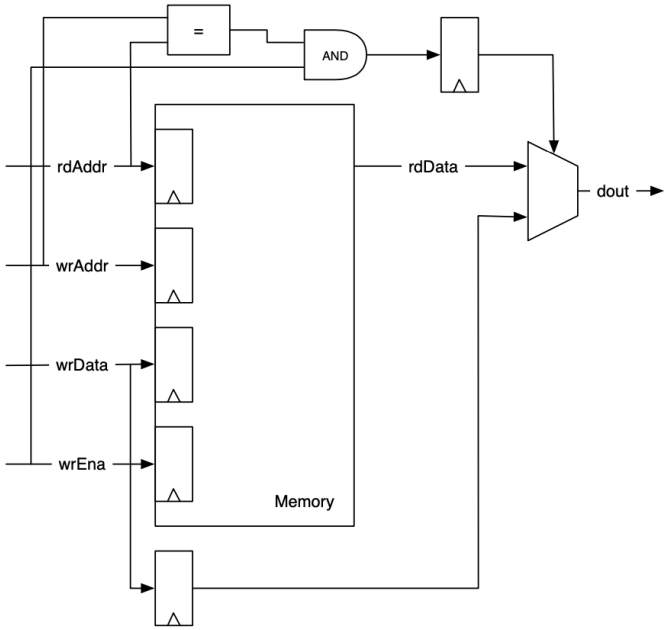


Figure 6.16: A synchronous memory with forwarding for a defined read-during-write behavior.

목록 6.3은 전달 회로를 포함하는 동기 메모리에 대한 치즐 코드를 보여줍니다. 다음 클록 사이클에서도 읽기

```
class ForwardingMemory() extends Module {
  val io = IO(new Bundle {
    val rdAddr = Input(UInt(10.W))
    val rdData = Output(UInt(8.W))
    val wrEna = Input(Bool())
    val wrData = Input(UInt(8.W))
    val wrAddr = Input(UInt(10.W))
  })
  val mem = SyncReadMem(1024, UInt(8.W))

  val wrDataReg = RegNext(io.wrData)
  val doForwardReg = RegNext(io.wrAddr === io.rdAddr && io.wrEna)

  val memData = mem.read(io.rdAddr)

  when(io.wrEna) {
    mem.write(io.wrAddr, io.wrData)
  }

  io.rdData := Mux(doForwardReg , wrDataReg , memData)
}
```


Chisel은 동기 쓰기 및 비동기 읽기가 있는 메모리를 나타내는 Mem도 제공합니

FPGA의 메모리는 2진 또는 16진 초기화 파일로 초기화할 수 있습니다. 파일은 해당 메모리에 있는 항목과 동일한 행 수를 가진 간단한 ASCII 텍스트 파일입니다. 각 문자는 단일 비트 또는 4비트를 나타냅니다. 일반적으로 이진 파일은 .bin 파일 확장자를 사용하는 반면 16진수 파일은 .hex를 사용합니다. loadMemoryFromFile을 사용하면 별도의 Verilog 파일이 생성되고 ChiselTest에서 작동합니다. 초기화는 readmemb 또는 readmemh에 대한 호출을 기반으로 합니다.

6.5 Exercise

마지막 연습에서 7-세그먼트 인코더를 사용하고 4비트 카운터를 입력으로 추가하여 디스플레이를 0에서 F로 전환합니다. 이 카운터를 FPGA 보드의 클록에 직접 연결하면 16개의 숫자가 모두 겹친 것을 볼 수 있습니다(7개의 세그먼트가 모두 켜집니다). 따라서 계산 속도를 늦출 필요가 있습니다. 500밀리초마다 단일 주기 틱 신호를 생성할 수 있는 다른 카운터를 만듭니다. 이 신호를 4비트 카운터에 대한 활성화 신호로 사용합니다.

생성기 함수로 PWM 파형을 구성하고 함수(삼각형 또는 사인 함수)로 임계값을 설정합니다. 위아래로 세어 삼각함수를 만들 수 있습니다. 몇 줄의 Scala 코드로 생성할 수 있는 조회 테이블을 사용하는 sinus 함수(섹션 10.3 참조). 변조된 PWM 기능을 사용하여 FPGA 보드에서 LED를 구동합니다. PWM 신호의 주파수는 얼마입니까? 드라이버가 실행하는 주파수는 무엇입니까?

```
val dout = WireDefault(0.U)

switch(sel) {
  is(0.U) { dout := 0.U }
  is(1.U) { dout := 11.U }
  is(2.U) { dout := 22.U }
  is(3.U) { dout := 33.U }
  is(4.U) { dout := 44.U }
  is(5.U) { dout := 55.U }
}
```

다음은 레지스터를 포함하는 조금 더 복잡한 회로입니다.

```
val regAcc = RegInit(0.U(8.W))

switch(sel) {
  is(0.U) { regAcc := regAcc}
  is(1.U) { regAcc := 0.U}
  is(2.U) { regAcc := regAcc + din}
  is(3.U) { regAcc := regAcc - din}
}
```